Enhancing Evolutionary Algorithm Performance with Knowledge Transfer and Asynchronous Parallelism

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Eric O. Scott
Master of Science
George Mason University, 2015
Bachelor of Science
Andrews University, 2011

Director: Sean Luke, Professor, and Kenneth A. De Jong, Professor Emeritus
Department of Computer Science

Summer Semester 2022
George Mason University
Fairfax, VA

Dedication

Guide me, Zeus, and thou, O Destiny,
to wheresoever you have assigned me.

—Cleanthes, 4th century B.C.E.

Acknowledgments

I am indebted foremost to my sagacious wife, Dr. Arianna Scott, for being my partner, teammate, support, and sounding board through the ups and downs of research life and the PhD journey, even while completing her own graduate program—and to my good friends Chris Greenley and Dr. Andrew Hoff for a years-long, indefatigable habit of "accountability group" meetings.

I am thankful to Dr. Mark Coletti and Prof. Katie Schuman for their mentorship and encouragement, and for the endless well of collaborative energy they have given in helping to think about applications of asynchronous algorithms. And to Dr. Jeff Bassett, who worked closely with me on much of the work in Chapter 5, and whose understanding of biological aspects of evolutionary theory and their potential implications for algorithm design is much deeper than mine.

I am especially grateful to Professor Kenneth De Jong, for continuing to advise me on this work as co-chair of my committee several years into his retirement, and for spending years helping to beat an initially inchoate research ambition down from "the 10,000-foot level" into a clear set of research questions. And to my chair, Professor Sean Luke, for his unremitting willingness to jump into the weeds to wrestle critically with any technical topic, and for having a sharp sense of when to tell me to "cut bait and start writing!" I am grateful to my committee for their supportive feedback—and particularly to Prof. Bill G. Kennedy for pouring carefully over the manuscript (though any errors that remain are entirely mine!). I'm also thankful to Dr. Stu Kauffman, for telling me many years ago to "always work on what you think is most important" when I was a wide-eyed intern at the Santa Fe Institute.

From MITRE Corporation, I am grateful to have received the funding and scheduling flexibility that I needed to complete a dissertation while working full time—and for a constant stream of exposure to applied research problems to hone my scientific intuitions and research-pitch skills against. I am especially grateful to Zoe Henscheid and Jessica Rajkowski for championing to get me all of the resources I asked for.

And finally I am deeply grateful for many things to my parents, Dr. Steve and Vivian Scott, who drove hundreds of miles to mow the lawn, plant the garden, seal our deck, cook, and clean dishes while I worked around the clock to finish the manuscript!

Table of Contents

List of Tables

List of Figures

Abstract

ENHANCING EVOLUTIONARY ALGORITHM PERFORMANCE WITH KNOWLEDGE TRANSFER AND ASYNCHRONOUS PARALLELISM

Eric O. Scott

George Mason University, 2022

Dissertation Director: Sean Luke, Professor, and Kenneth A. De Jong, Professor Emeritus

Search and optimization problems are widespread throughout business and engineering, but these computational problems are often complex enough that they must be approached with heuristic algorithms. Evolutionary algorithms (EAs) offer a very general framework for solving many of these tasks, but their computational complexity can be difficult to manage when they are applied to mature and large-scale classes of problems. From my experience working on EA applications, I became concerned about two EA efficiency challenges that have been inadequately addressed to date: 1) parallelization strategies for evolutionary algorithms routinely suffer from *idle CPU resources* that go unused, and 2) customizing evolutionary algorithms with the domain-specific prior knowledge that they require in order to solve useful problems often requires costly and time-consuming research programs.

In response to the idle-resources problem, I have engaged in a detailed study of asynchronous steady-state evolutionary algorithms (ASEAs) and their ability to better utilize large clusters of CPU resources. I studied issues of speedup, evaluation-time bias, and excess computational effort in ASEAs—concluding in particular that evaluation-time bias is a less severe problem for these algorithms than many practitioners have assumed.

Next, I have addressed the problem of prior knowledge in EAs by engaging in a broad

preliminary study of evolutionary knowledge transfer (EKT) and multi-task optimization. Motivated by natural examples of "innovation engines" that repurpose solutions to past tasks to find solutions to complex future tasks, I studied issues of transferability in different problem classes, negative transfer, and representation-based knowledge transfer approaches for evolutionary algorithms. My contributions in this area include proofs of a new set of no-free-lunch theorems for various types of transfer optimization, and several novel algorithms to address EKT challenges—including a many-source population-seeding algorithm that avoids negative transfer fairly easily, a multi-task Cartesian genetic programming approach, and a representation-learning algorithm that is able to learn and transfer genotype-phenotype maps across problem classes.

Chapter 1: Introduction

> The survival of the fittest is a slow method for measuring advantages. The experimenter, by the exercise of intelligence, should be able to speed it up.
>
> —Alan Turing [1950]

A very widespread and important swath of human activity is concerned with the closely related tasks of *search* and *optimization*. Almost all professions and domains of business and engineering involve daily efforts to reduce waste, increase performance, find novel courses of action, and to find accurate predictive models of phenomena in the natural and artificial worlds. Many of these problems are complex enough, however, that in practical cases they can only be solved with heuristic algorithms such as genetic and evolutionary algorithms (EAs), which trade away performance guarantees in exchange for being able to solve many problems adequately in a feasible amount of time. The performance of these algorithms is affected by a number of complex considerations. In this dissertation I study two of those considerations in particular: 1) the utilization rate of parallel computing resources when objective evaluation times vary, and 2) the more subtle problem of configuring algorithms with heuristic knowledge that can help to efficiently navigate problem-specific search spaces.

1.1 Evolutionary Computation

Evolutionary computation (EC) is one general class of heuristic approaches (and, more specifically, *meta*-heuristic approaches, which I will describe in Chapter 2) to designing probabilistic algorithms that can be used to solve search and optimization problems. This school of computing is best known for conforming to a bio-inspired metaphor, in which

algorithms solve problems through an iterative process of *variation* and *selection* that resembles Darwinian evolution in many respects.[1] The chief advantage that these evolutionary algorithms have over other algorithmic techniques is that they make few explicit assumptions about the type or structure of problems that they are solving—to the point that they are often effectively applied as "black box" algorithms which make relatively little use of problem-specific information [Doerr, 2020].[2] This means that, on the one hand, EAs offer a very general framework for performing search and optimization: they serve as a means to solve (or approximately solve) problems that are complex, combinatorial, non-linear, non-convex, non-differentiable, of mixed-type, NP-hard, or otherwise too poorly understood for a less general approach to be useful. It also means, however, that EAs risk performing inefficiently. In line with the traditional distinction that artificial intelligence (AI) draws between "strong" and "weak" methods of problem-solving, algorithms that utilize specialized knowledge about a problem or problem domain generally find better solutions and/or require fewer computational resources than highly general, one-size-fits-all procedures that make no use of such information [Laird et al., 1986, Newell and Simon, 1976].[3]

[1] That said, the metaphor is optional: the core motivating property of most evolutionary algorithms is the idea of *parallel search*, in which a set of multiple solution instances is iteratively updated to collaboratively explore (and exploit) an objective landscape. As with neural networks and other bio-inspired approaches to computation, authors vary a great deal in how far they go toward emphasizing the natural metaphor or drawing inspiration from biological research.

[2] For example, EAs typically do not assume that the function to be optimized is continuous, or that its derivative is known (which is required by the gradient descent optimizers so widely used in neural networks); nor do they assume, in a control context, that the system to be controlled has linear dynamics, or that the function to be optimized is described by a quadratic expression (which is required by the linear-quadratic regulator of optimal control theory).

[3] Alan Turing [1950] was the first to conceive of evolutionary algorithms, and also the first to observe that because they proceed primarily through probabilistic sampling of a solution space (i.e., mutation), they may often be inefficient. This concern about EA inefficiency echoes to this day—particularly among advocates of reinforcement learning (such as Sutton and Barto [2018], p. 8). Recently, however, the ease with which EAs can be scaled and parallelized on large compute clusters—as well as their ability to handle complex search problems such as neural architecture search for deep learning—has helped to allay these concerns and to renew interest in EAs within the broader machine learning community [Elsken et al., 2019, Salimans et al., 2017].

1.2 Prior Work on Managing EA Efficiency

In my experience applying evolutionary algorithms to various simulation problems in economic regulation [Scott et al., forthcoming], neuroscience [Chen et al., 2021, Venkadesh et al., 2018, Zou et al., 2021], robotics [Scott et al., in press], and agent-based modeling [D'Auria et al., 2020], a key challenge has been managing the computational complexity that evolutionary methods tend to entail. Many of these problems involve high-dimensional search spaces and objective functions that require complex simulations to be executed that take a long time to run (even with the help of GPU acceleration). Problems of this kind seem to be particularly commonplace in mature projects that have moved beyond toy problems and are ready to begin solving useful and detailed instances of a real-world class of problems [Harada and Alba, 2020]. As both the need for and access to high-performance computing (HPC) resources continue to proliferate in the 21st century, the kinds of problems that EAs are applied to increasingly tend to be computationally intensive in this way.

The EC research community has used several broad strategies to improve on the efficiency of evolutionary algorithms for computationally expensive applications. These are illustrated by a hierarchical diagram in Figure 1.1. Like other meta-heuristic algorithms—of which evolutionary algorithms are the largest and most diverse subfamily [Blum et al., 2011, Stork et al., 2020]—EAs can be configured in a wide variety of different ways using various parameters, subcomponents, and design decisions.

Some of these decisions may improve the algorithm's ability to find good solutions while using *fewer queries* to the objective function. This can be achieved either by utilizing cross-cutting, "meta-level" insights that apply broadly to many domains [Back, 1994, Blickle and Thiele, 1996], or by applying deep human analysis and/or extensive parameter sweeps to design domain-specific search strategies for narrow application areas (such as traveling salesmen problems, real-valued optimization, or specific subsets within these domains) [Burke et al., 2010, Hutter et al., 2009, López-Ibáñez et al., 2016, Ochoa et al., 2015].

Other design decisions can improve performance by reducing the *cost of each query* to the

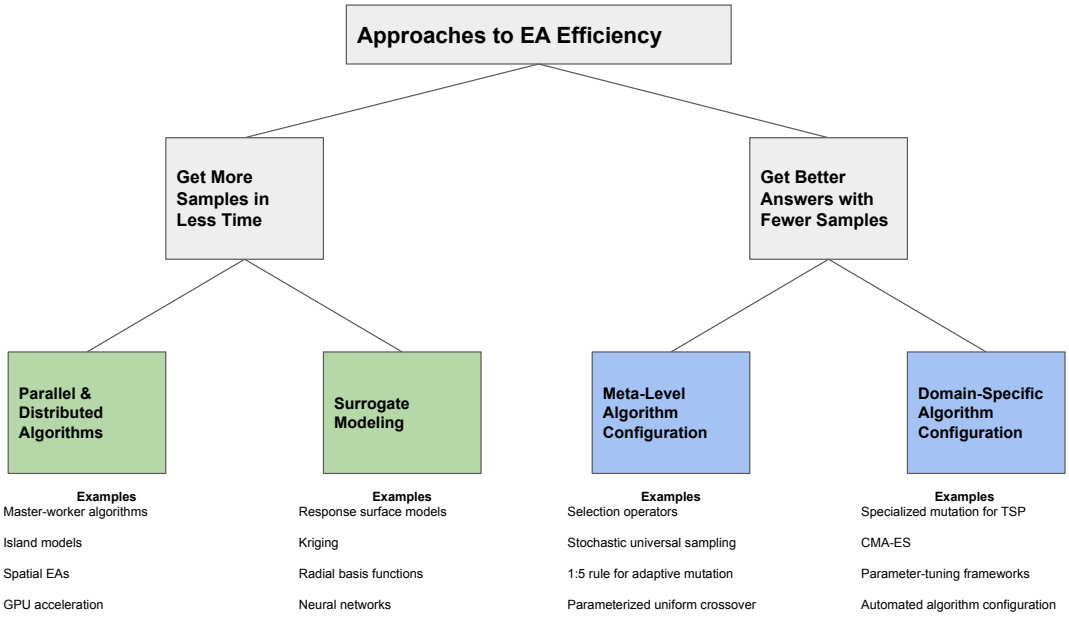

Figure 1.1: A high-level breakdown of the strategies the community has used to address the efficiency of evolutionary algorithms.

objective function, so that a greater number of samples can be obtained by the algorithm in the same period of time. The most significant approaches of this kind fall into two categories. Parallel evolutionary algorithms (PEAs) [Alba, 2005, Alba and Tomassini, 2002, Harada and Alba, 2020] use multiprocessing and/or distributed processing to exploit the "embarrassingly parallel" structure of EAs to evaluate many search points simultaneously. Surrogate models, meanwhile, explicitly build a statistical model of an objective function while it is being optimized, so that some queries can be replaced by much lower-cost queries to the surrogate model [Alizadeh et al., 2020, Jin, 2005]. These two approaches—PEAs and surrogate modeling—are complementary and are often used together.

1.3 Motivating Problems

In my work on EA applications, I have become concerned with two key challenges that have been inadequately addressed by prior research:

1. First, traditional parallelization strategies for evolutionary algorithms (like those indicated on the left-hand-side of Figure 1.1) suffer from an omnipresent **idle time problem**: at each generation, all the processors must synchronize and wait for the individual (or batch of individuals) that take the longest to evaluate to complete before evolution can proceed. In some cases this can lead to a large fraction of available CPU resources being left idle while an EA runs—especially in cases where numerous processors are used, as in modern high-performance computing clusters.

2. Second, performing domain-specific research to customize evolutionary algorithm components to a problem class (like in the examples listed on the right-hand-side of Figure 1.1) is often an essential step in being able to solve useful problems. But research programs of this kind are **costly and time-consuming**, and the results of such efforts usually **fail to generalize** to new domains (or even to new problems within the domain of interest). This situation is analogous to the widely lamented problem of generalization and robustness in contemporary machine learning [Schölkopf et al., 2021].

In this dissertation, I investigate solutions to each of these challenges: specifically **asynchronous steady-state evolutionary algorithms** (ASEAs), and **evolutionary knowledge transfer** (EKT), respectively. These two sets of algorithmic approaches are complementary, in the sense that they arise from separate theories and can be used independently of (or in combination with) one another. But each of these methods addresses a major bottleneck in the efficiency of EAs as they are currently applied.

1.4 Proposed Solutions

My research has taken me first to perform a thorough analysis of ASEAs and variations of them that respond to aspects of the idle-time problem. Ultimately, however, improving the efficiency of EA parallelization alone can only go so far toward addressing the computational

costs that are involved in solving complex search and optimization tasks. In the latter portion of this dissertation, then, I turn to the burgeoning field of EKT as a potential means of further improving EA performance. This second portion of my work here is more speculative and preliminary than the ASEA results, but has given me the opportunity to address several fundamental open questions that surround how and when knowledge transfer might play a significant role in evolutionary computation.

1.4.1 Asynchronous Steady-State EAs

Asynchronous steady-state EAs have risen in popularity in recent years as a solution to the idle-time problem. In contrast to *generational* EAs—which process the entire population at once in each generation—ASEAs follow a steady-state population model, in which single solutions are generated, evaluated, and integrated into the population on an individual basis. By performing steady-state evolution asynchronously across multiple processors, ASEAs can make better use of parallel computing resources than generational algorithms when the time it takes to evaluate solutions is relatively long and varies.

The asynchronous steady-state model has only recently begun to become widely used, however, particularly as problems involving computationally expensive objective functions and large parallel computing environments with many processors have increased in popularity [Coletti et al., 2020, 2021, Durillo et al., 2008, Harada et al., 2020, Yagoubi, 2012, Yagoubi et al., 2011]. ASEAs bring the advantage of increased processing throughput during evolution. As a side effect, however, the search trajectories of ASEAs differ from their generational counterparts in complex and difficult-to-analyze ways. In this dissertation, I study three significant open questions that are of potential concern to practitioners who apply ASEAs: namely 1) the problem of how much *speedup* an ASEA may be expected to produce (since reducing idle CPU resources may not always lead to better solutions), 2) the problem of *evaluation-time bias*, in which ASEAs are sometimes biased toward evolving solutions that are computationally inexpensive to evaluate (in addition to or instead of high in quality), and 3) the problem of *excess computation*, in which the additional evaluations that an

ASEA can perform with recovered resources may be useless on certain problems (ex. because the algorithm must still wait for longer-evaluating individuals to complete processing before progress can be made). I have published most of these results in peer-reviewed conferences and journals [Scott and De Jong, 2015, 2016b, Scott et al., 2021, in press], and present them here in a freshly integrated form, along with some new analytical and experimental results.

1.4.2 Evolutionary Knowledge Transfer

By far the majority of evolutionary algorithms' uses are *single-task* applications: in which an algorithm begins with some human-configured baseline input, executes iteratively for a number of steps until a stopping condition is reached, and then returns the best answer it has found so far. When a new problem instance needs to be solved, a single-task algorithm is reinitialized to its starting point, and uses zero information from prior experience on other problems.

Evolutionary knowledge transfer refers to evolutionary algorithms that solve multiple distinct tasks (either in sequence or in parallel), gathering information from one or more tasks that it uses to alter its heuristic approach while solving one or more additional, related tasks. Until just a few years ago, transfer-based EAs were rarely used and had never been deeply studied. Recently, however, a number of authors have begun to systematically investigate the challenges that surround the idea of retaining and reusing experience in evolutionary algorithms—building on a few seminal algorithms invented by Gupta et al. [2016c], Kelly and Heywood [2017], Nguyen et al. [2015b], and others.

At a broad, speculative level, EKT can be motivated by observing real-world instances where organizations face multiple optimization tasks that share a clear and obvious similarity (such as when a t-shirt factory needs to solve job-shop scheduling problems for different seasons of the year [Zhang et al., 2021b]), or it can be motivated by appealing to shared causal principles that underlay some broad domain or environment (such as robotics). A third provocative inspiration for evolutionary approaches to knowledge transfer arises from observations of the reuse of information in biology and technology: where serendipitous

repurposing of solutions to highly disparate problems and niches has often created breakthroughs and innovations [Arthur, 2009, Lenski et al., 2003]. *Innovation engines* of this kind [Nguyen et al., 2015b] have the potential to take an indirect route to solving problems, potentially arriving at complex solutions to problems that would be very difficult to approach directly [Stanley and Lehman, 2015].

These three narratives vary in the view of EKT that they lead to, and in their degree of speculation or optimism about the potential of knowledge transfer methods to broadly improve the state of the art for search and optimization. But all three lead to a similar array of basic challenges and open problems that, because they are poorly understood, complicate attempts to introduce knowledge reuse into evolutionary computation.

In this dissertation, I study three significant classes of these open questions that lie at the heart of the young field of evolutionary knowledge transfer and its feasibility: 1) the question of what kinds of *problem classes* knowledge transfer is a useful paradigm for, 2) the challenge of *negative transfer* and the associated task of *knowledge source selection*, and 3) how *knowledge representations* can be used to encode generalizable knowledge to transfer across tasks (and more specifically how such knowledge can be encoded in *solution representations*). I have published some of these preliminary findings as peer-reviewed workshop and conference papers [Scott and Bassett, 2015, Scott and De Jong, 2017, 2018]—but many of them are presented here for the first time.

1.5 Contributions and Roadmap

The remainder of this dissertation will present a number of studies that I have conducted on these two broad and complementary facets of evolutionary computation.

After an extensive background discussion in Chapter 2 covering both areas and presenting concrete research questions for each, I will turn in Chapter 3 to analyzing asynchronous evolutionary algorithms. My primary contributions in this area include an analytic proof of the expected performance speedup of an ASEA under restricted conditions, and an empirical

study of evaluation-time bias in ASEAs. I conclude that evaluation-time bias is less prevalent in most applications than practitioners have tended to fear, and that its effects are mostly confined to the initialization phase of the algorithm.

Another contribution I offer toward understanding asynchronous EAs is an analysis (not previously published) of the effects that different initialization strategies have on these algorithms' behavior—as it turns out that ASEAs are especially sensitive to the choice of initialization method in a way that traditional EAs are not. I also investigate one example of a class of hybrid algorithms that are only partly asynchronous—specifically the *quasi-generational EA*, which has been previously proposed in the literature [Durillo et al., 2008, Fonseca and Fleming, 1998]—and show that prior work that hypothesized that it has advantages over the classical ASEA was mistaken.

Finally, I present recent results that investigate a specialized selection operator that can be used with the ASEA to improve its performance on applications where high-quality solutions are associated with longer evaluation times of candidate solutions.

With the ASEA study complete, I then turn in Chapter 4 to the two related problems of transferability in problem classes and the issue of negative transfer. Here I prove a series of novel no-free-lunch theorems, showing analytically and empirically for the first time that the general principle that no algorithm can outperform other algorithms on average across all possible problems ("no free lunch") does indeed still apply when considering various types of knowledge transfer in optimization. I focus for the remainder of this chapter on instance-based transfer in a sequential setting (which has been somewhat neglected by the literature's recent trend toward focusing on multi-task transfer instead), demonstrating problem classes that it works well on, and proving a complexity result on the leading-ones problem that clarifies the conditions under which sequential knowledge transfer can reduce an EA's running time on this problem from $O(n^2)$ to $O(n \log n)$. To my knowledge, this is the first asymptotic runtime result that has been proven for transfer optimization. In this chapter I also contribute a many-source variation of instance-based transfer—where a repertoire of knowledge from many source tasks is used to solve a single target task—and

show that in some cases this can offer a very effective antidote to the widespread problem of negative transfer.

In Chapter 5, then, I contribute two new algorithms for evolutionary knowledge transfer, demonstrating that both of them are able to learn and transfer solution *representations*—one in the context of real-valued vector optimization problems, the other in a genetic programming context. These approaches serve to mitigate the limitations of most EKT methods, which commonly rely on the transfer of low-level solution instances which generalize poorly to new tasks.

With all the details covered and explained, I turn briefly in Chapter 6 to some closing reflections on the significance of both of these fields—asynchronous evolution and knowledge transfer—and how the results I present here contribute to their respective trajectories.

Lastly, because software tends to take a backstage role in written science, a silent contribution of this dissertation has been many contributions to the ECJ software framework [Scott and Luke, 2019], and also a novel EC software framework—the Library for Evolutionary Algorithms in Python (LEAP)—which I have developed in collaboration with Coletti et al. [2020] in response to the many limitations and inflexibilities of existing EC software frameworks. I have presented LEAP at peer-reviewed workshops and at formal talks at NASA Jet Propulsion Laboratory and MITRE Corporation, among other venues.

Chapter 2: Background

In this chapter I aim to give a thorough overview of the state of the art in asynchronous evolutionary optimization and evolutionary knowledge transfer (EKT), respectively. I will give special attention to several key research challenges that both fields face, but I will not attempt an entirely comprehensive survey of either area. Because these topics are somewhat independent, I will first survey parallel EAs and prior work on asynchronous interpretations of them in section 2.1, and then turn separately to EKT in section 2.2. Both sections will conclude with a set of research questions that the research in the remaining chapters of this dissertation is built around.

2.1 Asynchronous Evolutionary Optimization

In this section I discuss how an asynchronous steady-state computation model for evolutionary algorithms arises naturally from the need to reduce idle computing resources on parallel architectures. While there are other reasons one might be interested in asynchronous EAs [Kim, 1994, Martin et al., 2015, Sambo et al., 2020], making efficient use of (massively) parallel computing environments is the most important, and has become an especially burning problem in recent years, as HPC clusters become the norm rather than the exception in many applications. Most of the applications I personally work with, moreover, involve expensive fitness evaluations and simulations—and thus benefit greatly from parallel and distributed environments, where asynchronous processing can mitigate certain problems.

Here I will first introduce asynchronous evolutionary algorithms at a high level, explaining how they arise from the simple idea of combining the master-worker model of parallel and distributed optimization with a steady-state model of evolution. Then in my review of the EC community's recent efforts to apply and understand the performance characteristics

of these algorithms, I find that three groups of practically important questions have not been answered: namely

1. how much *speedup* asynchronous algorithms achieve over their traditional, synchronous (generational) counterparts,

2. whether and to what degree asynchronous algorithms introduce an *evaluation-time bias* that favors quick-evaluating (i.e., computationally inexpensive) solutions over slow-evaluating ones, and

3. how to mitigate the problem of *excess computation* that asynchronous algorithms may have in applications where higher-quality solutions take longer to evaluate than lower-quality ones.

I motivate each of these points below in sections 2.1.2, 2.1.3, and 2.1.4, respectively, and I restate them in section 2.1.5 as research questions that guide this dissertation.

2.1.1 Problem-Solving with Parallel Meta-heuristics

Evolutionary algorithms are part of a wider family of algorithms that are generally referred to as "meta-heuristics" [Blum et al., 2011, Stork et al., 2020]. Most meta-heuristics share in common the trait of combining a fairly general search strategy—which aims to balance exploratory and exploitative behavior while navigating a solution space—with domain-specific functions and customizations that improve an algorithm's performance on particular problems.[1] Evolutionary algorithms are the largest and most diverse subfamily of meta-heuristic algorithms.[2] This is because one of the properties that often distinguishes EAs from other meta-heuristic approaches is the emphasis that they place on maintaining a *population* of solutions during the search process as a means of balancing exploration and exploitation.

[1] Since its introduction by Blum and Roli [2003a], the term "meta-heuristics" and the field it has come to denote has been interpreted in several ways—some more helpful than others. I prefer the view that a *meta-heuristic* is in essence a general, "meta-level template" for defining a heuristic search algorithm—a template that is abstract enough to define a search strategy largely independently of the application domain.

[2] Examples of non-evolutionary methods in this family include simulated annealing and tabu search [Glover and Laguna, 1998].

This concept of *population-based search* can be implemented in diverse ways and is a common thread through most EA families—including genetic algorithms, evolution strategies, genetic programming, particle-swarm optimization, ant colony optimization,[3] etc.—and it makes these algorithms naturally amenable to execution on parallel architectures such as multi-core machines, distributed clusters, and massively parallel hardware such as GPUs.

Panmictic Master-Worker EAs

Because of their natural affinity with parallelism, parallel and distributed implementations of EAs have been deeply studied, and can be broadly divided into two major families [Alba, 2005]:

1. *panmictic* models,[4] in which variation and selection operators are applied to a single monolithic population, and

2. *structured-population* models (including both island models and spatial EAs), in which multiple sub-populations (sometimes called *demes*) are evolved separately with some degree of loosely coupled communication.

Each of these families of population models has different advantages and motivations. Structured-population models use a subdivided population and network topologies to reduce the cost of communication among processors, for instance. Minimizing communication costs is important in applications where the goal is to execute many evaluations of an *inexpensive* fitness function as quickly as possible. Island models address this by reducing network overhead,[5] whereas spatial EAs—such as the cellular EA, where small populations in each "cell" interact only with their neighbors—restrict communication in ways that may be more easily implemented on a GPU [Soca et al., 2010]. Parallelization of all three families (panmictic, island, and spatial) has been studied extensively [Alba and Tomassini, 2002,

[3]Because EAs are often equated with population-based algorithms, the term is inclusive of related algorithm families that do not reference evolution in their names—such as particle-swarm optimization and ant-colony optimization.

[4]From πᾶς (all, every) + μίξις (mixing), since all individuals are able to mate with all other individuals.

[5]This can also be useful in cases where genomes are very large and thus non-trivial to transmit across processing nodes, as when dealing with large weight vectors for neural networks.

Cantú-Paz, 1998, Nowostawski and Poli, 1999]. The recent survey by Harada and Alba [2020] gives a detailed discussion and taxonomy.

Panmictic master-worker methods[6]—also known simply as "global parallel" EAs [Harada and Alba, 2020]—have generally been less studied by researchers (mostly on account of their relative simplicity). These algorithms, however, offer the simplest and most widely used parallel architecture for evolutionary computation. A master-worker EA maintains a single, global population on the master processor, and dispatches individuals in the population out to worker processors to have their fitness evaluated. Sometimes reproductive operators are also executed on worker nodes, if they are expensive enough that executing them on the master would introduce a significant bottleneck. Master-worker EAs are beneficial when the cost of fitness evaluation (and/or reproductive operators) is substantially greater than the cost of transmitting individuals from the master to the workers. These algorithms thus yield significant speedup when applied to problems where the computational cost of fitness evaluation dwarfs other costs.

In my experience, a very wide class of applications that EAs are a good fit for meet this description. As access to and the need for high-performance computing power has grown in recent years, more and more applications of meta-heuristics are attended by *expensive* fitness functions. For example, I have often worked with colleagues on applying EAs to search decision spaces or parameter-tuning spaces that are associated with complex simulations [D'Auria et al., 2020]. In an economic modeling context, an EA might evolve a sequence of business transactions that an agent executes during a simulation of a tax system [Scott et al.,

[6]Historically these have, along with similar algorithms throughout computer science, almost universally been referred to as "master-slave" algorithms (including in my own past publications). This vocabulary has always been somewhat in tension with the U.S. Constitution, however, particularly since the passage of the XIIIth amendment in 1865; my own friends have often been shocked at how cavalier computer scientists are about antebellum metaphors (and also "male" vs. "female" cable connectors—but I digress). As has often been observed (for instance, with the famous "secretary problem"), the names we give to canonical algorithms often reflect the prejudices of the era in which they were named [Christian and Griffiths, 2016, p. 11–12]. Most papers and software APIs in recent years have thus been quietly replacing this terminology as the community becomes disillusioned with casual analogies to enslaved people—and I adopt that trend here. Capitalism, however, is alive and well in America, so—while granting that today's computers are as yet no more capable of voluntary servitude in which they own their own labor than they are of the involuntary kind (current events notwithstanding)—I will be ruthless in giving jobs (still doled out by "masters" who orchestrate the means of production) to metaphorical "workers" throughout this dissertation!

forthcoming]; or in a neuroscience context, EAs can be used to tune the free parameters of individual neuron models [Venkadesh et al., 2018], synaptic connections [Zou et al., 2021], or learning rules like spike-timing dependent plasticity (STDP) [Chen et al., 2021] to fit complex neural network behavior to neurophysiological data, or to act as a controller for an autonomous vehicle [Scott et al., 2021]. EAs are also frequently applied in adversarial machine learning, to generate or select difficult example data to challenge a machine learner's ability to generalize [Scott et al., in press]. In all of these examples, I have encountered problems where the fitness evaluation procedure is extremely expensive (taking anywhere from seconds to hours to execute one or more simulations), and where all other sources of overhead in the evolutionary process are negligible by comparison. In some, moreover, fitness evaluation is stochastic, and more than one fitness sample is required to accurately estimate a solution's expected quality (a problem I have studied elsewhere, although it is outside the scope of this dissertation [Scott and De Jong, 2016a]).

Given the rapidly increasing popularity of these kinds of applications, the panmictic global-population model and the simple master-worker topology for parallel evaluation has become especially important in evolutionary algorithm applications in recent years.

Generational and Steady-State EAs

A second important classification of evolutionary algorithms is based on whether their population dynamics follow a *generational* model or a *steady-state* model (also known as *non-overlapping* generations and *overlapping* generations, respectively).

The most widespread general model of evolutionary population dynamics used in computer science is the non-overlapping generations model. Originating in mathematical models of population genetics in the early 20th century (and forming the setting of well-known theoretical results in biology, such as Hardy-Weinberg equilibrium), this model considers the case where "the cycle of birth, maturation, and death includes the death of all organisms present in each generation before the members of the next generation mature" [Hartl and

Clark, 2007]. Canonical examples of these generational models include early genetic algorithms and the subfamily of evolution strategies known as (μ, λ) ("mu comma lambda") algorithms—which use a population of μ parents to generate λ offspring (where most often $\mu = \lambda$)—as well as many others. Most particle-swarm optimization implementations and genetic programming algorithms, for example, can be said to follow a (μ, λ)-style model, in the sense that they utilize non-overlapping generations.

Algorithms with overlapping (steady-state) dynamics were studied in an influential way by Whitley [1989], whose `GENITOR` system used an overlapping-generations model in which offspring individuals are generated, evaluated, and compete for a place in the population one-at-a-time in such a way that they often coexist with their parents in the same population. This approach was motivated in part by a desire to "achieve faster feedback relative to the rate at which new points of the search space are being sampled." In general, methods of this type are referred to as *generation gap* algorithms, where the value of the generation gap refers to the number of offspring that are generated at each step: a generation gap of one indicates one-at-a-time generation of offspring, but larger gaps are possible, in which some arbitrary number of individuals are generated at each step [De Jong and Sarma, 1993, Grefenstette, 1981, Sarma and De Jong, 2000].

In the evolution strategies community, overlapping-generation models were historically known as $(\mu+\lambda)$ ("mu plus lambda") strategies—where $\mu = |P|$ parents are used to generate $\lambda = |O|$ offspring, and then the new parents are selected from the union $P \cup O$ of both populations.[7] Today the notational distinction between (μ, λ)-style and $(\mu + \lambda)$-style population models serves as a popular written (and spoken) shorthand throughout the evolutionary computation community to distinguish non-overlapping-generation or "generational" algorithms on the one hand from overlapping-generation or "steady-state" algorithms on the

[7] This is an abuse of notation which I will be guilty of repeating several times in this dissertation. Formally, populations are multisets (there can be multiple copies of identical individuals in the population at the same time), not sets. Properly, the kind of multiset combination indicated here should be termed the *sum* of two multisets rather than the union—but set notation is familiar and easy to follow, so I follow the convenient fiction of pretending that populations are sets when describing EAs at a high level of abstraction or in pseudocode. Alternatively, one can simply imagine that all individuals include a unique identifier such that no two individuals are ever identical.

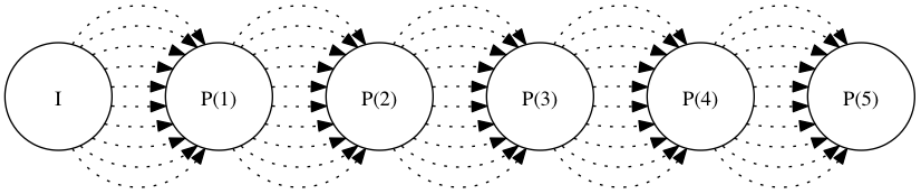

Figure 2.1: A generational master-worker EA synchronizes after each population has been evaluated in parallel.

other. The relative merits of each model are often understood in terms of the feedback loop governing the evolution of the population: generational algorithms have a longer feedback loop, since new information about fitness is not used until a full generation has elapsed, whereas steady-state algorithms have a tighter feedback loop, incorporating information about fitness immediately into the search process.

Asynchronous Steady-State EAs

A drawback to the steady-state, $(\mu+\lambda)$-style approach to evolution is that the model cannot be directly parallelized. Generational, (μ, λ)-style algorithms are "embarrassingly" parallel: all λ individuals can be evaluated simultaneously and independently of one another at each generation, as illustrated in Figure 2.1 and Algorithm 1. But in the steady-state model, only one individual (or a small batch of individuals, when the generation gap > 1) is created at a time, and no additional individuals are created until *after* that individual (or batch) has been evaluated and had a chance to be integrated into the population. This strict *ordering constraint* on the sequence of variation, evaluation, and selection events in the algorithm limits any attempt to scale the steady-state algorithm up to handle large computational loads via parallelism.

Asynchronous steady-state evolutionary algorithms (ASEAs) achieve parallelism by relaxing this constraint: these allow the order of events in the evolutionary process to vary depending on the time it takes for each individual to evaluate. ASEAs were first described by Grefenstette [1981], and implementations began to appear in the 1990s for both single-

Algorithm 1 A General Master-Worker Evolutionary Algorithm

```
1: function MASTERWORKEREVOLUTION(μ, generations)
2:     P ← INITIALIZE(μ)                          ▷ Initialize the population in parallel
3:     for i ← 0 to generations do                ▷ Begin generational evolution
4:         P′ ← BREEDOFFSPRING(μ, P)              ▷ Produce offspring (possibly in parallel)
5:         P′ ← EVALUATE(P)                       ▷ Evaluate fitness values in parallel
6:         P ← P′
```

and multi-objective problems [Kim, 1994, Stanley and Mudge, 1995, Talbi and Meunier, 2006]. While they have only been lightly studied to date, ASEAs have steadily gained importance as HPC paradigms have changed across science and engineering [Durillo et al., 2008]. Implementations of asynchronous optimization have been used for most of the major evolutionary algorithm families—such as genetic programming [Harada and Takadama, 2013, 2014, Jakobović et al., 2014, Maxwell, 1994] and differential evolution [Olenšek et al., 2011]—and also for closely related meta-heuristic paradigms such as ant colony optimization [Stützle, 1998] and particle swarm optimization [Koh et al., 2006, Mussi et al., 2011, Venter and Sobieszczanski-Sobieski, 2006]. The primary benefit that has been cited for using ASEAs is that they can avoid *idle time* that traditional generational EAs suffer in some applications (I will discuss this motivation in more detail below, in section 2.1.2). Another significant benefit of ASEAs is that they can be easily built in such a way that they are resilient to the failure of worker nodes, or such that the number of worker nodes can vary dynamically while the algorithm runs (as nodes come online and go offline arbitrarily).[8]

I will describe the basic ASEA and some of its variations in more detail in Chapter 3, but briefly, in an ASEA new individuals are typically generated one-at-a-time by selection and reproduction operators as CPUs become available, and are integrated into the population immediately when they finish having their fitness evaluated. In this sense, ASEAs operate much like $(\mu + 1)$-style EAs, except that they use more than one processor. Figure 2.2

[8]The robustness of ASEAs to node failures was recognized early by Grefenstette [1981]. While in this dissertation my experiments do not focus on this aspect of these algorithms, robustness does play a significant role in many of the HPC-oriented EA applications that I have been involved with.

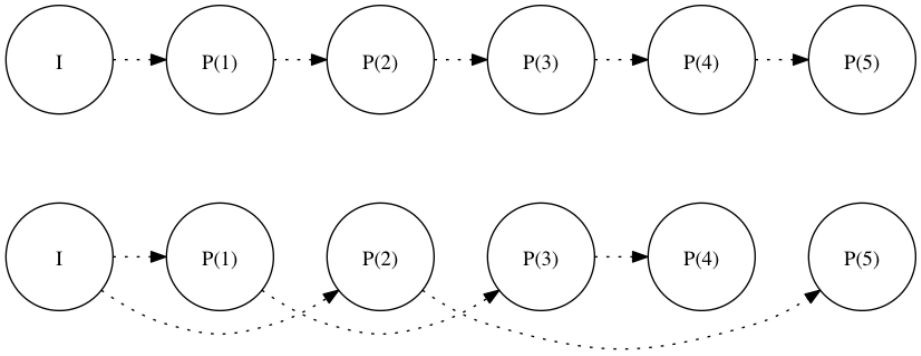

Figure 2.2: In steady-state evolution, individuals compete for a space in the population at the immediately subsequent step (**Top**). In the asynchronous steady-state EA, several evolutionary steps may pass before an individual enters the population (**Bottom**).

illustrates the difference between the traditional (single-processor) steady-state EA and the ASEA: the ASEA allows more than one individual to be evaluated simultaneously—but while a single individual evaluates, several other individuals may complete evaluating in the meantime and enter the population, causing considerable evolutionary change.

Selection Lag

As Rasheed and Davison observe in their early application of an asynchronous EA, "the creation of a new individual may not be affected by individuals created one or two steps ago because they have not yet been placed into the population" [Rasheed and Davison, 1999]. The number of evolutionary steps that pass while a single individual is being evaluated on a free processor has been called the *selection lag* by Depolli et al. [2013]. This order-shuffling effect introduces considerable complexity into ASEA behavior whenever some solutions take longer to evaluate than others, and this remains an obstacle to fully understanding the strengths and weaknesses of ASEAs as an optimization strategy [Rasheed and Davison, 1999]. A sequential steady-state EA always has a selection lag of 0, but Depolli et al. prove that the average selection lag in a $(\mu + 1)$-style asynchronous EA is $T - 1$ steps, where T is the number of worker processors. This holds even when fitness evaluation times vary

considerably from individual to individual or processor to processor.

Non-Asynchronous Parallelism and Quasi-Generational Algorithms

The ASEA is not the only possible parallel interpretation of steady-state evolution—Wakunda and Zell [2000], for example, describe a $(\mu+\lambda)$-style steady-state evolution strategy with "local tournament selection" that generates several offspring and evaluates them in parallel in a synchronous fashion, but chooses only one of them to integrate into the population. This parallel steady-state model maintains a strict ordering constraint on the evaluation of individuals and involves no asynchronous communication among processors—albeit with the trade off that only a limited form of parallelism is possible (discarding all but 1 individual in each parallel batch).

It is also possible, conversely, to conceive of asynchronous evolutionary algorithms that do not adhere strictly to a steady-state model—I will discuss "quasi-generational" and "semi-asynchronous" algorithms in section 2.1.3, for example [Durillo et al., 2008, Fonseca and Fleming, 1998, Harada, 2020b]. Nevertheless, the ASEA has been by far the most common parallel scheme for steady-state evolution (and the most common asynchronous scheme for panmictic evolution in general).

Asynchrony vs. Structured Populations

Figure 2.3 summarizes the various kinds of parallel evolutionary algorithms (PEAs) that I have discussed at this point, using a two-dimensional matrix. The generational master-worker EAs in the lower-left quadrant are the most common PEA in day-to-day practice among engineers. Panmictic ASEAs (in the top-left quadrant) have been less studied, but have become increasingly attractive in practice in recent years. Structured-population parallelization schemes have been heavily studied by the research community, but because they involve somewhat more complexity to implement and deploy in a cluster computing environment, practitioners tend only to use these algorithms when they have a specific reason to.

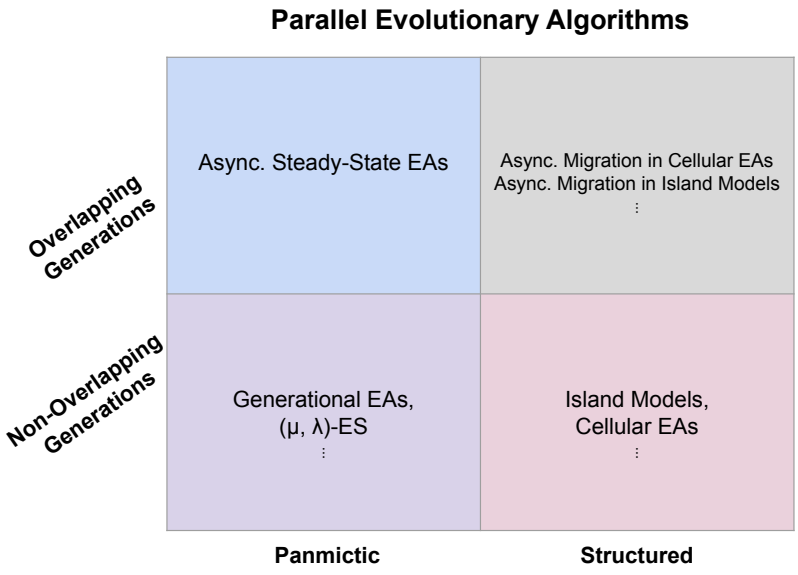

Figure 2.3: A simple two-way classification of parallel evolutionary algorithms by population structure and whether generations overlap during evolution.

In general, structured-population algorithms are typically built atop a generational model of evolution: both evolution within each cell or deme (that is, sub-population) and migrations between them occur at regular generation boundaries. Communication procedures based on (asynchronously) overlapping evolution have sometimes been used for inter-deme signals in structured-population models, however: island models have been created that permit asynchronous migrations between subpopulations [Alba and Troya, 2001, Dufek et al., 2021, Grefenstette, 1981, Liu et al., 2002], and asynchronous cell updating has been studied in cellular EAs [Giacobini et al., 2003]. But in general, asynchronous mechanisms have not been advertised as promising major performance advantages for structured-population models, and I am not presently aware of any work that has implemented or studied structured-population models that include asynchronous steady-state evolution internally within each sub-population for *intra*-deme evolution.

That said, there is nothing preventing (asynchronous) steady-state evolution from being

applied to the demes in a structured population model (by, for example, using steady-state evolution to update the internal population of each of the demes in an island model). Jakobović et al. [2014] have suggested that this may be useful in the same circumstances that island models with non-overlapping generations are beneficial: namely in cases where algorithmic overhead is high enough that the master process becomes the algorithm's primary performance bottleneck. On arguable exception to this gap in the literature is the work of Merelo et al. [2013], who have introduced a "pool-based" EA that allows subsets of a centralized population to be farmed out asynchronously to worker processors for evolution, as a sort of intermediate between panmictic and structured population models.

2.1.2 Idle Processor Time and Asynchronous Speedup

Evaluation-Time Variance

In the applications I have been involved with, my colleagues and I often observe a non-negligible amount of parameter-dependent variance in the evaluation times of the simulation models being tuned. At any given generation, the slowest individual in the population may take several times longer to evaluate than the fastest-evaluating individuals.

Variance in evaluation times has become especially important as applications that involve using search and optimization to tune the parameters of computationally expensive simulations become more popular in science and engineering (ex. [Carlson et al., 2014, Churchill et al., 2013, Tayarani et al., 2015, Van Geit et al., 2008]). Variance in evaluation times arises for different reasons in a wide class of applications: when tuning the parameters of a computationally intensive simulation, for instance, some configurations may cause the simulator to engage in more expensive and time-consuming operations than others. In genetic programming, a large, bloated program structure will take more time to evaluate than a small, parsimonious one [de Vega et al., 2020, Martin et al., 2015, Sambo et al., 2020]. As a third example, in a distributed evolutionary algorithm, evaluation times may vary if the load or computational capacities of the available compute nodes are heterogeneous.

Regardless of its source, variance in evaluation times can have a pronounced impact on the performance and efficiency of parallel evolutionary algorithms. In particular, classical master-worker EAs that use a generational model of evolution (as in Algorithm 1) leave some processors idle as they wait for the longest-evaluating individual in each generation to return a fitness value [Cantu-Paz, 2000]. All the processors in a generational algorithm must wait for all the other processing jobs to complete before moving to the next generation. This can lead to a significant amount of CPU idle time: Figure 2.4 illustrates this with 10 simulated evaluation times sampled uniformly from $[0.5, 1.0]$. The lightly shaded region shows the idle CPU time induced as 9 of the 10 nodes wait for the next generation. In some applications, as much as 50% of the available computational resources have been observed to be left idle as a direct result of variance in evaluation time Churchill et al. [2013]. The greater the variance in evaluation times, the lower the utilization of the processors.

When large numbers of processors are used, this problem may become exacerbated: the more jobs that are run in parallel, the greater the running time of the longest-running job is likely to be (by virtue of taking a larger number of samples from the distribution that defines evaluation times), and thus the greater the ratio of CPU resources left idle. As it has become normal to use large computing clusters to approach complex optimization problems in recent years (my colleagues and I, for example, have recently used our LEAP software framework to run evolutionary algorithms that use upwards of 2,400 processors at a time [Coletti et al., 2020]), this problem poses a severe scaling and efficiency challenge to evolutionary algorithm applications.

Asynchronous Algorithms as a Solution to Idle Time

The general problem of idle resources during synchronous data collection affects a wide class of distributed algorithms of which evolutionary methods are just one example. In a machine learning context, for instance, mainstream approaches to federated learning across edge devices typically impose a synchronous synthesis step [McMahan et al., 2017]. But this leads to the so-called "straggler problem," in which the system must wait for the slowest learner

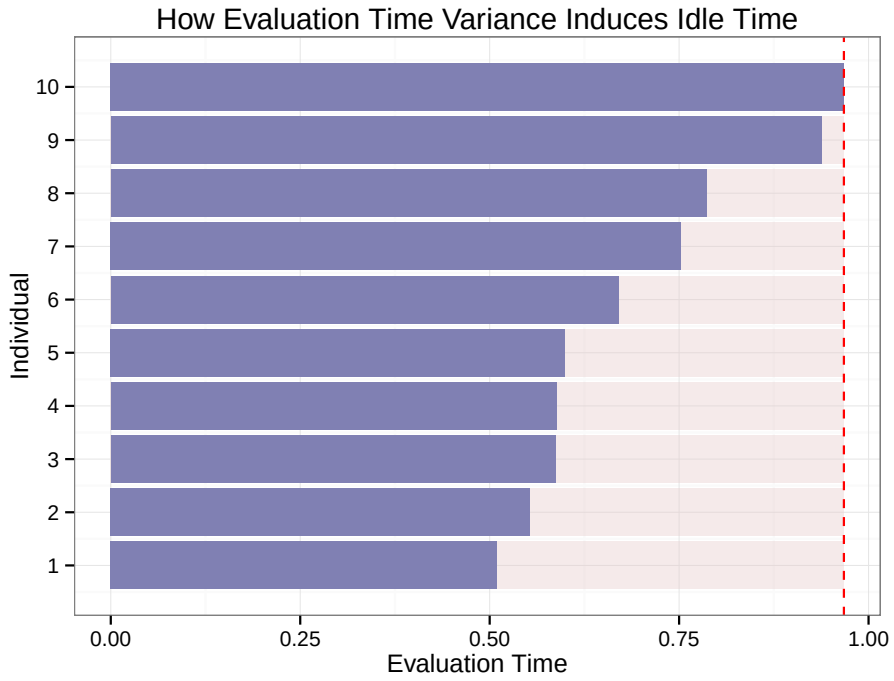

Figure 2.4: When there is variance in individual evaluation times, a parallelized generational EA suffers idle time. The fraction of time that each processor spends idle in each generation is determined in large part by the evaluation time of the longest-evaluating job across all the processors.

in its network to complete training [Xia et al., 2021]. This has recently motivated a family of asynchronous federated learning algorithms that avoid idle time, but which potentially raise other issues—such as biasing the global learner toward information provided by faster-running local learners [Chai et al., 2020, Chen et al., 2020b]. Though the details differ, this situation is closely analogous to similar behavior that occurs with asynchronous EAs.

In evolutionary search and optimization, asynchronous interpretations of fitness evaluation and steady-state evolution can eliminate this idle time that results from evaluation-time variance. When fitness evaluation is the most expensive operation in the algorithm, asynchronous parallel EAs have the virtue of being able to keep an unlimited number of processors operating at a nearly 100% utilization rate. While earlier work on asynchronous EAs by authors such as Zeigler and Kim [1993] was motivated by other interesting properties—such as the observation that ASEAs are capable of utilizing an unlimited number of processors (potentially much greater than the population size) [Kim, 1994]—more recent work by researchers and practitioners alike has sought to reclaim idle resources. There seems to be no effective way to accomplish this except by moving away from a synchronous, generational model and introducing a generation gap (ex. [Alba and Troya, 2001, Depolli et al., 2013]),[9] and ASEAs have now been widely applied to problems with evaluation-time variance.

For instance, Churchill et al. apply an asynchronous multi-objective evolutionary algorithm (MOEA) to a tool-sequence optimization problem for automatic milling machine simulations. They find that both the synchronous and asynchronous methods achieve solutions that are comparable in quality after a fixed number of evaluations, but that the asynchronous method completes those evaluations in 30–50% less wall-clock time [Churchill et al., 2013]. Yagoubi et al. similarly apply an asynchronous MOEA to the design of an engine part in a simulation of diesel combustion, with favorable results after a fixed number of evaluations [Yagoubi, 2012, Yagoubi et al., 2010]. Overall, the last decade has seen a small surge of new publications that apply or analyze asynchronous EAs of this kind [Coletti

[9]Though, as previously mentioned, examples like the "quasi-generational" algorithm of Durillo et al. [2008] suggest that hybrid asynchronous-generational approaches may be possible.

et al., 2021, Gong et al., 2015, Harada et al., 2020, Jakobović et al., 2014, Martin et al., 2015, Reisch et al., 2015, Said and Nakamura, 2014, Salto and Alba, 2015, Tagawa and Takeuchi, 2015, Zăvoianu et al., 2015].

Understanding Asynchronous Speedup

While ASEAs show clear benefits in many applications, their search behavior differs from both generational and single-processor steady-state algorithms in ways that are poorly understood. Overall, the performance of the asynchronous EA depends in a readily evident but poorly understood way on the number of worker processors and the distribution of individual evaluation times. In particular, while one can often guarantee that an ASEA will perform better on a particular problem than a generational algorithm in terms of fitness evaluations per unit time (i.e., *throughput*), those additional fitness evaluations may not always lead the algorithm to find better solutions, or to find a good solution in less time. Runarsson has shown empirically, for example, that as the number of worker processors is increased, more function evaluations are needed for his asynchronous evolution strategy to make progress on unimodal functions [Runarsson, 2003].

Few studies have attempted to tease out a theoretical or empirical understanding of what kinds of problems ASEAs may be poorly- or well-suited for. While my focus in this dissertation is on single-objective problems, some analysis of the asynchronous master-worker model has taken place in the context of multi-objective optimization, using MOEA approaches that are inspired by the steady-state EA. In empirical studies on small test suites, Durillo et al. [2008] and Zăvoianu et al. [2013a] each find that the asynchronous approach performs well at finding good Pareto fronts in less time than other approaches. Zăvoianu et al. also use an argument based on Amdahl's law [Amdahl, 1967] to put a lower bound on the speedup in evaluations-per-unit-time that the asynchronous approach provides as compared to a generational approach. They suggest that the asynchronous EA can provide some improvement in computational capacity even when evaluation times are very short and have negligible variance. When evaluation times are much longer than the EA's sequential operations (i.e.,

reproduction, selection), however, negligible speedup is predicted by their model unless there is variation in evaluation times.

What these past studies have not established is under what conditions, or by how much, an ASEA will be able to more quickly find high-quality solutions to problems. For example, as a steady-state model, it seems the faster feedback loop that the ASEA uses for optimization may lead it to have greedier behavior. This could make it more prone to falling into local optima on some problems than a similarly configured generational algorithm. In general, practitioners would like to know whether they can reliably expect beneficial results from an asynchronous EA. Little to no such guidance currently exists in the literature.

A Four-Part Classification of Evaluation-Time Properties

When approaching questions about what kind of speedup ASEAs will offer in different scenarios, it is helpful to distinguish between a handful of broad problem classes. In particular, the impact of asynchronous evaluation on an algorithm's search trajectory will depend in part on the evaluation-time characteristics of particular problems. In the simplest (i.e., linear) analysis, there are four main ways that fitness and evaluation time can be related:

1. Evaluation time may be stochastic but completely **non-heritable** (i.e., independent of both fitness and any other genetic trait in the individual). This kind of evaluation-time variance is common, for example, as a result of operating system scheduling effects, or of running a distributed algorithm on a cluster with heterogeneous resources (ex. nodes with different architectures or CPU speeds).

2. Evaluation time may be **heritable but independent of fitness**, in the sense that it is a trait that is determined (or partly determined) by the values of a solution's genes, but in such a way as to be completely independent of fitness.

3. Evaluation time may have a **correlation** with fitness such that **better solutions are slower-evaluating**. This may happen, for example, when evolving parameters for

a simulation in which successful solutions lead to longer-lasting simulations, such as successfully driving a vehicle without crashing it.

4. Likewise, evaluation time may have a **correlation** with fitness such that **better solutions are faster-evaluating**. This can happen, for example, when evolving a solution to problem in which initial, low-quality solutions execute an exorbitant number of computational operations (This often happens when tuning the parameters of spiking neural networks, in which low-quality solutions often have excessively high neuronal firing rates).

These four classes are not exhaustive (non-linear/non-convex correlations don't neatly fit into any of the four) or completely mutually exclusive (real-world problems may exhibit aspects of all four categories at once), but they serve as a convenient first-order taxonomy for classifying ASEA applications.

Since I introduced this four-part classification in Scott and De Jong [2015], the community has often used it as a starting point for defining experimental evaluations of asynchronous evolutionary algorithms. Harada et al. [2020], for example, focus their evaluation of an asynchronous variation of the MOEA/D algorithm on the non-heritable, better-is-faster correlation, and better-is-slower correlation cases.

2.1.3 Evaluation-Time Bias in Asynchronous EAs

While asynchronous steady-state EAs can eliminate idle time, they do so at the cost of introducing a dependence between the evolutionary trajectory that the population takes and the evaluation-time characteristics of the problem. Specifically, it seems that these methods exhibit an implicit selection bias that favors individuals that are cheap to evaluate. This effect was predicted by Grefenstette [1981], but to my knowledge was first empirically observed and reported by Yagoubi et al. [2011]. It has been frequently cited since as a potential source of unexpected or unwanted behavior in ASEAs.

In some cases, an evaluation-time bias may be desirable. Martin et al. [2015] observe, for

instance, that a bias toward fast-evaluating individuals in genetic programming may provide a favorable parsimony pressure in ASEAs. Sambo et al. [2020] have recently exploited this effect intentionally to favorably bias a genetic programming algorithm to avoid evolving complex and bloated solutions to symbolic regression problems. In their experiments, using evaluation-time bias to control complexity worked better at preserving the accuracy of the solutions than traditional bloat-control mechanisms. Outside of genetic programming, Yagoubi et al. [2011] construct an artificial example of a multi-objective optimization problem in which penalizing high-quality solutions by giving them long simulated evaluation times improves the performance of an ASEA by helping to prevent premature convergence.

In general, however, practitioners are wary of evaluation-time bias, as it may either hinder the algorithm's ability to find high-quality solutions quickly, or it may otherwise introduce unwanted implications. In scientific applications, especially, the goal is often to find regions of a model's parameter space that maximize goodness of fit to some data set—an additional pressure toward models that are computationally efficient may undermine the validity of a study's conclusions in some cases.

Understanding Evaluation-Time Bias

Intuitively, the reason that practitioners anticipate an evaluation-time bias in asynchronous EAs is that, while an individual with a long evaluation time is being executed on one processor, the other processors might evaluate numerous faster individuals. An example of this is shown in Figure 2.5, which shows the sequence of evaluation times for 100 steps of an asynchronous EA simulation as it minimizes a paraboloid function. Evaluation time is proportional to fitness in this simulation, and the EA utilizes 10 simulated processors.[10] The dashed lines emphasize the times that an individual with a particularly long evaluation begins and ends. Once the single long-evaluating individual completes, more than 50 individuals have completed evaluation and had a chance to compete for a place in the

[10]Because the algorithm is operating with simulated evaluation times, only the relative differences in evaluation times matter, and I show time in arbitrary units.

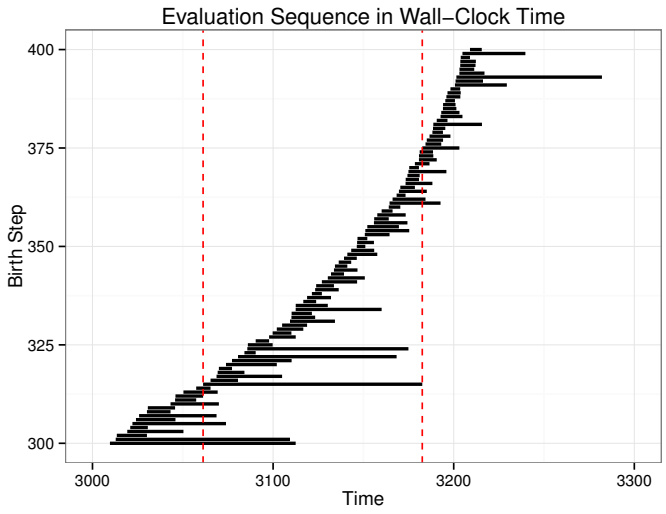

Figure 2.5: A Gantt-style visualization of a sequence of fitness evaluation durations in an ASEA, with the start and end time of one relatively long-evaluating individual marked with red dashed lines. There were ten processors in this simulation, so exactly ten jobs are executing in parallel at any given point in time.

population. This would appear to put long-evaluating individuals at a disadvantage, since in some cases fast individuals have more opportunity to reproduce.

To date, however, no studies have systematically quantified the severity of evaluation-time bias or studied specifically when or how it occurs during the execution of an ASEA.

Proposed Solutions to Evaluation-Time Bias

As a consequence of these concerns—and in the absence of clear evidence for or against the risks that evaluation-time bias presents—some authors have suggested novel algorithms that combine some aspects of existing synchronous and asynchronous EAs, in an attempt to achieve the best of both worlds. In particular, Fonseca and Fleming [1998] have proposed what they call a *quasi-generational* EA (QGEA) in which idle time is reclaimed by generating extra offspring to fill the vacant CPU resources. As individuals complete evaluating, they are added to an offspring population. When the offspring population reaches a size equal to the parent population, it replaces the parent population, and a new, empty offspring

population is created. In this way, observe Durillo et al., "the master does not have to wait until all the individuals of a generation have been evaluated" [Durillo et al., 2008].

While the quasi-generational EA has been suggested and discussed, to my knowledge Durillo et al. [2008] are the only authors who have implemented a QGEA (applying it to a multi-objective optimization problem). A few other closely related "semi-asynchronous" models have been introduced by various authors, however [Chitty, 2021, Harada and Takadama, 2020, Mazière et al., 2020, Sanhueza et al., 2017]. Harada [2020b], for example, has empirically studied a variation of the ASEA that does not integrate newly evaluated individuals into the population immediately, but waits a number of steps defined by an "asynchrony parameter" α before integrating them. By varying α, they can obtain fully asynchronous behavior, fully synchronous (generational) behavior, or something in between. Harada has shown that this semi-asynchronous model performs well (with appropriate choices of the α parameter) on a suite of multi-objective optimization problems—but they have not specifically studied evaluation-time bias with this algorithm.

What all of these approaches share in common is the hypothesis that a hybrid algorithm design can be devised—something between a traditional generational model and a steady-state asynchronous model—that captures desirable characteristics of both algorithms. The advantages and disadvantages that the QGEA and related hybrid models have, however, in comparison to the generational EA and ASEA remain an open question. This is particularly the case with respect to evaluation-time bias—where it remains unclear whether a hybrid model exists that can avoid evaluation-time bias while maintaining the performance advantages of asynchronous evolution.

2.1.4 Specialized Selection Strategies for Asynchronous EAs

Some authors have proposed specialized selection operations that aim to alleviate particular challenges that arise in parallel steady-state EAs. Some of these are specific to particular algorithms or applications. For instance, Harada et al. [2020] introduce a specialized parent-selection operator to adapt the popular MOEA/D algorithm for multi-objective optimization

to an asynchronous steady-state population model. Others are more generally applicable: Wakunda and Zell [2000] introduce a selection method based on the population's median fitness value as way of allowing steady-state selection to behave more like (μ, λ) selection—and thereby making it easier to apply traditional self-adaptation mechanisms to parallel $(\mu + \lambda)$ evolution strategies.

More recently, Harada [2020a] has introduced a specialized parent-selection operator for asynchronous steady-state EAs that specifically targets the problem of evaluation-time bias. Observing that evaluation-time bias occurs when fast-evaluating individuals effectively make "progress" more quickly than slow-evaluating ones (in terms of creating increasingly long lineages of parents and offspring), Harada's operator introduces a *search frequency* value that is maintained for each individual. This value is used as way of taking the length of each individual's evolutionary history into account when choosing parents. This allows sets of individuals that have exploited evaluation-time bias and thus made extra "progress" to effectively be throttled, so that lineages that are disfavored by evaluation-time bias can "catch up."

A distinctive property of steady-state evolutionary algorithms (both in their traditional, single-processor form and in parallel renditions) is that they include evolutionary selection at *two* points in the algorithmic loop [De Jong, 2006, p. 54]:

1. selection occurs when parents are chosen to generate offspring from—this is termed ***parent selection***,

2. but selection also occurs when individuals are selected to be replaced by a newly evaluated offspring—this is often referred to as ***survival selection***, since this procedure determines whether an existing individual in the population "survives" or is replaced by the new individual.

Because there are two different sources of selection at work in this family of algorithms, steady-state methods are prone to having high selection pressure. Because high selection

pressure tends to yield a greedy algorithm that performs poorly in many applications, practitioners who use steady-state algorithms typically configure one of these two operators to use a selection strategy that has little to no selection pressure, thereby reducing the overall pressure. Different choices of low-pressure operators have been studied—such as performing selection based on a FIFO queue [De Jong and Sarma, 1993, Parker and Parker, 2006]—but *random uniform selection* is most commonly used in practice.

What these operators typically have in common is that they make no use of fitness information: individuals with good fitness are not favored over individuals with poor fitness. One insight I present in this dissertation is the recognition that this property presents an opportunity for a specialized approach to selection: when used as a parent-selection method, these operators are able to select individuals that have *not yet been fully evaluated* and to use them as parents to generate offspring. I will return to this idea below when formulating this suggestion as a research question.

2.1.5 Research Questions

Building on the review of the ASEA literature I have conducted in this section, several research questions have governed the experiments I pursue in this dissertation (and particularly in Chapter 3).

The first concerns the speedup of the ASEA, and whether it tends to reliably produce a performance improvement over the (μ, λ)-style alternative. I will use a combination of analytical proofs and experiments to examine the following research question:

Research Question 1. ***Asynchronous Speedup****: How much speedup does the ASEA show over a traditional, generational EA in different scenarios? And are there simple cases where an ASEA should clearly be avoided?*

I will approach this question by making use of the four-part classification of optimization problems based on evaluation-time properties that I presented in section 2.1.2, performing empirical tests of ASEA behavior on examples of each, and presenting some analytical results

that place bounds on the speedup in the special case of non-heritable evaluation times.

Next, I turn to the question of evaluation-time bias, to understand its causes and severity, and whether it can be mitigated in a straightforward way. I pose this as a two-part research question:

Research Question 2. ***Evaluation-time Bias****:*

A) Under what conditions does evaluation-time bias occur in an ASEA, and how severely might it effect an algorithm's problem-solving ability?

B) Can the evaluation-time bias of asynchronous evolution be mitigated by adopting a quasi-generational approach as described by Fonseca and Fleming [1998]?

Finally, I will focus in on the opportunities provided by specialized selection operators for ASEAs. Specifically, I will introduce a parent-selection operator that selects among not just the current (fully evaluated) population P, but also from the set E of new individuals whose fitness is currently being evaluated. I refer to this method that selects parents from $P \cup E$ as "Selection WhilE Evaluating" (SWEET). This approach to selection may have the potential to improve ASEA performance in cases where higher-quality solutions take longer to evaluate.

Research Question 3. ***Selection WhilE EvaluaTing (SWEET)****: Does including individuals in selection that are still being evaluated reduce "excess work" in ASEAs when evaluation time has a better-is-longer correlation with fitness (longer individuals are better)? i.e., does it lead to an improvement in performance? Does it impact performance when the evaluation time has the opposite (better-is-faster) correlation with fitness?*

On a methodological level, the evolutionary behavior of the asynchronous EA is complex and difficult to describe in a purely analytical way. For instance, replicator dynamics are one tool (from evolutionary game theory) that can be used to study the dynamics of evolutionary algorithms [Ficici et al., 2000]. But the replicator dynamics of an asynchronous evolutionary

system are non-Markovian, making them more difficult to analyze. As such, while I use some analytical results where possible in this dissertation, I will rely heavily on empirical studies to target these research questions and improve the community's understanding of asynchronous EAs.

2.2 Evolutionary Knowledge Transfer

Having completed my background review of parallel and asynchronous evaluation schemes in evolutionary algorithms, I now turn to the second major subtopic of this dissertation: how knowledge transfer can be used to improve the performance of EAs on complex tasks. For the rest of this chapter, I will be primarily concerned with giving a broad motivational view of evolutionary knowledge transfer (EKT) and the different ways in which it has been approached. In particular, I begin in section 2.2.1 by presenting three high-level theories of why and how knowledge transfer might be useful in optimization. These different perspectives are speculative (given how young the field is), but provide complementary (and sometimes contrasting) reasons for investing research effort into this field at an abstract level. Then in section 2.2.2 I give background on what I mean by "heuristic knowledge," and summarize the different ways that knowledge has historically been integrated into evolutionary algorithms for single-task problem-solving. The remainder of this section is then devoted to a selective survey of the EKT literature to date (section 2.2.3), and in particular to three classes of open questions that form the core of the field's challenges (sections 2.2.4, 2.2.5, and 2.2.6).

Building on these open challenges, my review culminates in a set of research questions in section 2.2.7 that will guide my research in Chapters 4 and 5.

2.2.1 Speculative Motivations for Knowledge Transfer

> Principles are appropriated from or suggested by that which already exists, be it other devices or methods or theory or functionalities. They are never invented

> from nothing. At the creative heart of invention lies appropriation, some sort of mental borrowing that comes in the form of half-conscious suggestion.
>
> —Brian Arthur [2009], p. 115.

Many problems clearly become easier to solve if we are able to re-use knowledge that an algorithm has gained while solving a different, related task. Evolutionary knowledge transfer (EKT) algorithms take information that is acquired via an automated process on one or more *source tasks* and use it as heuristic knowledge while solving one or more *target tasks*. As a general problem-solving paradigm, knowledge transfer can be motivated in a few different ways. Three of the principle historical models for understanding knowledge transfer come out of psychology and biology: namely the identical-elements theory of Edward Thorndike, the general principles model of Judd, and Darwin's principle of exaptation.

In this section I discuss motivations for transfer that arise from practical settings and nature. In subsequence sections, I will look more specifically at how knowledge transfer has been realized in implementations of evolutionary algorithms and what problems these implementations raise.

The Identical Elements Principle

The most direct justification for using a transfer approach is when two or more tasks are readily at hand that clearly display strong and *obvious similarities* to one another. In the early 20th century, Thorndike argued that knowledge transfer across tasks in human problem-solving involves an *identical-elements principle* (IEP): this principle predicts that positive transfer can only occur in cases where two tasks share a significant overlap in concrete, low-level features.[11] A canonical example comes from elementary mathematics education: the

[11] Thorndike's emphasis on identical elements marked a significant departure from the popular *formal disciplines* theory that dominated Western educational philosophy until the 20th century. This theory has roots in the mental faculty theories of Aristotle, Thomas Aquinas, and Thomas Reid, and often invokes the analogy of the brain as a muscle that can be trained through exercise—in which case learning mathematics or memorizing Latin texts was believed to strengthen general faculties such as reasoning and memory, which could then transfer to an infinite variety of subjects. The formal disciplines theory is considered to be largely discredited today [Leberman and McDonald, 2016].

identical-elements theory predicts that once a child has learned to add numbers, they will have an easier time learning to multiply them, because the standard arithmetic algorithms for addition and multiplication share similar low-level symbol manipulations [Leberman and McDonald, 2016] (and in particular the pattern of dividing multi-digit computations into a series of single-digit operations [Wu, 2009]).

In computing, organizations often have need to solve a sequence of related optimization problems that share clear structural features. Problems in vehicle routing, scheduling, simulation tuning, pattern recognition, etc., often arise not one-at-a-time, but as a set or stream of tasks that bear significant structural similarities to one another (perhaps because they are based around similar geographical topologies, or simulations built from similar components, etc.) [Hart and Sim, 2014]. Zhang et al. [2021b], for instance, cite the example of manufacturers that face seasonal variation in the production of t-shirts or jackets: in the summer, numerous t-shirt orders lead to a challenging job-shop scheduling problem with high utilization, whereas a similar problem for the winter season involves lower utilization and is easier to solve, but still shares significant common properties. Knowledge transfer is a natural response to scenarios of this kind, and some of the earliest examples of knowledge transfer in evolutionary algorithms have been motivated by a desire to avoid requiring algorithms to "learn from scratch" on each new instance of a family of related optimization tasks (ex. [Ramsey and Grefenstette, 1993]).

The IEP view of transfer points to clear examples where transfer may be useful, but at the same time it suggests that transfer is of limited use outside these obvious cases of high similarity between tasks. One possible conclusion of this view is that knowledge transfer is, in general, best viewed as a niche approach: a good tool to have in one's toolbox if closely similar tasks arise, but one that will only be beneficial in rare cases. Alternatively, however, Gupta et al. [2016c] argue that with modern cloud-computing paradigms, it may be possible to collect a vast diversity of problems from practitioners around the world into a centralized location—in which case the probability of finding two or more tasks that are sufficiently similar to each other may be increased significantly. Several optimization researchers have

envisioned similar ideas for a centralized repository of optimization tasks for the purpose of knowledge sharing.[12] Such a system is purely hypothetical at this point, however, and many open questions remain about its feasibility—such as whether the similarity of tasks can be estimated automatically with enough accuracy for transfer opportunities to be discovered, and whether the *memory swamping* problem[13] can be overcome (in which the task of searching for relevant source tasks becomes so complex that it outweighs the benefits of knowledge transfer) [Fikes et al., 1972, Markovitch and Scott, 1988, Salamó and López-Sánchez, 2011].

Shared Causal Principles

A second major way to arrive at transfer methods is to focus on domains where common underlying causes appear to be broadly shared by a variety of problems or tasks in some large domain. Robotics tasks, for example [Yu et al., 2020], ultimately require navigating the same basic set of physical laws that hold in our universe—such that a principle that is relevant for solving one task may also be relevant in other tasks even if they are superficially dissimilar. In this view, knowledge transfer is likely to be most widely applicable and generalizable when it focuses on identifying and transferring these *shared causal principles*. An early proponent of a general principles view was Judd, whose 1908 underwater dart-throwing experiment claims to show that students who received an explanation of the principles of refraction performed better when learning to throw darts at an underwater target after the depth of the target was changed. While later experimenters had difficulty reproducing this result [Hendrickson and Schroeder, 1941], Judd's claim that understanding general principles plays an important role in facilitating transfer remains influential in psychology.

It is challenging to apply the causal-principle view of transfer in a computing context, because it is not easy to design algorithms that reason about abstract relationships that may be realized differently in different contexts. Aspects of this problem are addressed by the study of computational analogy-making, as in the influential structure-mapping theory

[12] For example, L. Graham, personal communication, October 12th, 2010.

[13] Known in the past as the *utility problem* in the context of cognitive architectures like Soar—the term "memory swamping" is less obscure and has become favored in recent years.

of Gentner [1983]—and some of the earliest work on knowledge transfer in heuristic search has sometimes taken the view that transfer is a kind of analogy-making [Carbonell, 1986]. Another relevant perspective is found in the work of Pearl [2009] and others on formally modeling causality and learning causal models from data. Pearl has suggested that causal models have a close relationship to knowledge transfer: causal models are distinguished from non-causal (correlational) models specifically in that the former are more likely to retain their predictive power when aspects of the environment are altered. Formal observations of this kind have recently had some impact on the machine learning community [Goudet et al., 2018, Lake et al., 2017, Schölkopf et al., 2021], but the role of causal knowledge in generalization and knowledge transfer has yet to be explored significantly in the meta-heuristics community.

Within machine learning, representation learning is one area where the causal-principle view has readily been applied to computational problems. In computer vision and natural language processing domains, transfer has been especially successful at allowing machine learning algorithms to re-use rich feature representations from models that have been pre-trained on large, general-purpose object detection and language data. Leading neural network software packages now routinely ship with "model zoos" that serve as a source for knowledge transfer in the form of re-used neural network layers and parameters [Shu et al., 2021, Such et al., 2018, Xu et al., 2021a].

Exaptation and Innovation Engines

Biological systems display reuse of genetic information in profound and far-reaching ways, offering a natural motivation for a third perspective on knowledge transfer. Take almost any phenotypic trait in a human or animal, and ask "what purpose did this trait evolve for?," and the answer tends to be a long narrative in which an intermediate structure evolved first for one purpose, then was co-opted for another, then another, and so on before it assumed its current form. Classical examples from animal morphology include lungs evolving from swim bladders (whose original purpose was to maintain neutral buoyancy) [Daniels et al., 2004, Perry et al., 2001], and feathers being adapted for flight (when their original purpose

in theropod dinosaurs may have been thermoregulation) [Prum and Brush, 2002]. Gould and Vrba [1982] famously introduced the term *exaptation*—indicating an aptness that comes "out of" something else—as a means of contrasting these features with *adaptations* (which is aptness "toward" something).[14] While determining the precise adaptive or exaptive history of specific morphological traits with confidence is difficult and requires careful empirical analysis [Baum and Donoghue, 2001, MacLeod, 2001, Martins, 2000, Smith, 2010], the picture is especially clear on the molecular level, where the duplication and modification of genes and protein domains—and their subsequent repurposing for new functions—is widespread and well-documented [Mistry et al., 2021]. Up to 90% of known protein domains (subcomponents) found in eukaryote genes, for instance, are reused in more than one gene [Orengo and Thornton, 2005].

Repurposing processes of this kind allow evolution to take an indirect route to solving problems, and sometimes facilitates complex adaptations that are unlikely to have been able to evolve more directly. An important minor thread of evolutionary knowledge transfer research has explicitly invoked exaptation as a process that should be studied in a computational context [Davies, 2014, Fentress, 2005, Graham, 2008, Mouret and Doncieux, 2009]. De Oliveria was one of the earliest to apply these concepts to evolutionary algorithms, introducing methods that take inspiration from the *shifting balance theory* of Sewall Wright [1932] (one of the earliest attempts to explain biological aptations in terms of crossing valleys in a fitness landscape) [de Oliveira, 1994]. While more recent authors tend to focus instead on the term "knowledge transfer" (owing to its familiarity from the machine learning literature), the bio-inspired concept of exaptation offers an important alternative lens through which to understand information reuse in optimization.

[14]Darwin's term for this concept was *pre-adaptation*. "The structure of each part of each species, for whatever purpose it may serve," he observed, "is the sum of many inherited changes, through which the species has passed during its successive adaptations to changed habits and conditions of life... A well-developed tail, having been formed in an aquatic animal, it might subsequently come to be worked in for all sorts of purposes, — as a fly-flapper, an organ of prehension, or as an aid in turning" [Darwin, 1859, ch. 5]. The terms exaptation, pre-adapation, "functional shift," and co-option are typically used interchangeably today.

In particular, a number of authors have more ambitiously argued that indirect problem-solving of this kind on a *massive scale* is essential to achieving innovative solutions to complex tasks in natural systems—and that transfer even from tasks that appear to be completely unrelated on the surface is necessary in order to solve hard problems. Arthur [2009], for instance, has argued that the development of human technology follows an exaptive pattern in which discoveries in diverse technological fields routinely facilitate unexpected breakthroughs on very different problems. Some of the first studies of evolutionary algorithms that use knowledge transfer among more than two or three tasks at a time were attempts to model natural examples of innovative systems: Lenski et al. [2003] devised a multi-task EA to model the evolution of complex phenotypic traits in biology, showing that complex Boolean-logic tasks were considerably easier to solve by co-opting solutions to simpler tasks. Arthur and Polak [2006] likewise introduced a model of technological evolution based on Boolean functions, showing that even a very simple genetic programming system based on random recombination of circuits could solve remarkably complex Boolean functions by reusing solutions to dozens or hundreds of simpler functions in a massive multi-task ecosystem.

This picture of widespread reuse and repurposing of evolved phenotypes differs significantly from the way that heuristic algorithms typically operate, including almost all evolutionary algorithms. Exaptation amounts to a heuristic that says "when solving a problem, first devise a solution to a completely different problem. Then co-opt that solution and refine it to solve the new task." Except in narrow cases (as discussed in the context of the identical-elements principle above), this indirect approach has not been a natural choice among engineers solving practical tasks.

Nguyen et al. [2015b] introduced the term "innovation engines" into the evolutionary computation community as a way to refer to massive multi-task systems that explicitly cultivate a diversity of knowledge transfer sources. An example of this behavior is notably evident in the PicBreeder evolutionary art system developed by Secretan et al. [2011]: they found that users of their online system were able to evolve a remarkable variety of complex images that resemble various objects, even where a standard evolutionary algorithm would

have failed to produce such an image with their particular solution representation and operators. This was in part because users were allowed to take solutions that other users had developed for one goal and evolve them toward new, distinct goals. Stanley and Lehman [2015] have argued extensively on the basis of the PicBreeder example that the paradigm of single-task optimization is fundamentally limited in its ability to solve complex problems: on complex tasks, traditional algorithms fall into severe local optima that prevent them from being able to make progress. They argue that an explicit pursuit of diversification and experience—which they term "stepping-stone" collection—is necessary in order to build algorithms that can solve complex problems.

The prevailing interpretation of "innovation engines" in the EC community, however, has been to equate stepping-stone collection with novelty-seeking mechanisms that encourage the evolution of a wide diversity of different structures within the context of a *single task* or objective function—typically by using human-defined metrics (known as *behavior characterizations*) to define diversity in a phenotype space [Lehman and Stanley, 2011, Pugh et al., 2016]. With some exceptions—such as the recent work of Wang et al. [2019] and Norstein et al. [2022] on robotic agents that explore and adapt to diverse environments—there has been less emphasis among these researchers on explicitly using different problems and associated fitness functions as a source of co-option and knowledge transfer.

Overall, the perspective of exaptation and innovation engines differs significantly from other motivations for knowledge transfer—such as identical elements or causal principles—in that it suggests that successful reuse and co-option has a tendency to be serendipitous, that it should be conceived as involving transfer among hundreds or thousands of different niches or problems rather than just two or three, that it is essential for evolving complex solutions to difficult tasks, and that useful examples of transfer may occur in unexpected ways and among problems that bear little obvious similarity.

The simplicity of exaptive mechanisms in evolution is also significant. Many discussions of knowledge transfer in AI associate it with sophisticated solutions to the problems of artificial general intelligence (AGI): such as using compositional reasoning and causal models

to generalize with very little training data [Lake et al., 2017], using massive language models as a generalized representation of knowledge and reasoning [Radford et al., 2019b], or more broadly with the executive functions of the prefrontal cortex in humans [Russin et al., 2020]. The relative simplicity of Darwinian evolution, however, suggests that transfer can achieve remarkable feats even without the guidance of sophisticated executive features.

2.2.2 Representations of Algorithmic Knowledge

> The fundamental problem of understanding intelligence is not the identification of a few powerful techniques, but rather the question of how to represent *large amounts of knowledge in a fashion that permits their effective use and interaction.* This shift is based on a decade of experience with programs that relied on uniform search or logistic techniques that proved to be hopelessly inefficient when faced with complex problems embedded in large knowledge spaces... Thus, we see AI as having shifted from a *power-based* strategy for achieving intelligence to a *knowledge-based* approach.
>
> —Goldstein and Papert [1977]

In this section I give a brief, but broad, summary of different ways that heuristic knowledge has historically been encoded in evolutionary algorithms that are built for single-task problem-solving. I use this to contextualize knowledge transfer in two ways: first, knowledge transfer is a potential solution to the high cost of traditional approaches to configuring algorithms with prior knowledge. Second, the strategies that have been used for evolutionary knowledge transfer can often be seen as variations on more traditional themes of algorithm configuration—such as seeding populations with high-quality solution instances, or tuning an algorithm's solution representation.

Structural and Heuristic Knowledge

The principle that algorithms require prior knowledge in order to solve problems efficiently is widely accepted in the AI community, and is often invoked in a variety of distinct contexts.[15] The kind of "knowledge" that algorithms can have tends to be vaguely defined in these conversations, and any definition of algorithmic knowledge would probably fall very short of the standards famously set in Plato's imagined dialogue with the mathematician *Theaetetus*.[16] But here I broadly distinguish between *structural* knowledge in algorithms on the one hand and *heuristic* knowledge on the other.

Structural knowledge occurs when algorithms use detailed information about a problem instance or its properties. Quadratic optimization algorithms, for example, explicitly encode knowledge about a problem's structure in a weight matrix Q as they minimize a function $f(\mathbf{x}) = \mathbf{x}^\top Q\mathbf{x}$. Algorithms that use structural knowledge are often seen as *white-box* approaches, and they are the focus of much of classical computational complexity theory. In some cases, having full structural knowledge yields very efficient algorithms that permit performance and accuracy guarantees for real-time and/or large-scale applications that would not otherwise be possible. In general, however, even complete structural knowledge of a problem is not always sufficient to yield efficient algorithms (as demonstrated by hundreds of problem classes that are NP-hard).

Heuristic knowledge, by contrast, occurs when algorithms use local strategies for ranking different decisions or "moves" that they may take in an iterative process. Heuristic approaches often sacrifice guarantees such as optimality, accuracy, or completeness in favor of being able to find adequate solutions to many complex problems efficiently. This includes heuristic scoring functions like those used in A^* search, general search procedures such as the Nelder-Mead simplex method [Nelder and Mead, 1965], and other strategies such as

[15] This concept was already so well-established forty years ago that Newell [1982] called it the "cliche of AI."

[16] That is to say, algorithmic "knowledge" very often is neither justified, true, nor a belief—to say nothing of modern problems such as Gettier cases, which further complicate the classical view of knowledge by showing that even a justified true belief is not sufficient for knowledge (even if it is necessary) [Zagzebski, 1994].

pattern databases which assemble collections of "moves" that may assist in more quickly finding solutions to classes of problems (such as the Rubik's cube) [Edelkamp, 2014, Korf, 1997]. It also includes specialized representations of solution spaces, which can simplify the structure of a problem or problem class [Amarel, 1968]. Today, heuristic algorithms are the main focus of *black-box complexity theory*, which treats complexity in terms of the number of queries that an algorithm requires in order to learn enough about a problem instance that it has little to no prior information about to be able to solve it [Doerr, 2020].

Many algorithms make use of both structural and heuristic knowledge simultaneously. Machine learning engineers leverage a limited form of structural knowledge about neural networks, for instance (specifically their synaptic weight matrices along with the derivative of their neuronal activation functions), when they apply algorithms based on gradient descent. Gradient descent itself, however, encodes a heuristic that builds on top of this structure.

Knowledge in Evolutionary Algorithms

As meta-heuristic methods [Blum et al., 2011, Stork et al., 2020], evolutionary algorithms are generally made up of a high-level algorithm template that is filled in by a series of components, parameters, and design decisions to obtain a heuristic algorithm for a particular application. The process of incorporating heuristic knowledge into an effective algorithm can be seen partly as the problem of making good design choices while implementing such a template.

Most evolutionary algorithms involve four main classes of design decisions at a minimum: 1) a fitness function to be optimized, 2) a representation of a solution space, 3) a means of initializing solution instances within this space, and 4) reproductive operators that select and modify said instances. Figure 2.6 summarizes several of the components of evolutionary algorithms that can be used to express heuristic knowledge.

Some algorithms also explicitly build machine learning models of the solution space while solving a problem. These include estimation of distribution algorithms (EDAs), surrogate modeling methods, and linkage-learning methods. EDAs learn and maintain generative

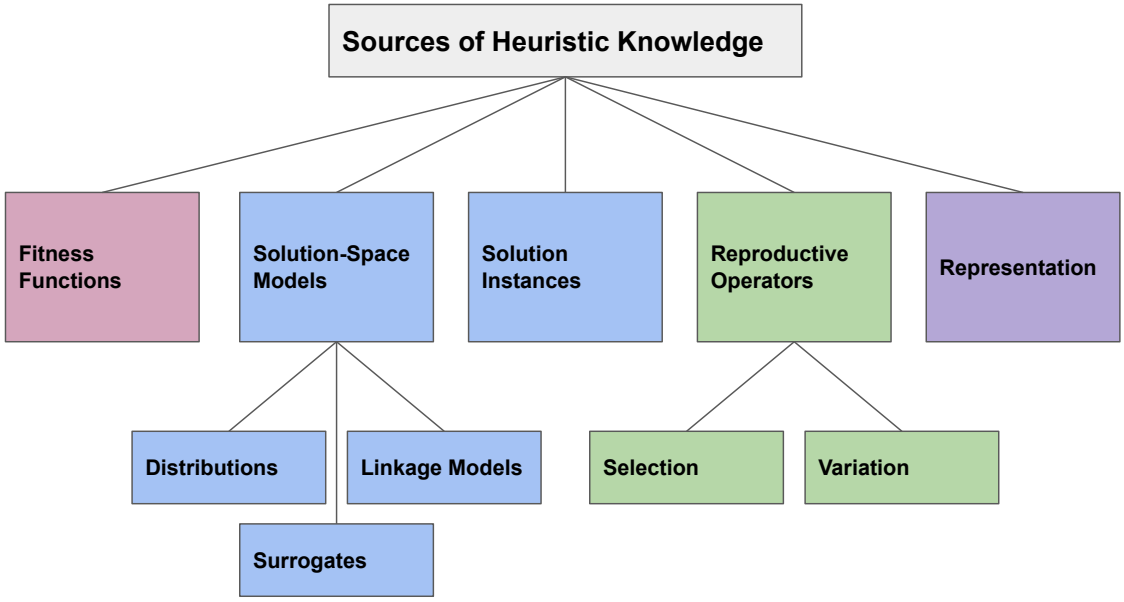

Figure 2.6: Aspects of evolutionary algorithm configuration where prior knowledge about a problem or problem class can appear.

models of a distribution for sampling high-quality solutions [Larrañaga and Lozano, 2001]. Some notable examples of EDAs include Bayesian optimization [Hauschild and Pelikan, 2011, Parsa et al., 2020], covariance-matrix adaptation (CMA-ES) [Hansen, 2016], and learnable evolution models (LEM) [Coletti, 2014]. Surrogate modeling approaches learn predictive models of the fitness function itself [Jin, 2005, 2011]. Linkage-learning methods, meanwhile, focus on learning models of pairwise or higher-order interactions in how variables relate to fitness [Baluja and Davies, 1997, Sastry and Goldberg, 2000, Yu et al., 2009].

"Gray-Box" Algorithm Design

Practitioners use the components in Figure 2.6 to configure EAs with prior knowledge in several ways. Much of this work involves in-depth human research and iterative testing of hypotheses to learn what kinds of heuristics tend to work well in a given domain. A striking example is the work of Whitley et al. [2009] on partition-based "tunneling" operators for the traveling salesman problem (TSP) [Tinós et al., 2015, Whitley et al., 2010]. Built upon

a sophisticated graph-theoretic analysis of the properties of Hamiltonian circuits as they appear in TSP instances, this work has devised special reproductive operators that are able to move directly to new local optima in a single step, bypassing poor-fitness regions in between. Whitley [2019] argues that results of this kind support the efficacy of a "gray-box" approach to optimization, in which randomized search algorithms are combined with deep human expertise to solve a variety of complex problems such as TSPs and Boolean satisfiability problems.

The counterargument to intensive research programs like Whitley's is that manually configuring evolutionary algorithms with deep expert knowledge requires advanced and specialized skills. This skill-and-knowledge bottleneck does not scale to the immense diversity of applications that EAs and other meta-heuristics are applied to, nor to the diversity of people—with different backgrounds, areas of expertise, and levels of familiarity with the EA literature—who use them to solve real-world problems.

Automated Algorithm Configuration

This concern over the skill bottleneck in the "gray-box" approach partly motivates the fields of automated algorithm configuration (AAC) [Hutter et al., 2009, López-Ibáñez et al., 2016] and hyper-heuristics [Burke et al., 2010, Tauritz and Woodward, 2018]—two closely related efforts that often use higher-level search and optimization algorithms to search over the space of algorithms itself. Work in these areas (much like analogous work on automated machine learning [He et al., 2021]) attempts to use automation to reduce the amount of prior input, effort, and human guidance that is required to design effective heuristics for complex problems or problem classes. These have been used to automatically synthesize components such as selection operators [Richter and Tauritz, 2018], local optimizers [Kamrath et al., 2020], and other heuristics [Illetskova et al., 2017, Pope et al., 2019] that perform well in many domains.

The clear challenge raised by AAC, however, is the computational cost involved in using algorithms to search for algorithm configurations. In fact, in an effort to cope with this

complexity, Bertels and Tauritz [2016] have argued that "asynchronous parallel evolution is the future of hyper-heuristics." Efficient parallelization schemes of the kind I surveyed in section 2.1, however, by themselves can only make a small dent in the broad problem of configuring algorithms with prior knowledge.

These concerns—the skill bottleneck that attends "gray-box" algorithm design, the computational cost of AAC, and the insufficiency of parallelization by itself to address the latter—are what motivate me to consider knowledge transfer as an alternative.

Population Seeding in a Single-Task Setting

Two specific strategies for placing *a priori* knowledge into evolutionary algorithms will prove relevant to my discussion of transfer below.

The first is *population seeding*. In research works, the initial population of solutions in an evolutionary optimization procedure is typically initialized randomly—often by sampling uniformly over some pre-defined multi-dimensional space. Random sampling encourages high initial diversity, allowing an algorithm to begin with an extremely exploratory sample and to move to more exploitative behavior as computation progresses. In many applications, however, clear alternatives to random initialization are available which can significantly improve an algorithm's performance. These strategies involve "seeding" a subset of the initial population with solutions that are derived from some process other than uniform random sampling [Grefenstette, 1987].

The knowledge sources used by population-seeding approaches vary. In some application domains, data or ad-hoc knowledge is available to algorithm designers that offers a clear path to improve population initialization. Kazemzadeh Azad [2018], for example, seeds a population that optimizes steel truss structures with solutions that are known to be feasible. In some cases, an explicit library of real-world data can be used to inform the generation or modification of solutions. Experimental results from x-ray crystallography, for example, yield large libraries of known stable folding structures for proteins. These can be used to initialize or otherwise inform meta-heuristic algorithms that aim to find and study additional

stable forms of the same molecules [Zaman et al., 2019]. Even in the absence of data or ad-hoc human knowledge that can inform initialization, practitioners often run one or more algorithms multiple times on the same problem instance—in which case the best-found solutions from earlier runs can be used to seed later runs. Whitley et al. [1991] used this principle to design a special approach to population seeding that alters the representation that is used for solutions in future runs.

In other applications, expert knowledge yields heuristics that can be used to stochastically generate solutions of higher quality than would occur with a uniform random approach [Paul et al., 2013, 2014]. Hopper and Turton [2001], for example, generate solutions to bin-packing problems under the constraint that objects are sorted in order of decreasing size—and then use the best solutions that result from this heuristic to seed an evolutionary algorithm that refines them. A related set of approaches initially use a highly informative but computationally costly search strategy just once up front to create an initial set of candidate solutions—then switch to a regular evolutionary algorithm to complete optimization more efficiently [Gaina et al., 2017, Shi et al., 2020a,b]. Hernandez-Diaz et al. [2008], for example, demonstrate this approach as a means of efficiently hybridizing a gradient-based heuristic with evolution. And Friedrich and Wagner [2015] approach multi-objective optimization by running a single-objective EA (namely CMA-ES) to solve several weighted variations of an original multi-objective problem—using the solutions obtained by the single-objective method to seed the population of a multi-objective algorithm that produces the final solution.

Broader reviews of population initialization strategies for meta-heuristic algorithms and EAs are given by Kazimipour et al. [2014] and Agushaka and Ezugwu [2022]. The main takeaway is that *instances*—that is, individual candidate solutions to a problem—can be an effective way of representing and injecting heuristic knowledge into an algorithm. I will return to this topic below when considering population-seeding approaches that transfer knowledge across distinct tasks. Solution instances by themselves, however, are limited in the information they can represent and convey—since they encode only specific points in a

search space.

Representation Learning in a Single-Task Setting

A second provocative direction for configuring evolutionary algorithms is to focus on the representation. Solution representations (a.k.a. *encodings*) and reproductive operators form two sides of the same coin: by altering the structure or semantics of the data structure that reproductive operators operate upon, representations have a profound impact on the search trajectory of an algorithm.

In an evolutionary context, representations can be conceived of as a genotype-phenotype map, which maps some *genotype space* $\mathcal{G}$ to a *phenotype space* $\mathcal{P}$. Genotypes, however, are often abstract representations (such as a bitstring) that can be conveniently manipulated by evolutionary operators, but that have no obvious meaning by themselves as solutions to a problem. Importantly, then, in many applications, fitness functions are not defined over a genotype space, but instead over a phenotype space in which candidate solutions are described in a fashion that is natural to the problem domain. A fitness function for traveling salesman problems takes graph tours as input, for instance, while a fitness function for real-valued optimization accepts vectors in $\mathbb{R}^n$. The selection of a good representation for the problem domain is commonly seen as one of the most important components of EA design—a view that is consistent with biologists' understanding of the crucial role that genotype-phenotype relationships play in natural evolution [Gerhart and Kirschner, 2007].

It is relatively straightforward to use a method such as meta-evolution to automate the selection of numeric design decisions such as population size, parameters for mutation, etc. [de Landgraaf et al., 2007, Grefenstette, 1986, Luke and Talukder, 2013]. The hyper-heuristics community, moreover, has shown that it is often possible to optimize more complicated parts of a heuristic algorithm's behavior by searching for combinations of operators or heuristics that complement each other's strengths [Burke et al., 2010]. The design of elaborate, domain-specific reproductive operators or encodings, however, is often viewed

as too complex and challenging to approach automatically [De Jong, 2007]. As such, despite the large literature on automatic EA configuration, and despite the central importance of genetic representations to EA performance, very little work has attempted to adapt a representation to a problem or class of problems.

Some early exceptions to this oversight are the efforts of Simões et al. [2014] and Watson et al. [2014] on using feed-forward neural networks and genetic regulatory networks (GRNs), respectively, to evolve genotype-phenotype maps for evolutionary algorithms. Simões et al. conduct an exploratory study of how several neural network encodings radically alter the local neighborhood of a mutation operator that takes a fixed step size in a real-valued genotype space. While they suggest that fruitful mappings of this kind can be automatically evolved, they do not propose a mechanism for doing so. Watson et al., meanwhile, show that GRN representations can be evolved effectively along with solutions to a problem in an online fashion, and that through repeated exposures to complex problems in this way, GRNs are able to evolve rich encodings that convey compositional information about sub-solutions that can be combined in novel ways to generate promising solutions. Some striking examples of this are demonstrated by Kouvaris et al. [2017], who extend the same GRN methodology.

2.2.3 Overview of Evolutionary Transfer Methods to Date

The vast majority of evolutionary algorithm applications are *single-task* applications, which approach one problem and do not reuse information from previous problems. The main exception to this rule has been automated algorithm configuration (AAC) methods that optimize the performance of an algorithm on a *class* of problems [López-Ibáñez et al., 2016, Tauritz and Woodward, 2018]: the goal of this kind of AAC approach is to learn or evolve a single algorithm configuration that performs well on a set of problem instances sampled from a distribution, and/or which generalizes robustly to perform well on unseen instances sampled from the same distribution.

Evolutionary knowledge transfer (EKT) is distinguished both from single-task evolution

and from AAC methods that focus on a fixed problem class. EKT aims to re-use information that is automatically gleaned from *source tasks* to improve an evolutionary algorithm's performance (i.e., solution quality, efficiency, or both) on one or more *target tasks*.[17] While forms of knowledge transfer have been heavily studied in several areas of machine learning (such as pattern recognition [Caruana, 1998], meta-learning [Smith-Miles, 2008], and reinforcement learning [Taylor and Stone, 2011]), attempts to accrue or reuse knowledge as an aid in solving search and optimization tasks more generally have been less common.

Evolutionary algorithms that use knowledge transfer, however, have begun to be intensely studied for the first time in the last few years. Until about five years ago, EKT algorithms tended to be isolated, one-off experiments that generated little general insight beyond a single application. Today, however, the community has begun to identify several significant threads of inquiry and to build research programs that systematically build insight into aspects of how to build effective knowledge transfer systems. The seminal work of Gupta et al. [2016c] in particular has spurred a great deal of recent work that specifically investigates multi-task optimization.

In this section I will give a brief survey of the diversity of EKT approaches that have been implemented to date—broken down in terms of 1) basic models of transfer (such as sequential transfer vs. multi-task transfer), 2) application areas, and 3) knowledge representations for transfer. I do not engage in a comprehensive review of this emerging area, however—several surveys of EKT or narrower sub-disciplines (such as multi-task EAs) have now been written which cover the field fairly well [Gupta et al., 2018, Liu et al., 2017, Xu et al., 2021b].

[17] How exactly this differs from AAC methods that train on many instances of a problem class (and generalized to new instances) is a difficult definitional exercise. I don't try to answer here whether the two are mutually exclusive, or whether they are overlapping algorithm categories that bear a complex "family resemblance." While AAC does gather experience from multiple tasks, perhaps it is best viewed as producing a kind of robustness (rather than knowledge transfer). One way to view AAC vs. transfer is to say that an AAC/hyper-heuristic approach performs knowledge transfer if and only if it attempts to learn an algorithm configuration on one task distribution that generalizes to instances that are sampled from a second, distinct distribution of tasks. This is similar to how knowledge transfer is typically defined in machine learning (i.e., in terms of two distinct distributions of instances)—and an example of this kind of knowledge-transfer hyper-heuristic is the work of Hart and Sim [2014] on heuristics for scheduling problems.

Heuristic Precursors to EKT

Efforts to incorporate knowledge reuse into search algorithms date at least to Sussman's `HACKER` system in the 1970s, an influential early example of a "problem solver whose performance improves with practice" [Sussman, 1975]. `HACKER` leveraged a library of solutions that it had generated to previously-encountered planning tasks to improve its ability to generate solutions to new tasks. This approach, which Sussman demonstrated in the famous Blocks World, can be seen as a precursor to later case-based reasoning systems. The ability to store, transfer, and repurpose learned problem-solving knowledge for new tasks continued to be a feature that some researchers hoped to incorporate into heuristic search algorithms throughout the early days of artificial intelligence [Carbonell, 1986, Langley, 1985]. One of the most ambitious early attempts at such a knowledge-acquisition system was developed in the context of the Soar cognitive architecture [Laird, 2012, Laird et al., 1987]. Drawing on the *chunking* theory of learning in humans, mechanisms for learning hierarchical sets of stimulus-response rules were integrated into Soar as a means of giving it the ability to improve its own performance through "practice," and to acquire a repertoire of knowledge that could be searched to allow the system to perform well on a wide variety of tasks (providing a so-called "universal weak method") [Laird et al., 1986].

While most of these historical approaches to knowledge retention and transfer in heuristic methods have had little direct influence on subsequent research on transfer in evolutionary algorithms, some of these early investigations did mature enough to begin investigating some significant and general research problems that are relevant to modern meta-heuristics. One such concern arises when a system aims to grow a repertoire of prior knowledge from multiple past experiences that can be applied to new problems. Such a system risks being "swamped by an ever-increasing repertoire of stored plans," as Fikes et al. [1972] put it in a now-classic paper that identified the *utility problem*. This concern highlights the importance that forgetting [Markovitch and Scott, 1988]—and especially strategic forgetting [Kennedy and De Jong, 2003]—may play in allowing a system to effectively make use of its past

experience.

The utility problem is the opposite of the *catastrophic forgetting* problem, in which a model is continuously exposed to new tasks, but tends to quickly forget any useful information it learned on prior tasks as new experiences overwrite past experiences, making it difficult to learn any generalizable skills. Catastrophic forgetting has gotten much more research attention in recent AI literature [Beaulieu et al., 2018, Rusu et al., 2016, Serrà et al., 2018, Wen and Itti, 2018], in part because of the dominance of artificial neural network models, which are naturally more prone to catastrophic forgetting than to the utility problem. The utility problem has continued to be studied in contexts other than deep learning, however—such as in the case-based reasoning literature, where the less obscure term *memory swamping* has become favored [Salamó and López-Sánchez, 2011]. Because population-based approaches to knowledge transfer and lifelong learning usually use a more granular approach—in which solutions to or knowledge from past tasks are stored as discrete entities—it seems that the utility problem (rather than catastrophic forgetting) may be poised to gain renewed significance in an optimization context.

Models of Transfer

Terms like *task*, *domain*, and *knowledge transfer* are often used in the EC literature without being defined. As I discuss approaches that have been devised to EKT, however, it is helpful to start out with a clear interpretation of these terms as a reference point.

Let a *task* $T = (\mathcal{S}, f(\cdot), F)$ be a solution space $\mathcal{S}$ and a fitness function $f : \mathcal{S} \mapsto F$ to be optimized. The fitness space F may be $\mathbb{R}$ (in which case the task is single-objective), or it may be multidimensional (in which case the task is multi-objective).

Furthermore, let a *domain* $D = (\mathcal{G}, \phi)$ be a genotype space $\mathcal{G}$ along with a mapping (or "encoding") $\phi : \mathcal{G} \mapsto \mathcal{S}$ that maps genotypes to *solutions*.

Now, taking inspiration from a similar definition of transfer *learning* given by Pan and Yang [2010], I can give the following definition of transfer optimization:

Definition 2.1. ***Transfer optimization*** *seeks to solve (i.e., optimize/satisfice/improve) a* target task T_t *in a* target domain D_t *using knowledge from at least one* source task T_s *solved in a* source domain D_s.

Evolutionary knowledge transfer (EKT) can now be interpreted as the application of evolutionary algorithm that is used to perform transfer optimization.

A number of sub-families of transfer optimization (and thus EKT) can be specified based on this definition. The most important division is between *sequential* transfer and *multi-task* transfer. Sequential transfer first "trains" on one or more source tasks, and then "tests" by transferring knowledge to one or more target tasks. Multi-task transfer, by contrast, solves more than one task simultaneously—allowing "omni-directional transfer" [Gupta et al., 2018]. The distinction between source and target tasks is blurred in multi-task transfer, and is often determined by the context of a particular operation at a particular time in an algorithm. A third problem-solving approach, *task-switching transfer*, alternates back and forth between two or more tasks while only working on one at a time—this can be seen as a kind of intermediate, sharing aspects of both sequential and multi-task problem-solving [Parter et al., 2008].

Orthogonally to these designations, transfer optimization approaches may be *homogeneous* (source and target tasks share the same domain, $D_s = D_t$) or *heterogeneous* ($D_s \neq D_t$). *Objective-heterogeneous* systems, furthermore, were recently introduced by Xue et al. [2021], and consider the case where the fitness spaces for each task have differing dimensionality (such as when transferring from a single-objective source task to a multi-objective target task).

Sequential transfer methods are arguably the most natural, but have been relatively rarely studied. Early examples of sequential transfer in the EC community appeared in a genetic programming context: Seront [1995], for instance, noted that John Koza's mechanism of creating automatically defined functions in genetic programming [Koza, 1994a] offered a potential means of learning, encapsulating, and reusing a library of concepts across program

synthesis tasks.

Multi-task transfer has been a latecomer in the evolutionary computation community—but has rapidly grown to become the most heavily studied form of EKT to date. Early examples of multi-task evolutionary algorithms originated in interdisciplinary simulations of natural systems [Arthur and Polak, 2006, Lenski et al., 2003]. These don't appear to have directly inspired any follow-up work in the artificial intelligence community. Instead, knowledge transfer concepts have typically entered EC from the machine learning community. Jaśkowski et al. [2008], for instance, introduced one of the first examples of a multi-task EA in a genetic-programming-based computer vision application (though Moriarty and Miikkulainen [1997] proposed multi-tasking much earlier as a natural outgrowth of their work on co-evolution of neural networks for concept learning).

Multi-task evolutionary algorithms came into their own with the introduction of the multi-factorial EA (MFEA) by Gupta et al. [2016c]. This innovative approach to multi-tasking in evolutionary systems inspired a significant burst of work on multi-tasking and knowledge transfer in evolutionary algorithms—much of it centered on researchers at Nanyang Technological University and their collaborators elsewhere in Asia—that has continued unabated ever since [Bali et al., 2017, Chandra and Gupta, 2016, Chandra et al., 2017a,b, Cheng et al., 2017, Da et al., 2016a,b, 2017a, Gupta et al., 2016a, Gupta and Ong, 2016, Gupta et al., 2016b, 2017, Jiang et al., 2016, Lim et al., 2016, Sagarna and Ong, 2016, Wen and Ting, 2016, Yuan et al., 2016, 2017, Zheng et al., 2016, Zhou et al., 2016]. The MFEA assigns each individual in a single population a "skill factor" associated with a particular task. After initialization, individual fitnesses are evaluated only on the task associated with their skill factor. When parents are selected to produce offspring, however, parents from different skill factors may be chosen with some (tunable) probability—otherwise parents are chosen from within the same skill factor. This inter-skill-factor crossover mechanism—termed *assortative mating*—is what facilitates inter-task transfer in the MFEA. Overall, the MFEA's dynamics are effectively very similar to an island model that uses different fitness functions on different islands, although the separation between subpopulations is implicit in

the mating mechanism rather than explicit. So far the community has typically treated the MFEA as a distinct evolutionary paradigm—with multi-population multi-tasking models proper being treated as a separate (if interrelated) approach [ElSaid et al., 2020, Li et al., 2020, Tang et al., 2021, Zheng et al., 2019].

Beyond the algorithm itself, the MFEA's more general innovation was the introduction of a new bio-inspired formalism—the multi-factorial optimization (MFO) framework—for representing the fitness of individuals in a multi-dimensional space, where each dimension corresponds to a different task. Gupta et al. [2016c] take care to distinguish multi-factorial optimization from the Pareto-dominance formalism used in multi-*objective* optimization (MOO): the two are not equivalent, since MFO aims to find at least one high-quality solution to each task while sharing information—not necessarily to find a single solution that balances trade-offs among competing tasks (as in multi-objective optimization). The clear distinction between the MFO and MOO frameworks is underscored by the large number of papers that have now emerged investigating multi-task optimization for multi-objective optimization problems [Chen et al., 2022a,b, Da et al., 2016b, Gupta et al., 2016d, Yi et al., 2021, Zhang et al., 2019]—i.e., applications in which multiple, distinct multi-objective problems (each with their own set of multiple objectives) are solved in parallel. The two paradigms, however, do share significant overlap, and multi-objective optimization techniques have often been used as a means of solving multi-task problems [Jiang et al., 2017, Le et al., 2012, Li et al., 2021].

A full review of multi-task optimization and related efforts is beyond my scope here (see Gupta et al. [2018] and Xu et al. [2021b] for more comprehensive treatments), as my experiments in this dissertation will focus more on sequential transfer. But I will highlight a few key features of this literature in subsequent sections below—such as the efforts that have been made to manage the problem of negative transfer.

Application Areas for EKT

Knowledge transfer relies fundamentally on the structure of the application domain: and specifically on the availability of relevant source tasks. Research on evolutionary knowledge transfer has focused on three main classes of application domain—namely combinatorial optimization, machine learning, and robotics.

In the vast majority of these applications, EKT has been applied to just a few tasks that have been hand-selected in advance by humans. As a result, the success of these applications may say as much about how the tasks they were tested on were selected by a human as they do about how well-suited the algorithm is to the application domain. Occasional papers have scaled knowledge transfer to larger sets of tasks (greater than 3) [Arthur and Polak, 2006, Lenski et al., 2003, Martinez et al., 2021, Sagarna and Ong, 2016, Yuan et al., 2016], but achieving many-source and many-task transfer of this kind raises a number of challenges that I will describe in more detail in the sections below.

Many researchers have focused intently on combinatorial optimization as a setting for knowledge transfer [Iqbal et al., 2014, Mei and Zhang, 2016, Zhou et al., 2016]. Early examples of EKT involved applications to scheduling problems [Cunningham and Smyth, 1997, Hart and Sim, 2014, Louis and McDonnell, 2004], and some significant efforts have recently developed sophisticated new methods of solving job-shop scheduling problems with multi-task genetic programming [Zhang et al., 2021a,b,c]. Feng et al. in particular developed a series of approaches to capacitated arc-routing (CARP) and vehicle-routing problems (CVRP) that rely on sequential evolutionary transfer [Feng et al., 2012, 2015a,b].

A second major application thread has been focused on using evolutionary knowledge transfer to help solve supervised pattern recognition and regression tasks (ex. [Chandra and Gupta, 2016]). Many genetic programming systems, such as the multi-task approach of Jaśkowski et al. [2008], have taken this approach, and work that focuses on on supervised learning applications and the evolution of neural networks in particular [Chandra and Gupta, 2016, Fernando et al., 2011, 2017, Mouret and Doncieux, 2009] continues to appear in

the annual IEEE Congress on Evolutionary Computing (CEC) special session on "Transfer Learning in Evolutionary Computation," among other venues [Chandra et al., 2017b, Haslam et al., 2016, Iqbal et al., 2016, Jaśkowski et al., 2014, Krawiec and Wieloch, 2010, Wen and Ting, 2016].[18] Genetic programming (GP) with knowledge transfer has occasionally been studied without reference to machine learning tasks—beginning with the multi-task PushGP systems of Bladek and Krawiec [2016] and Soderlund et al. [2016], which synthesize implementations of several string-processing functions simultaneously. Most EKT approaches that use GP, however, have been oriented at pattern recognition.

The third natural application area for EKT has been evolutionary robotics. As I discussed above in section 2.2.1, one significant thread of this work has been inspired by the novelty search mechanism introduced by Lehman and Stanley [2011], and which has inspired a large family of "quality-diversity" algorithms that sometimes overlap with knowledge transfer methods [Pugh et al., 2016]. In particular, the family of "illumination algorithms" introduced by Mouret and Clune [2015]—and particularly their MAP-elites method—has led to a number of multi-task algorithms which have been applied to robot control tasks [Huizinga and Clune, 2018, Nguyen et al., 2016, 2015a, Norstein et al., 2022, Wang et al., 2019]. Independently of this work, Kelly and Heywood [2017] have developed a "tangled program graph" representation that permits a significant degree of re-use of sub-programs across multiple tasks in genetic programming. After first having it demonstrated that this GP representation could perform as well as major reinforcement learning approaches (namely deep Q-learning [Mnih et al., 2013]) on standard Atari video-game-playing benchmarks, this approach has been further developed with additional features to enhance its performance on visually intensive reinforcement learning tasks [Bayer et al., 2022]. Finally, coming from the multi-task optimization community, an early demonstration of multi-task optimization was the multi-robot path planning work of Ong and Gupta [2016], and Martinez et al. [2021]

[18] It's worth noting that when we apply evolutionary algorithms to transfer learning in machine learning, the resulting systems can be viewed as *both* a transfer optimization task *and* a transfer learning task. One definition emphasizes that two different optimization problems must be solved for the sake of model-fitting on different tasks, while the other emphasizes that two different underlying datasets or distributions are at work.

have used a variation of the MFEA to evolve neural networks for deep reinforcement learning tasks for 3-D robot arm manipulation.

While combinatorial optimization, machine learning, and robotics have been the main application areas for EKT, EKT approaches and multi-task EAs have been applied to other application areas as well (see, for example, the recent survey of many applications given by Gupta et al. [2022]). Some of these include continuous optimization [Bali et al., 2017, Zheng et al., 2016], constrained optimization [Lim et al., 2016], and software test generation [Sagarna and Ong, 2016].

Knowledge Representations for EKT

Whether a sequential or multi-task algorithm is being applied to a small number of tasks or to many tasks, all EKT methods require some means of representing and transferring knowledge gained on one task to another. The strategies for achieving this closely follow the categories of heuristic knowledge representation that I discussed above in section 2.2.2.

For example, model-based approaches to knowledge transfer include methods based on learning and reusing the parameters of distribution models from estimation of distribution algorithms across tasks (ex. [Shahriari et al., 2015, Swersky et al., 2013, Zhang et al., 2019]). Algorithms of this kind have only recently been developed—based for example on mixture models or CMA-ES [Li and Li, 2021]—and their advantages or limitations in comparison to other EKT approaches are not yet well understood. Model-based transfer approaches based on surrogate models have also recently been developed. Ji et al. [2021], for example, use multiple surrogate modeling techniques to convert a single-task optimization problem into a multi-task optimization problem—an example of a broader pattern in which multi-task optimization can be used as a means toward solving single-task problems [Gupta et al., 2018].

Instance-based transfer approaches, however, have been especially common. These approaches use whole (or partial) solution instances themselves as a representation of transferable information. Early examples of instance-based transfer have included "case retrieval"

and "case injection" methods for TSPs and job-shop scheduling [Louis and McDonnell, 2004, Oman and Cunningham, 2001]. The simplest instance-based transfer model is *population-seeding transfer*. Just as in the single-task case that I survey in section 2.2.2, population-seeding transfer inserts solutions that have been saved from prior tasks into the initial population of an EA when it solves a target task. Potter et al. [2005], for example, use population seeding to evolve complex robot behaviors by initializing evolutionary populations with solutions to prior, related tasks. Approaches of this kind first appeared very early in heuristic reasoning systems, such as in the SAMUEL system of Ramsey and Grefenstette [1993] (who argued that "genetic learning systems need not learn from scratch.").

Instance-based transfer is the dominant approach to knowledge transfer in evolutionary computation at the present time. Multi-factorial algorithms, multi-task island models, population-seeding algorithms, etc., all typically rely on solution instances as the unit of transfer (though, as I will discuss below in section 2.2.5, strategies for combining instances from related problems have become quite sophisticated). Beyond population seeding, other instance-based algorithms have included crossover-based reuse [Jaśkowski et al., 2014] (where a special crossover operator is able to periodically select parents from a fixed archive of solutions to past problems), injection-island EAs [Eby et al., 1998] (a specialized island model in which solutions to low-fidelity simulation models are fed periodically to higher-fidelity models in a feed-forward fashion), and heterogeneous island models [Skolicki, 2007].

2.2.4 Transferability and Problem Classes

> In reality, all arguments from experience are founded on the similarity which we discover among natural objects, and by which we are induced to expect effects similar to those which we have found to follow from such objects. And though none but a fool or madman will ever pretend to dispute the authority of experience, or to reject that great guide of human life, it may surely be allowed a philosopher to have so much curiosity at least as to examine the principle of human nature, which gives this mighty authority to experience, and makes us

> draw advantage from that similarity which nature has placed among different objects.
>
> —David Hume, *An Enquiry Concerning Human Understanding*, IV.2, 1748.

There are three fundamental *ingredients to successful transfer* that I will refer to repeatedly throughout this dissertation (Figure 2.7). The first is the *problem class* that an EKT method is being applied to. If a problem space tends to produce problems that have little to no relationship to one another whatsoever, then transfer is unlikely to be successful. Second, even on problem classes where similar problems do occur—such similarity may occur with fairly low frequency. *Source selection* is therefore a vital component of knowledge transfer: selecting knowledge sources, within a wider set of potential choices, from which to transfer information. Finally, assuming that suitable source tasks are in place, an effective *transfer strategy* is needed in order to exploit the opportunity for information reuse. I begin here by discussing the first ingredient—problem classes—before addressing the remaining two below.

Helpful or harmful relationships among tasks ultimately arise from the decisions that are made about what source and target tasks are chosen for an EKT algorithm to be applied to. Even at their smartest, EKT algorithms are limited by the potential for positive transfer that is present in the task set they are given. And this in turn is limited by the degree and frequency with which instances of the problem class that is being considered display opportunities for transfer.

With the exception of the recent trend toward using "model zoos" of pre-trained models for computer vision and other machine learning domains [Shu et al., 2021, Such et al., 2018], for the most part AI practitioners are not accustomed to keeping databases of past problems at the ready for their algorithms to learn from. More importantly, it is not clear what kinds of problems or problem classes are likely to benefit from transfer, and which are not—or what kind of guidelines practitioners can follow when deciding to use transfer methods or gathering problems to act as knowledge sources.

While I have reviewed examples of successful EKT applications in robotics, combinatorial

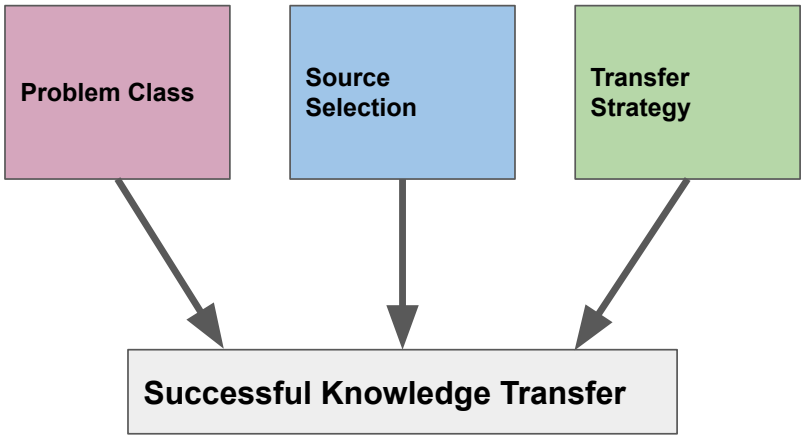

Figure 2.7: The scheme of "three ingredients" of successful evolutionary knowledge transfer that I use to organize the problems studied in this dissertation.

optimization, and machine learning, it remains difficult to say in advance what kinds of domains and problem classes knowledge transfer is an appropriate strategy for. Reporting bias is likely to impact the community's picture of transfer's usefulness as well—as task sets that benefit from transfer are naturally more likely to be published, whereas work that finds transfer to be ineffective on a given task set will be less likely to be published.

One way to phrase this problem is by asking what the probability is that, for a given problem class, two (or more) randomly chosen problem instances will display a positive, negative, or neutral transfer relationship. I do not know of any work that has studied transferability in evolutionary algorithms through this lens.[19] Asking this question raises the possibility that some problem classes may display no potential for transfer on average—particularly if results akin to the no-free-lunch theorems for optimization hold for transfer [Wolpert and Macready, 1997].

[19]This idea does bear some remote resemblance, however, to the recent trend in machine learning of viewing transfer through the lens of *taskonomy*: that is, the systematic study of problems (or problem classes) and how they are related to one another in terms of knowledge transfer potential [Zamir et al., 2018].

No Free Lunch Theorems and Transfer

The original no-free-lunch theorems (NFLTs) for optimization algorithms were proved by Wolpert and Macready [1997]. This work emerged from Wolpert's wider program, which investigated the Humean question of when machine learning and search algorithms can [Wolpert, 1996a] and cannot [Wolpert, 1996b, Wolpert et al., 1995] make valid inductive inferences from incomplete data (see Adam et al. [2019] for a recent review of these theorems and their broad influence on AI and machine learning). Wolpert and Macready show formally that, over the set of all possible objective functions $f : \mathcal{X} \mapsto \mathcal{Y}$ that map a given solution space $\mathcal{X}$ to a set $\mathcal{Y}$ of performance values (or, in the "sharpened" NFLTs of Schumacher et al. [2001], over any set of such functions that are closed under permutation of their values), there can be no "*a priori* distinctions" between algorithms: that is, no algorithm will obtain better performance values than any other when averaged over all possible $f \in \mathcal{F}$, where $\mathcal{F} = \mathcal{Y}^{\mathcal{X}}$.

The classic no-free-lunch theorem for optimization only covers algorithms that are applied to a single, static (unchanging) fitness function. At the same time, however, Wolpert and Macready [1997] proved a similar theorem that applies to dynamic problems: problems whose objective function changes over time. This *time-dependent NFLT* shows that, given a problem whose initial objective function at time $t = 1$ is f_1, no *a priori* distinctions among algorithms will occur when averaging over all possible time-dependent mappings $T : \mathcal{F} \times \mathcal{N} \to \mathcal{F}$. These mappings define how the objective function that begins in the state f_1 evolves over time.

Optimization algorithms that use knowledge transfer can be seen as a special case of time-dependent problems. Consider a sequential transfer algorithm, for example, that executes m iterations to learn about (i.e., to solve or otherwise analyze) a source task f_s, and then

switches to solving a target task f_t. This scenario can be expressed as a time-varying function

$$T_{f_s,f_t}(t) = \begin{cases} f_s & \text{if } t \leq m, \\ f_t & \text{otherwise .} \end{cases} \tag{2.1}$$

So knowledge-transfer problems in optimization can be expressed as a time-varying objective. But in imposing this constraint on the possible values of the dynamic schedule T, I have violated the core assumption of the time-dependent NFLT, which only holds when considering *all* possible mappings $\mathcal{F} \times \mathcal{N} \to \mathcal{F}$, rather than only mappings that follow the more constrained form of Equation 2.1. The time-dependent NFLT, moreover, does not rule out *a priori* distinctions among algorithms when averaging across different choices of $f_1 = f_s$. For these reasons, the classic theorems of Wolpert and Macready [1997] by themselves are not able to model the questions that I should like to ask about how knowledge transfer may perform in different problem classes.

2.2.5 Negative Transfer and Source Selection

I now turn to the second key "ingredient" of successful transfer in the scheme of Figure 2.7: source selection. The most glaring and fundamental question in transfer (whether for learning or optimization) is "where does useful source information come from?" The promise of knowledge transfer is founded on the hypothesis that 1) effective methods are available to exploit source tasks that are similar to a given target task, and 2) that source tasks in the world can be found that are exploitable in this way.

The problem of how to effectively exploit useful source tasks is intimately tied to the question of how to avoid *negative transfer*. Negative transfer occurs when knowledge transfer has an adverse effect on the performance of an algorithm instead of a beneficial effect—i.e., when the solution quality or efficiency of a method is better when no transfer is used at all than when transfer is attempted. Because the potential that two or more problems have

for positive (or neutral) transfer is typically not known for certain in advance—and because task similarity can be difficult to judge—the possibility of negative transfer is ubiquitous in applications. Many approaches to knowledge transfer focus, then, on ensuring that negative transfer can be preempted and mitigated.

Task-Level Similarity Estimation

A variety of adaptive mechanisms have recently been introduced to combat negative transfer in evolutionary knowledge transfer. Most of these methods have been studied in the context of multi-task optimization. In particular, Bali et al. [2019] introduced an influential online method based on probabilistic mixture models and Kullback-Leibler divergence for estimating a matrix of task-specific transfer parameters in their MFEA-II algorithm. These weights serve to control the degree to which any pair of tasks in the system exchanges information—with more similar tasks sharing information more often than less similar tasks. MFEA-II thus effectively acts as a *source-selection* method. This landmark revision to the MFEA of Gupta et al. [2016c] has helped to inspire a series of similar approaches. Martinez et al. [2021], for instance, extend the transfer-parameter matrix approach of Bali et al. [2019] into a transfer-parameter tensor for applications that evolve deep neural networks for reinforcement learning tasks. The extra dimensions of this tensor allow their A-MFEA-RL algorithm to learn transfer parameters that are not only specific to a source-target task pair in a multi-task system, but are also specific to particular *layers* of the multi-layer neural networks that are being evolved.

Gene-Level Similarity Estimation

While MFEA-II and related algorithms focus on source selection at the task level, increasing or decreasing the frequency with which solution instances are transferred from particular tasks, other approaches have examined strategies for selectively *constructing* compound solution vectors that borrow gene values from a variety of different tasks. These methods

perform source selection on a finer level, picking and choosing individual genes (i.e., variables) from different sources. Ma et al. [2021], for example, introduce a gene-level approach to estimating gene similarity based on Kullback-Leibler divergence. This allows them to construct a specialized crossover operator that pulls specific genes from parents that are optimized for different source tasks, based on which source-task subpopulations in the multi-task system display the most similarity to a given target-task subpopulation with respect to each individual gene. Chen et al. [2022b] have recently developed this idea further, by introducing another solution-construction strategy which borrows genes from many tasks, and by perturbing the hybrid individual that is constructed in this way to generate multiple offspring.

Other Source Selection Methods

Both these classes of adaptive multitasking—the task-level similarity estimation mechanisms of MFEA-II and the gene-level similarity mechanisms introduced by Ma et al. [2021]—have been developed primarily in the context of vector-based solution representations (such as real-valued optimization). Noting that these approaches don't generalize trivially to more complex evolutionary representations, Zhang et al. [2021a] develop their own adaptive multi-task system for genetic programming, based on a phenotypic measure of similarity between genetic programming trees.[20] They apply this approach to evolving heuristics that solve dynamic job-shop scheduling problems in combinatorial optimization, showing favorable performance over a single-task approach and the MFEA.

Most of these adaptive approaches for avoiding negative transfer rely on comparing different subpopulations for different tasks in a multi-task setting—typically by applying Kullback-Leibler divergence in some fashion as a measure of similarity between distributions of solution phenotypes or genes. This approach exploits the fact that several tasks are solved simultaneously in multi-task optimization, so task-specific subpopulations can be

[20]Parse-tree-style representations are one of the main traditional approaches to using evolutionary algorithms to directly evolve computer programs [Koza, 1994b]—so measuring the similarity between individuals or populations in tree-based GP requires defining a similarity measure across tree structures.

compared online in the midst of the run (before they converge). One variation on this theme is the source-selection method of Zhang et al. [2019], which utilizes estimation of distribution algorithms (EDAs), and uses a Wasserstein distance to directly compare the estimated distributions as they optimize different tasks—but the basic principle of doing distributional comparison while multitasking is the same.

This distributional approach may be misleading or inapplicable in some cases. Having populations with similar distributions in some space at some point during an evolutionary run may not imply that two problems are significantly similar, and conversely, similar problems may not always share similar evolutionary distributions. For example, in genetic programming, solutions to two problems might have very different phenotypes, but a subcircuit from one problem may nonetheless be useful as a partial solution to another problem. To my knowledge, possibilities of this kind have yet to be considered in literature on adaptive solutions to negative transfer.

Adaptive EKT methods have also primarily been developed in ways that are specific to multi-task optimization. Less work has investigated how negative transfer can be avoided in a sequential transfer setting.

Many-Source Transfer as Human-Machine Teaming

Ultimately, solutions to source-selection and negative transfer open up a greater possibility for human-machine teaming in evolutionary knowledge transfer. As long as EKT requires a human to correctly and accurately identify a suitable source task that leads to positive transfer on a given target task, this kind of technology will tend to be limited to niche applications where transferability is "obvious" (as I discussed in section 2.2.1). A promising route to reduce the reliance of EKT on humans' source-selection ability is to build algorithms that are able to use *many knowledge sources*—only a small percentage of which may be viable sources for transfer in a given application. This opens up the possibility of a relationship in which a human selects a (perhaps large) set of problems that they believe have some chance of being useful (either in general or for a particular target task), and then the algorithm

works with this material to narrow down and isolate the material that is indeed useful for a given target task.

Currently, most applications of EKT involve just two or three tasks, and these are typically manually curated in advance by a human based on their intuitions of problem similarity and where positive transfer might be possible. For instance, one might observe with Seront [1995] or Kurashige et al. [2003] that a robot that has used genetic programming to learn a sequence of actions that allow it to 'stand up' vertically might subsequently have an easier time learning to 'walk forward.' Likewise, Kelly and Heywood [2017] initially demonstrated their "tangled program graph" genetic programming system on small sets of reinforcement learning environments, consisting of three tasks each that a human has grouped together.

Applications that involve dozens or hundreds of tasks, however, offer an algorithm the opportunity to search for useful information among a wider set of sources. A number of authors have begun to experiment with multi-task optimization on larger task sets [Bladek and Krawiec, 2016, Huizinga and Clune, 2018, Martinez et al., 2021, Nguyen et al., 2016, Sagarna and Ong, 2016, Soderlund et al., 2016, Yuan et al., 2016]. Liaw and Ting [2017] refer to such applications, which aim to solve greater than three objectives simultaneously, as *many-task* optimization, in analogy to how the term many-*objective* optimization is used in the Pareto optimization community. Sequential (as opposed to multi-task) EKT methods that use many sources have rarely been built, but I apply the broad term *many-source* transfer to refer to sequential and multi-task approaches alike that use more than three sources tasks.

Scaling to many tasks is necessary to reduce the burden on humans to select perfect knowledge sources for transfer-based algorithms. It also seems to be necessary in order to exploit surprising or *serendipitous* kinds of transfer that the innovation-engine theory of transfer I discussed in section 2.2.1 arguably requires. "The key problem," conclude Stanley and Lehman [2015] from their experience with robotics and evolutionary art, "is that the stepping stones that lead to ambitious objectives tend to be pretty strange. That is, they

probably aren't what you would predict if you were thinking of only your objective." As the number of tasks increases, however, the cost and complexity of source selection necessarily becomes an increasingly important part of an EKT algorithm's performance. It is important, then, that problem classes exhibit the right kind of structure such that useful source problems do appear with some non-negligible probability—otherwise, many-source approaches will fall into the memory-swamping problem I covered in section 2.2.1. To date, however, little work has looked explicitly into how the performance of many-source methods scales with task set size, or how it hinges on the properties of a problem class.

2.2.6 Representing Knowledge for Transfer

The third major ingredient of successful knowledge transfer is the transfer strategy that is applied. The majority of work on EKT has focused on instance-based knowledge transfer. Knowledge representations, however, have the potential to represent knowledge in a more general way that may be much more widely applicable than a particular instance. Instance-based approaches—which are by far the most common in current work on EKT—may be suitable for applications where it is reasonable to expect problems to share a great many "identical elements." But solution instances by themselves are very limited in the kinds of information they can represent. A simple translation or rotation of the search space, for instance, could render such information moot. In particular, it seems that in many problems it may be desirable to transfer not just information about solutions that have been successful in the past, but also about what kind of search strategies should be used: what variables should be varied together, for example, or what kind of subcomponents have proved useful for building compositional solutions.

Representation-based knowledge transfer aims to provide this kind of solution. In section 2.2.2 I reviewed the small body of work that exists on learning representations for single-task evolutionary algorithms. Even less work has studied representation learning in a multi-task setting. Watson et al. [2014] and Kouvaris et al. [2017] offer one of the few examples to date of explicit representation learning for knowledge transfer. They show

that a genetic regulatory network representation of a genotype-phenotype map is capable of learning to repeat genetic patterns that have been useful on previous tasks (or on previous attempts to solve a single, highly multi-modal task). This system evolves a matrix representing the representation simultaneously with solutions, but mutates the matrix much more slowly than the solutions themselves (so that the matrix tends to encode long-term information, while the solution vectors act as the short-term solution instances).

Lenski et al. [2003] follow perhaps the simplest approach to the reuse of representational components in GP: they evolve a single individual that solves multiple Boolean function synthesis tasks simultaneously, and which has a complex executable structure that allows sub-solutions to be reused across multiple tasks. Different outputs of a single evolved program are assigned to different tasks, but the program is free to reuse function calls internally across tasks. Kelly and Heywood [2017] have used a similar general multi-task approach to evolve rule-based controllers for Atari video games. They found that, by reusing internal decision structures across multiple games, they were able to learn to play sets of three distinct games competitively with the same computational budget that was normally necessary to learn just one game independently. An alternative approach to re-use in graph structures can be found in the component-sharing model of Arthur and Polak [2006] (where logic circuits evolve by using whole solutions to other tasks *as subcomponents* themselves). Such a purely compositional approach to function synthesis has sometimes been called "endosymbiotic" evolution [Watson, 2006], in reference to the theory that early eukaryotic organisms evolved their organelles by symbiotically absorbing other prokaryotic organisms whole.

Beyond implicit examples of this kind, however (in which genetic programming approaches "automatically" enable a limited form of representation learning), little to no additional work has explicitly examined representation-based transfer in evolutionary computation.[21]

[21]This is a far contrast from the machine learning literature, where transfer based on learned representations is extremely common in both research and practice.

2.2.7 Research Questions

At a high level, I am motivated to the study of evolutionary knowledge transfer by the third speculative vision that I discussed in section 2.2.1: the hypothesis that evolutionary algorithms built out of simple algorithmic mechanisms can be used to build *innovation engines* much in the sense of Nguyen et al. [2015b], except by relying on a diversity of source tasks (rather than novelty mechanisms) to spur complex problem-solving. In order to understand the ingredients that may be necessary for artificial systems of this type to be possible, however, here my investigation is limited to preliminary studies of each of the three ingredients of EKT success—problem classes, source selection, and transfer strategies.

Here I present a number of research questions, toward which I will present several preliminary and exploratory investigations in Chapter 4 and Chapter 5. The excursionary nature of these investigations will contrast somewhat to the questions I presented in section 2.1.5, but through them, I establish a number of novel theoretical and empirical insights that advance the goal of understanding EKT mechanisms that may facilitate increasingly innovative meta-heuristic problem-solving systems in the future.

Problem Classes

In section 2.2.4 I discussed the central importance that the task set that a human selects for an evolutionary knowledge transfer experiment plays in its success or failure. Because this selection is likely to be heavily influenced by the broader problem class that tasks are selected from (for example, tasks may be arbitrarily selected from some historical database, or from the space of possible problem instances), the transferability properties of problem classes are important in determining whether an application space is a suitable setting for transfer to be applied.

Beginning at a foundational level, I will first use a combination of analytical proofs and experiments to examine the following two-part research question:

Research Question 4. ***Transferability within Problem Classes****:*

A) Do broad no-free-lunch theorems hold for evolutionary knowledge transfer (analogous to single-task optimization)?

B) What kind of problem classes is positive transfer in optimization likely to occur in with non-negligible probability?

Source Selection

Next, once a transfer-friendly problem class and task set is fixed, the task of source selection remains a challenge (so that negative transfer can be avoided). As I noted in section 2.2.5, most work on source selection for EKT has focused on multi-task problem-solving. A natural question when dealing with sequential problem-solving (i.e., based on historical experience) is whether there is a way to analyze the available source tasks to determine *a priori* which are good candidates for transfer in a sequential transfer model:

Research Question 5. ***Transfer Prediction****: Can sampling strategies for analyzing fitness landscapes yield effective predictors of sequential transferability between source and target tasks?*

Sequential transfer raises a second provocative possibility for the source-selection problem: if a large pool of source tasks is available, and a method is available that is good at weeding out sources of negative transfer, then it may be possible to use a *many-source transfer* algorithm to rapidly hone in on positive transfer sources:

Research Question 6. ***Many-Source Transfer****: Is many-source transfer a viable strategy for avoiding negative transfer?*

I posit that in some cases at least, a many-source variation of population-seeding transfer will meet this criterion.

I will present the results of Research Questions 4, 5, and 6 in Chapter 4, which focus on the broad question of transferability in problem classes when instance-based sequential transfer is applied.

Transfer Strategies

The final key ingredient of EKT success is the transfer strategy that is used to exploit information from a source task. As I discussed in section 2.2.6, it seems that knowledge formats that are based on representation learning have the potential to convey a great deal more generalizable knowledge than solution instances by themselves would typically be able to. But representation learning has rarely been studied (even for single-task optimization), so this hypothesis remains largely untested. Here I will pursue two kinds of representation-based evolutionary knowledge transfer.

First, I investigate a Boolean-function domain, where solutions are themselves executable objects with internal structure. This situation is analogous to neural network training, where the layered structure of a neural network allows earlier layers to learn a representation that is reused across various parts of later layers. Cartesian genetic programming (CGP) [Miller, 2011] is a popular evolutionary system that shares this phenomenon with neural networks. I use this to empirically examine the ability of a multi-task evolution approach based on hard parameter sharing to solve a set of Boolean tasks.

Research Question 7. ***Representation-Based Transfer for CGP****: Can genetic programming with graph structures facilitate multi-task learning via shared sub-circuits, analogously to how multi-task neural networks can learn via hard parameter sharing?*

Second, I will perform experiments in domains that naturally use a vector-phenotype representation. The aim here will to be show that a simple automated algorithm configuration approach based on meta-evolution is able to learn linear mappings from genotype to phenotype that produce useful and transferable knowledge:

Research Question 8. ***Representation-Based Transfer for Vector Phenotypes****: Can learning a genotype-phenotype map representation for a class of problems yield representations that generalize to problems with different global optima, and which are sampled from different classes?*

In this case I will be examining the task of learning a representation that works well for a *class* of problems (rather than a single problem instance)—and then test how that representation transfers to a new class.

Chapter 3: Asynchronous Parallelization of Evolutionary Algorithms

3.1 Research Plan

In this chapter I investigate several of the aspects of asynchronous evolutionary algorithm behavior and performance that I emphasized in my review of the field in section 2.1.

After a describing the asynchronous steady-state EA (ASEA) in detail in section 3.2 and presenting a study of how different initialization strategies affect its behavior, my first core research question, Research Question 1 (see section 2.1.5), focuses on *speedup* in asynchronous steady-state EAs. The simplest way to conceive of speedup is in terms of fitness evaluations per unit of time—throughput speedup. But true speedup takes solution quality into account—considering the number of fitness evaluations needed to reach a fitness threshold. I study both kinds of speedup in section 3.3, proving analytical results that quantify the former, and presenting experimental studies to examine the latter.

Concerns around evaluation-time bias in ASEAs arise naturally from this examination of their true speedup properties (Research Question 2). I conduct several empirical studies of evaluation-time bias in section 3.4, in particular examining the quasi-generational EA (QGEA) [Durillo et al., 2008, Fonseca and Fleming, 1998] to see if it offers a feasible middle ground between the advantages of ASEAs and more traditional generational EAs.

Lastly, in section 3.5 I turn to the question of whether ASEAs engage in "excess computation" when solving problems in which higher-quality solutions take longer to have their fitness evaluated (Research Question 3). I introduce a specialized selection operator (SWEET) into an ASEA and show empirically that it has desirable properties on this class of problems.

3.2 The General Asynchronous EA

Asynchronous master-worker EAs eliminate the primary source of idle time in parallel evolutionary processes by ensuring that, any time a processor has become free, a new offspring individual is immediately produced to take its place. Almost all asynchronous global-population EAs to date have been built on the steady-state model: individuals are integrated into the population one-at-a-time immediately after their fitness has been evaluated, in the style of a $(\mu + 1)$-EA.

A general pseudocode template for asynchronous evolution is described in Algorithm 2 from the perspective of the master processor. The abstract template defined by this algorithm describes the class of algorithms that I study throughout this chapter. Algorithm 2 is abstract enough to admit of a number of important variants that can be implemented by different choices of its subroutines—portions of this chapter will investigate some of these variations.

Algorithm 2 The General Asynchronous EA

```
1: function asynchronousEvolution(μ, T, steps)
2:     P ← asyncInitialize(μ, T)          ▷ Initialize the population asynchronously
3:     for i ← 0 to steps do              ▷ Begin steady-state evolution
4:         send(breedOne(P))
5:         integrate(nextEvaluatedIndividual(), P)
```

First, the algorithm calls an asynchronous evaluation procedure that generates an initial population of individuals and begins evaluating them (line 2). I will study several possible choices of initialization procedure below in section 3.2.2, but I assume that regardless of exactly how initialization proceeds, the asyncInitialize() procedure always leaves exactly one processor free when it returns. So if there are T processors, $T - 1$ of them will be busy at the moment that Algorithm 2 advances to line 3.

Evolution then proceeds in the **for** loop, which generates an offspring individual and

sends it to the free processor (line 4). The BREEDONE() procedure here is responsible for performing both parent selection and applying reproductive operators to generate a single offspring; the details of these are application-specific, and I will explain my choice of reproductive operators later in the sections for each experiment that I present below. Line 5 of the algorithm then blocks by calling NEXTEVALUATEDINDIVIDUAL(), which waits until an individual completes processing and then returns it. The individual then competes for a place in the population—the details of this being handled by the INTEGRATE() procedure—and, since one processor is now once again free in the computational environment, the loop takes us back to line 4 to generate a new offspring.

3.2.1 The Asynchronous Steady-State EA

When the INTEGRATE() procedure takes the form of a steady-state survival selection process, I refer to the resulting algorithm as a simple asynchronous EA (following terminology I introduced in Scott and De Jong [2015]), or an asynchronous steady-state EA (abbreviated just **ASEA**). Such a procedure is given in Algorithm 3. Here, the newly evaluated individual `ind` competes against an individual chosen by SELECTONE()—if the offspring individual is BETTERTHAN the other, then it replaces the selected individual in the population. I typically choose to consider one individual to be BETTERTHAN() another if its fitness is greater than *or equal to* the other.[1]

Algorithm 3 Steady-state insertion into a population

```
1: function INTEGRATESTEADYSTATE(ind, P, μ)
2:     if |P| < μ then                                  ▷ Population not yet full?
3:         P ← P ∪ {ind}                                ▷ Add the individual
4:     else
5:         replaceInd ← SELECTONE(P) ▷ Select a random or poor individual to replace.
6:         if BETTERTHAN(ind, replaceInd) then    ▷ Replace it if the new one is better
7:             P ← (P − {replaceInd}) ∪ {ind}
```

[1] This nuance ensures that change can still occur in the population even when fitness does not change.

Of the five abstract procedures that appear in Algorithm 2, the INTEGRATE() and ASYNCINITIALIZE() methods are especially important. I will consider alternative choices of each of these later in this chapter. In section 3.4 I will consider an alternative choice of INTEGRATE() that dispenses with the steady-state selection of Algorithm 3 and instead implements a quasi-generational EA (QGEA). Right now, however, I will take a closer look at the ASYNCINITIALIZE() step.

3.2.2 Initialization Strategy Experiments

The initialization step of asynchronous evolution is more important than it may at first appear.[2] In a generational algorithm, initialization is straightforward and routine: an initial population of individuals' genomes is generated (often via uniform random sampling of gene values), and then each individual has its fitness evaluated before the next generation begins to be generated. In asynchronous algorithms, however, strategies for asynchronously initializing a starting population can have profound and long-lasting effects on a run's problem-solving trajectory.

When speaking with other practitioners who use ASEAs, I have found that there is no clear consensus on how, exactly, a population ought to be initialized (that is, evaluated) *asynchronously*. But the different initialization procedures that practitioners choose when implementing an ASEA can sometimes have significant performance implications. In this section I examine some pitfalls that different initialization strategies may encounter, and give a preliminary assessment of their respective performance benefits.

Regardless of these strategies, however, all asynchronous initialization strategies by definition involve some possibility of a re-ordering effect in evaluations. In section 3.4 below, I will show that this effect at initialization time in particular—far more than at other points in the evolutionary process—is responsible for evaluation-time bias in the behavior of asynchronous algorithms that follow the template of Algorithm 2.

[2]The results presented in this section have not previously been published.

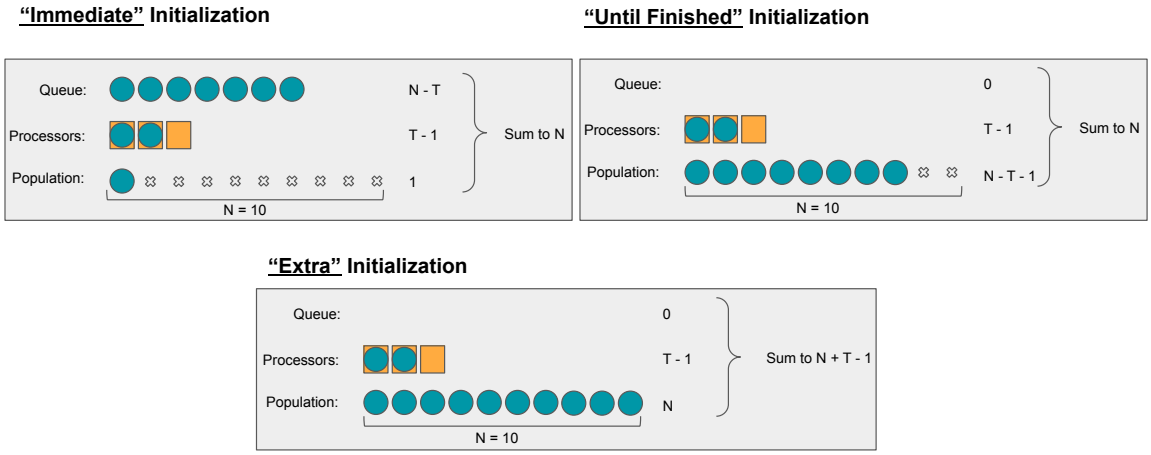

Figure 3.1: Illustration of the three asynchronous initialization strategies that I study here. Each diagram shows what the state of the algorithm looks like at the moment that initialization logic ends and regular, steady-state evolution begins (circles indicate the individuals generated during random initialization, squares indicate the three processors that are available in this example). In the *immediate* strategy (**top left**), evolution begins when exactly one individual has completed evaluating and enters the population. In the *until-finished* strategy (**top right**), evolution begins as soon as all initial individuals have either evaluated or are in the midst of evaluation, and one processor becomes free. With the *extra* strategy (**bottom**), evolution begins as soon as the population is full—and a few extra random individuals are generated in the meantime to keep all but one processor busy.

Hypotheses

There seem to be three equally basic interpretations of asynchronous initialization. The three possible strategies that I have seen colleagues gravitate toward are illustrated by diagrams in Figure 3.1. Each of these can be described in terms of how far evaluation of an initial set of generated individuals proceeds before the initialization procedure terminates and steady-state evolution begins:

- Coletti et al. [2019], for example, use an ***immediate*** strategy: they assume that individuals are queued in a parallel cluster when SEND() is called, and they enter the steady-state regime of Algorithm 2 immediately after the *first* initial individual completes (i.e., as soon as the population contains a single individual that can act as a parent). The use of a processing queue (implicitly added to by SEND() when

the processors are already busy) is a natural pattern for practitioners to fall into, since many distributed computing environments—such as the `dask` library [Rocklin, 2015] that the LEAP evolutionary computing framework relies upon [Coletti et al., 2020]—readily support arbitrary-length job queues for clusters of many processors.[3]

- At the opposite extreme, the ECJ software framework [Luke, 2017, Scott and Luke, 2019] uses an ***extra*** strategy for its asynchronous initialization: steady-state evolution does not begin until the population is completely full of evaluated individuals ($|P| = \mu$), and to keep all the processors busy until that time, a few extra individuals ($T - 1$, to be exact) are generated—for a total of $\mu + T - 1$ random initial individuals.

- In my most recent work with colleagues [Scott et al., 2021, in press], we have adopted a middle-ground ***until-finished*** strategy: exactly μ random individuals are generated, but steady-state evolution begins as soon as all individuals are either evaluated or *currently evaluating* on the processors. At this point in time, the population contains $\mu - T + 1$ individuals that can be used as parents.

All three of these strategies satisfy the assumption I made in Algorithm 2 that when the INITIALIZE() procedure returns, there is exactly one free processor available in the computing environment. But they differ significantly in how many individuals have been added to the initial population at the moment that evolution begins: just one (*immediate*), $\mu - T + 1$ (*until-finished*), or μ (*extra*).

Figure 3.1 emphasizes the case where $T << \mu$—i.e., the number of processors is considerably less than the population size. In this case, the behaviors of the three algorithms all differ, but we have $\mu \approx \mu - T + 1$, and thus the *until-finished* and *extra* strategies behave somewhat similarly to one another (with just a minor difference in how many individuals

[3] And while the other initialization strategies I study here do not rely on a queue of waiting jobs in the experiments that I present, this is partly an artifact of my assumption that the number of processors T is held constant. In large-scale, real-world distributed computing environments, asynchronous algorithms must be robust to the periodic failure of nodes and/or changes in the number of processing resources that are available in the midst of an algorithm run. Queuing effects are thus likely to affect many kinds of ASEA at scale—although I largely assume away this caveat throughout this chapter for simplicity.

they sample and the point at which evolution begins):[4]

Hypothesis 3.1. *When $T << \mu$, the performance of an ASEA using the* extra *strategy will behave considerably more similarly to the* until-finished *strategy than it will to the* immediate *strategy.*

In this scenario, furthermore, the *immediate* strategy behaves radically differently, in that the evaluation queue of length $\mu - T$ that it begins evolution with never shrinks in size: it continues to be added to and removed from as steady-state evolution continues. This introduces a significant delay into the evolutionary feedback loop, by artificially inflating the selection lag of all individuals by a factor of $\mu - T$, or approximately μ when $T << \mu$. Using the *until-finished* strategy as a reference point, I state this observation as the following hypothesis:

Hypothesis 3.2. *When $T << \mu$, an ASEA using the* immediate *strategy will take significantly longer to reach high-fitness solutions on a typical optimization problem than the* until-finished *strategy.*

Another instructive edge case occurs when the number of processors is equal to the population size ($T = \mu$). Under this assumption, the *immediate* and *until-finished* strategies are now exactly equivalent: both generate μ initial individuals and begin evolution with one individual in the population (and an empty queue of waiting jobs). In this case, the pathological queuing behavior of the *immediate* strategy is eliminated:

Hypothesis 3.3. *When $T = \mu$, an ASEA using the* immediate *strategy behaves no differently than one using the* until-finished *strategy.*

But in this $T = \mu$ case, the *extra* initialization strategy behaves very differently from the

[4]Throughout this dissertation, I will introduce a lot of numbered hypotheses as a means of framing and presenting results. But don't read too much sophistication into this organizational device. Like much of science [Bartz-Beielstein, 2006, Mayo, 1996], most of this work began less with a solid theory to test (*à la* Karl Popper) and more of a "huh, that's funny" (to paraphrase a popular Asimov quote). It's just easier to summarize a bunch of hypotheses than a bunch of "funnies."

other two: it creates fully $2\mu - 1$ individuals during initialization—resulting in almost double the initial exploratory sampling of the search space:

Hypothesis 3.4. *When $T = \mu$, an ASEA using the* extra *strategy behaves significantly differently from one using the* until-finished *strategy.*

The performance effect of this extra initial sampling will be problem-dependent. In the experiment below, however, I happen to choose a problem and algorithm parameters in which extra up-front exploration is considerably beneficial—so I will look for a performance boost in the data when the *extra* strategy is used.

Methods

For clarity, the three initialization strategies that I study are specified as pseudocode in Algorithms 4, 5, and 6. To test the differences between these algorithms, I ran an ASEA with a real-valued representation on an exponential fitness function:

$$f(\mathbf{x}) = \exp\left(\sum_{i}^{n} x_i\right). \tag{3.1}$$

This function is useful for testing ASEA behavior, because it introduces exaggerated fitness differentials between individuals within each generation—and when evaluation times are proportional to fitness, this can help to bring out dynamics that are affected by evaluation times. I simulate the evaluation times for these experiments to create problems that have either a positive correlation between evaluation time and fitness or a negative one.

To test the slate of hypotheses above, I prepared experimental configurations with different population sizes and different numbers of processors. The algorithm I apply here is a real-valued EA with additive Gaussian mutation applied to each gene. Mutation is applied with probability $p = 1/L$, and the steady-state components of the ASEA use random selection of competitors for survival selection (always keeping the best individual) and binary tournament selection for parent selection. No crossover is used.

Algorithm 4 *Immediate* Asynchronous Initialization

```
1: function immediateAsyncInitialize(μ, T)
2:     P ← ∅                                          ▷ Start with an empty population
3:     for i ← 1 to μ do                              ▷ Queue all individuals for evaluation
4:         Send(randomIndividual())
5:     P ← P ∪ {nextEvaluatedIndividual()}            ▷ Collect the first to complete
       return P
```

Algorithm 5 *Until-Finished* Asynchronous Initialization

```
1: function untilFinishedAsyncInitialize(μ, T)
2:     P ← ∅                                          ▷ Start with an empty population.
3:     for i ← 1 to μ do                              ▷ Queue all individuals for evaluation.
4:         Send(randomIndividual())
5:     while |P| < μ − T + 1 do                       ▷ Collect individuals until the queue is empty.
6:         P ← P ∪ {nextEvaluatedIndividual()}
       return P
```

Algorithm 6 *Extra* Asynchronous Initialization

```
1: function extraAsyncInitialize(μ, T)
2:     P ← ∅                                          ▷ Start with an empty population.
3:     for i ← 1 to T − 1 do                          ▷ Fill up all but one processor.
4:         Send(randomIndividual())
5:     while |P| < μ do                               ▷ Collect individuals until the population is full.
6:         Send(randomIndividual())
7:         P ← P ∪ {nextEvaluatedIndividual()}
       return P
```

Table 3.1: Median area-under-curve (AUC) results for each initialization strategy on exponential problems. Statistical tests here compare the *immediate* and *extra* strategies against the *until-finished* control. Asterisks indicate the smallest p-value that significance is achieved at with a Wilcoxon rank-sum test: $* = 0.05$, $** = 0.005$, and $*** = 0.0005$.

Environment	Pop. Size	# Procs	*Until-Finished*	*Immediate*	*Extra*
Exponential (correlated)	10	5	6.47e+11	**5.98e+11*** **	**6.60e+11***
Exponential (correlated)	50	5	2.81e+11	**2.86e+10*** **	**3.10e+11***
Exponential (anti-correlated)	10	5	8.02e+11	**7.63e+11*** **	7.97e+11
Exponential (anti-correlated)	50	5	4.79e+11	**1.95e+11*** **	5.03e+11
Exponential (correlated)	10	10	2.15e+12	2.15e+12	**2.18e+12*** **
Exponential (correlated)	50	50	1.36e+12	1.35e+12	**1.47e+12*** **

For each initialization strategy and experimental configuration, I measure the area under the best-so-far fitness curve (AUC) after running the algorithm for a fixed number of steps. In each sub-experiment, I ran each initialization strategy for 50 independent runs to collect mean best-so-far and median AUC results.

Results

The performance of all three initialization strategies for population sizes of 10 and 50 with the number of processors fixed to $T = 5$ is shown in Figure 3.2 (for the positively correlated problem) and Figure 3.3 (for the negatively correlated problem). These graphs test the case in which $T << \mu$. **As Hypothesis 3.1 predicts,** the *extra* and *until-finished* strategies perform very similarly both in terms of mean best-so-far trajectory and AUC values. And **as predicted by Hypothesis 3.2**, the *immediate* strategy takes far longer to converge than the other approaches as the population size grows.

These results are confirmed by statistical tests in Table 3.1, which reports the median AUC values for each experiment, along with Wilcoxon rank-sum tests that confirm significant differences in the reported medians. The Wilcoxon tests here are conducted by treating the *until-finished* strategy as a baseline, and comparing each of the other strategies against

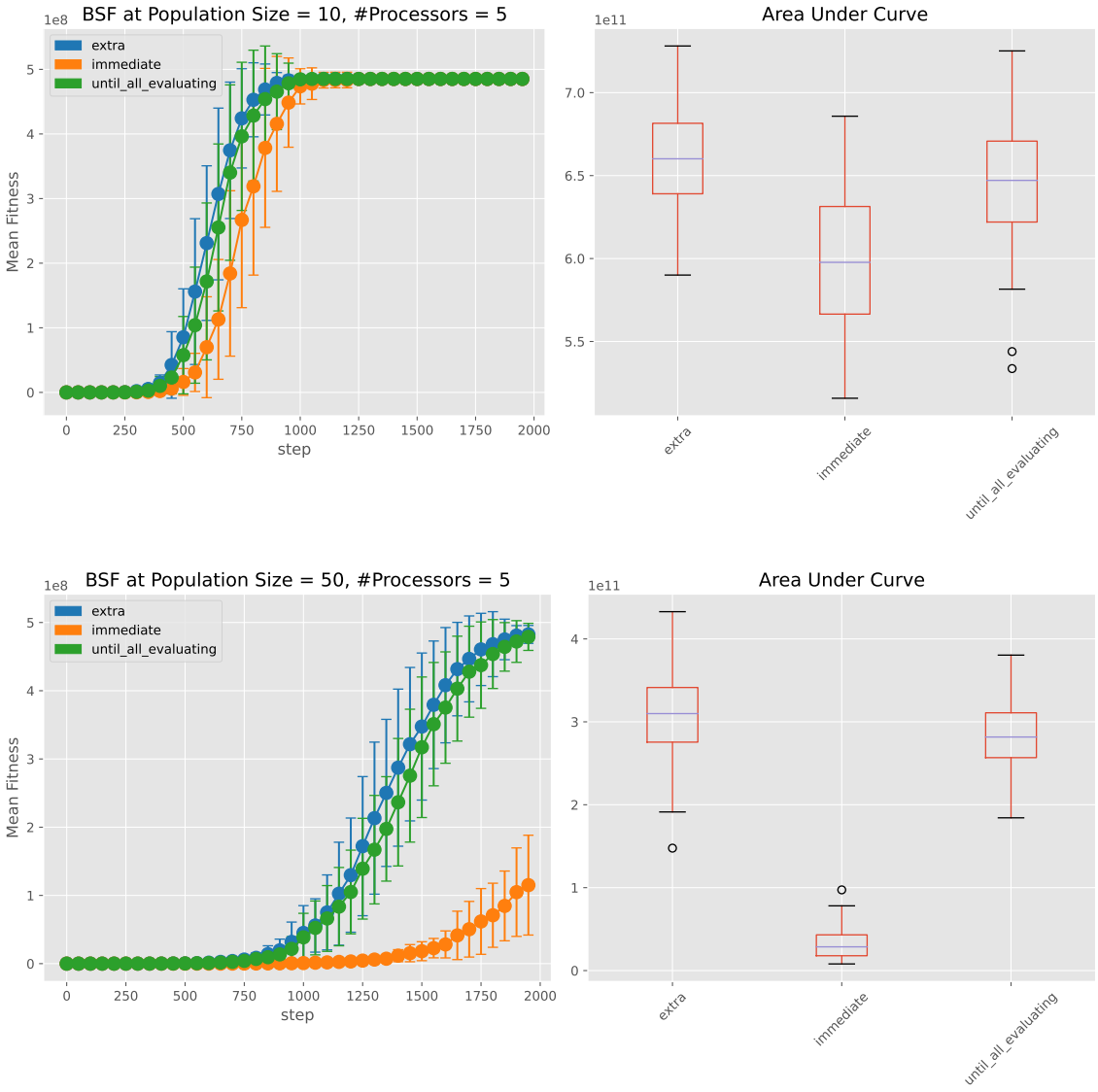

Figure 3.2: Mean best-so-far trajectories of an ASEA with **5 processors** using three different initialization strategies on the **correlated** exponential landscape (i.e., where evaluation time is proportional to fitness).

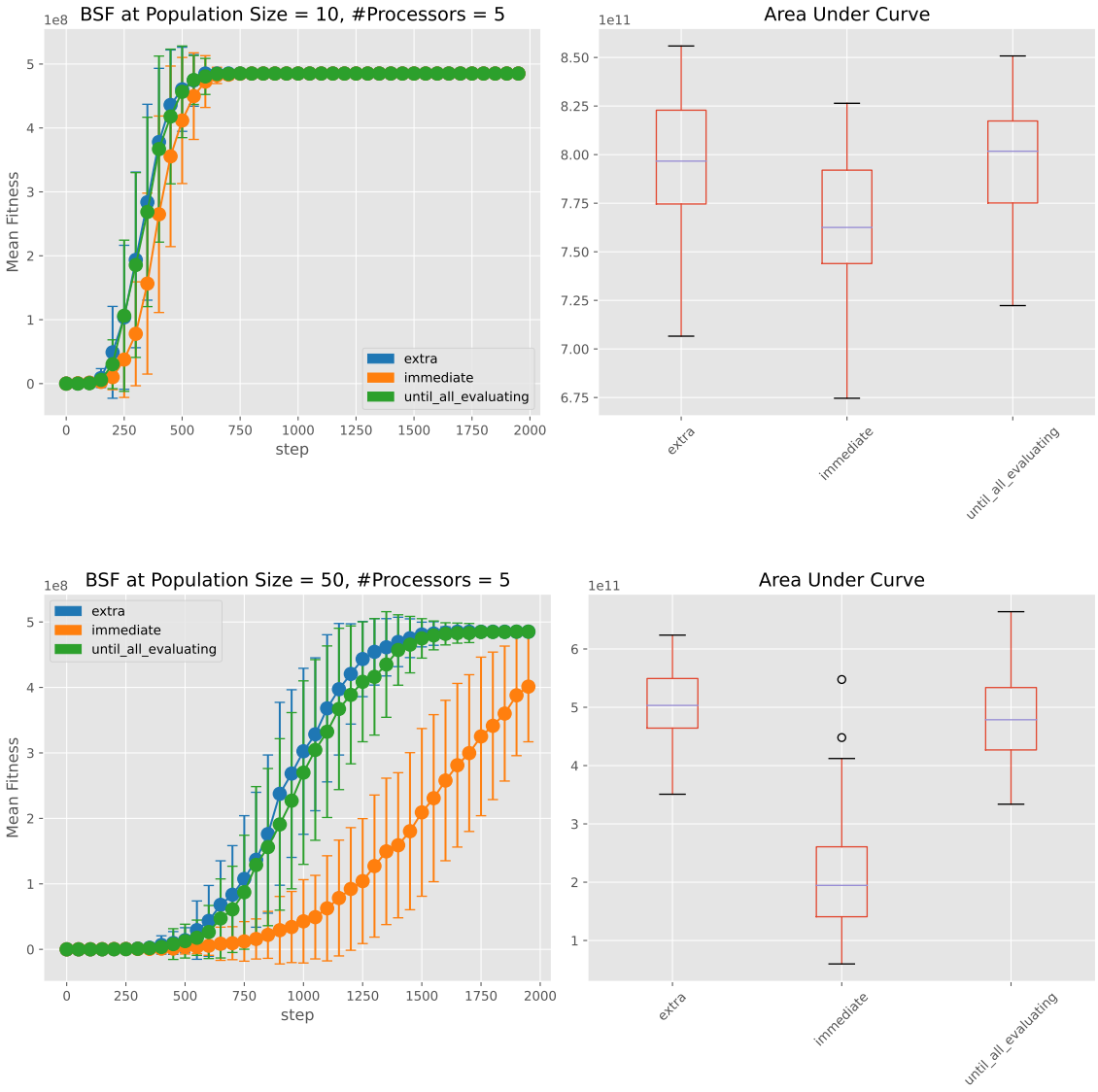

Figure 3.3: Mean best-so-far trajectories of an ASEA with **5 processors** using three different initialization strategies on the **anti-correlated** exponential landscape (i.e., where evaluation time is negatively proportional to fitness).

it.[5] Interestingly, the *extra* strategy shows a small (and weakly significant) performance improvement over the *until-finished* approach on the positively correlated problems (where better solutions take longer to evaluate), but not on the anti-correlated case (where better solutions evaluate more quickly).

My remaining two hypotheses treat the case where $\mu = T$. Figure 3.4 shows results of similar experiments conducted on positively correlated landscapes. In these experiments, there are no dramatic performance differentials for the *immediate* strategy, because no queueing effects occur when $\mu = T$. Specifically, I see no evidence of a performance difference between the *immediate* and *until-finished* strategies, **confirming Hypothesis 3.3**. I do observe a tendency for the *extra* strategy to converge more quickly toward higher-quality solutions, however—and this benefit is strongly statistically significant (again by Wilcoxon tests in Table 3.1). This result **confirms my fourth and final prediction, hypothesis 3.4.**

Conclusions on Initialization

In this section I have analyzed three common asynchronous initialization strategies for evolutionary algorithms that practitioners often implement. I have shown that the *immediate* strategy in particular—which uses a queue of individuals to maintain a backlog when the population size is greater than the number of processors—should be used with extreme care, as it can greatly lengthen the delay in the evolutionary feedback loop. The *until-finished* and *extra* strategies, meanwhile, perform fairly similarly under the conditions I have tested here.

In general, it may be preferable to use a hybrid scheme, in which a synchronous strategy is used for initialization before transitioning to steady-state evolution. I revisit this topic in section 3.6.1 as a potential avenue for future work.

[5] I did not bother with a Bonferroni correction, since I am only testing two simultaneous hypotheses at a time in each experiment. The results that confirm the hypotheses are significant at $p < 0.0005$, however, so they would easily satisfy Bonferroni.

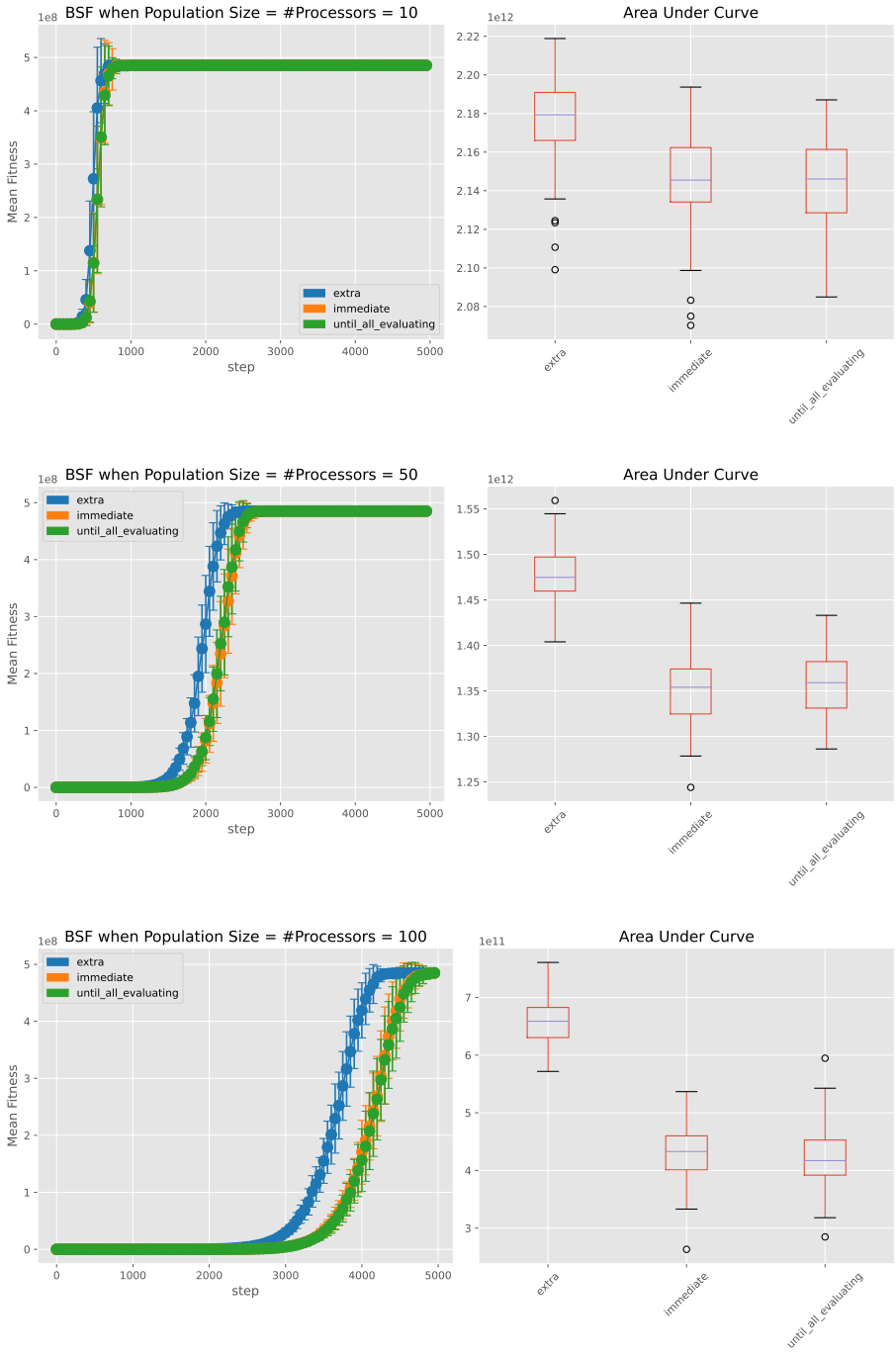

Figure 3.4: Mean best-so-far trajectories of an ASEA with **the number of processors equal to the population size** using three different initialization strategies on the **anti-correlated** exponential landscape (i.e., where evaluation time is negatively proportional to fitness). Shown are results for population sizes of $\mu = 10$ (top), 50 (middle), and 100 (bottom), respectively.

3.3 Parallel Speedup with Asynchronous Evaluation

Asynchronous EAs are interesting primarily because they promise to increase the total number of individuals that complete evaluation per unit of wall-clock time—i.e., *throughput*. I measure throughput by considering the time it takes for an EA to execute a fixed number of evaluations. I call the ratio between two algorithms' throughput the *throughput speedup*.

Extra fitness evaluations do not necessarily translate to an increase in progress toward a goal, however. Being able to evaluate more candidate solutions per unit time does not by itself guarantee that an algorithm can find a higher quality solution to a problem. It is important to distinguish throughput speedup from *true speedup*, which would take the quality of the resulting solution into account.[6]

In this section I characterize the speedup that an ASEA offers over a parallel generational algorithm. I present throughput speedup results for the ASEA, first analytically in section 3.3.1 and then empirically in section 3.3.2. After that I return to the more difficult question of true speedup with further experiments in section 3.3.3.

3.3.1 Analytical Lower Bounds on Throughput Speedup

As elsewhere in this chapter, I assume that the evaluation time of individuals dwarfs all other EA overhead, and thus that an asynchronous EA has near-zero idle time. Under this assumption, the speedup in throughput that the asynchronous EA offers is completely described by the amount of idle CPU resources it recovers. Algebraically, the throughput speedup is thus expressed as the ratio

$$S = \frac{T_{\text{generational}}}{T_{\text{asynchronous}}} = \frac{1}{1 - \hat{I}}, \tag{3.2}$$

where $\hat{I}$ is the fraction of CPU resources that the generational EA would have left idle.

In this section[7], I consider the case where individual evaluation times are independent

[6] Zăvoianu et al. [2013b] use the term 'structural improvement' for what I call 'throughput speedup' here.

[7] With the exception of some moderate clarifications such as Theorem 3.2, I have published all the results

and identically distributed according to some distribution. That is: I focus on the case in which evaluation time is independent of a solution's fitness and its genotype. Furthermore, I assume for simplicity that the ratio of CPUs available to the population size is 1, i.e., there is one CPU for each individual in the population.

Under these assumptions, we[8] proceed to analyze the expected value of S, deriving a reasonably tight lower bound for the speedup of the ASEA. Although many of the assumptions made here may not hold (or at least not hold exactly) in applications, these results nevertheless provide us with a quantitative handle on what actual values for the throughput speedup S might look like. We will see in section 3.3.2 that the bounds derived here are only slightly violated as we relax some of their assumptions.

Now, the most important value in determining the fraction $\hat{I}$ of CPU resources that are left idle in a given generation (and thus the asynchronous speedup S) is the evaluation time of the longest-running job (or individual) on any of the processors. This is clear from Figure 2.4 in the previous chapter: the size of the lightly shaded idle region in each processor is determined by the gap between the finishing time of the job on each processor and the finishing time of the longest-running job.

With this in mind, we begin our analysis of speedup under uniform evaluation times with the following lemma:

Lemma 3.1. *If $P = \{Y_1, Y_2, \ldots, Y_n\}$ be a set of random variables that are i.i.d. and follow a uniform distribution $\mathcal{U}(a, b)$. Then*

$$\mathbb{E}[\max[P]] = b - \frac{1}{n+1}(b-a). \tag{3.3}$$

Proof. The idea of the proof is to recognize that the probability of $\max[P] \leq x$ is equal to

from this section in Scott and De Jong [2015].

[8] Welcome, dear reader, to the first-person plural! In this section, as in much of mathematics literature, "we" refers to you and me, since the convention is to assume that the reader is in some sense "present" with me, working through the results [Morgan, 1996]. Odd as it may be, I rather like this tradition.

the probability that all of the samples $Y_i \in P$ are less than or equal to x. From this we can obtain an integral expression for $\mathbb{E}[\max[P]]$, which can be solved in closed form.

From the theory of order statistics, the probability that the maximum value in P obtains a given value is characterized by the cumulative distribution

$$p(\max[P] \leq x) = \prod_{i=1}^{n} p(Y_i \leq x). \tag{3.4}$$

Taking the derivative of both sides yields the p.d.f.

$$p(\max[P] = x) = \frac{\mathrm{d}}{\mathrm{d}x} \prod_{i=1}^{n} p(Y_i \leq x). \tag{3.5}$$

Now we can combine this equation with integration by parts to express the expected value of the maximum value as a function of the c.d.f. of the Y_i's:

$$\mathbb{E}\left[\max[P]\right] = \int_{-\infty}^{\infty} x \cdot p(\max[P] = x)\mathrm{d}x \tag{3.6}$$

$$= \int_{-\infty}^{\infty} x \frac{\mathrm{d}}{\mathrm{d}x} \prod_{i=1}^{n} p(Y_i \leq x)\mathrm{d}x \tag{3.7}$$

$$= \left[x \prod_{i=1}^{n} p(Y_i \leq x) - \int \prod_{i=1}^{n} p(Y_i \leq x)\mathrm{d}x \right|_{-\infty}^{\infty} \tag{3.8}$$

This expression cannot be solved in closed form for most distributions. The c.d.f. of a uniform distribution over the interval $[a, b]$, however, is given by

$$p(Y_i \leq x) = \int_a^x \frac{1}{b-a}\mathrm{d}y = \frac{x-a}{b-a}. \tag{3.9}$$

Substituting into Equation 3.8, we have

$$\mathbb{E}\left[\max[P]\right] = \left[x\left(\frac{x-a}{b-a}\right)^n - \int\left(\frac{x-a}{b-a}\right)^n \mathrm{d}x\right|_a^b \tag{3.10}$$

$$= b - \frac{1}{n+1}(b-a), \tag{3.11}$$

and we have proved the lemma. □

Now we proceed to prove the following general lower bound on expected idle time that applies to any evaluation-time distribution that is bounded from above by $b \in \mathbb{R}^+$. We will proceed to derive corollaries for the uniform distribution afterwards (as a special case).

Theorem 3.1. *Let the number of processors in a generational evolutionary algorithm be equal to the population size, and let the evaluation times of the individuals in each generation be drawn i.i.d. from a probability distribution. Furthermore, assume that there is some $b \in \mathbb{R}^+$ such that the distribution's p.d.f. $f(x) = 0$ for all $x > b$.*

*Then the **expected fraction of idle time** $\mathbb{E}[\hat{I}]$ incurred by the generational algorithm is bounded from below by*

$$\mathbb{E}[\hat{I}] \geq \frac{1}{b}\left(\mathbb{E}[\max[P]] - \mathbb{E}[Y]\right). \tag{3.12}$$

Proof. Let $P = \{Y_1, Y_2, \ldots, Y_n\}$ be a set of random variables drawn i.i.d. from the distribution, where $Y_i \in [0, b)$ represents the evaluation time of the ith individual in the population. Then, as I illustrated in Figure 2.4 in the previous chapter, the absolute idle time suffered by the generational EA is the total number of CPU-seconds processors spend waiting for the longest-evaluating individual to complete:

$$I = \sum_{i=1}^{n}(\max[P] - Y_i). \tag{3.13}$$

To express idle time as a normalized value between 0 and 1, $\hat{I}$, we divide I by the total number

of CPU-seconds available to the algorithm during the generation, which is $n \max[P]$.

$$\hat{I} = \frac{I}{n \max[P]}. \tag{3.14}$$

Computing the expected value of this ratio distribution is difficult for most distributions, in part because $\max[P]$ and I are not independent. The following inequality is easier to work with:

$$\mathbb{E}[\hat{I}] = \mathbb{E}\left[\frac{I}{n \max[P]}\right] \tag{3.15}$$

$$\geq \mathbb{E}\left[\frac{I}{nb}\right] \tag{3.16}$$

$$= \frac{1}{nb}\mathbb{E}\left[\sum_{i=1}^{n}(\max[P] - Y_i)\right] \tag{3.17}$$

$$= \frac{1}{b}\left(\mathbb{E}[\max[P]] - \mathbb{E}[Y]\right), \tag{3.18}$$

where the last step follows by the linearity of expectation. □

We now confine our attention to the case where the Y_i's are sampled from a uniform distribution over the interval $[a, b]$. We will use the general bound from Theorem 3.1 to derive a lower bound on the expected idle time specific to this scenario.

Corollary 3.1. *Let the number of processors n in a generational evolutionary algorithm be equal to the population size, and let the evaluation times of the individuals in each generation be drawn i.i.d. from a uniform distribution $\mathcal{U}(a, b)$.*

Then the ***expected fraction of idle time*** $\mathbb{E}[\hat{I}]$ *incurred by the generational algorithm is bounded from below by*

$$\mathbb{E}[\hat{I}] \geq \left(\frac{b-a}{b}\right)\left(\frac{1}{2} - \frac{1}{n+1}\right). \tag{3.19}$$

Proof. The result follows by applying Lemma 3.1 like so:

$$\mathbb{E}[\hat{I}] \geq \frac{1}{b}\left[b - \frac{1}{n+1}(b-a) - \left(a + \frac{b-a}{2}\right)\right] \tag{3.20}$$

$$= \left(\frac{b-a}{b}\right)\left(\frac{1}{2} - \frac{1}{n+1}\right). \tag{3.21}$$

□

Note that since $\max[P]$ quickly approaches b as $n \to \infty$, this lower bound on $\mathbb{E}[\hat{I}]$ will be tight for large n. This can be seen by referring to Equations 3.15 and 3.16.

At this point we have proved bounds on the expected normalized idle time $\mathbb{E}[\hat{I}]$ in a generational EA. In experiments, however, it is more straightforward to measure the speedup between two algorithms than to directly measure the idle processing resources. Our final theorem rearranges our results to give a lower bound on speedup:

Theorem 3.2. *Let the number of processors equal the population size and evaluation times be i.i.d. from* $\mathcal{U}(a, b)$.

Then the ***expected throughput speedup*** $\mathbb{E}[S]$ *that the asynchronous steady-state EA exhibits over a generational algorithm is bounded from below by*

$$\mathbb{E}[S] \geq \left(1 - \left(\frac{b-a}{b}\right)\left(\frac{1}{2} - \frac{1}{n+1}\right)\right)^{-1}. \tag{3.22}$$

Proof. The result follows by using Jensen's inequality to rearrange inequality 3.19 to match the definition of speedup (Equation 3.2).

For clarity, let the RHS of inequality 3.19 in Corollary 3.1 be denoted by C. Then we

have

$$\mathbb{E}[\hat{I}] \geq C \tag{3.23}$$

$$1 - \mathbb{E}[\hat{I}] \leq 1 - C \tag{3.24}$$

$$\frac{1}{1 - \mathbb{E}[\hat{I}]} \geq \frac{1}{1 - C} \tag{3.25}$$

$$\frac{1}{\mathbb{E}[1 - \hat{I}]} \geq \frac{1}{1 - C}, \tag{3.26}$$

where the inequality reverses direction a second time in step 3.25 because both sides are positive (i.e., $\mathbb{E}[\hat{I}]$ and C are both between $(0, 1)$).

The LHS of Equation 3.26 is almost in the form of speedup, but not quite. To complete the rearrangement, let $X = 1 - \hat{I}$. Because $\hat{I}$ is normalized to always be between 0 and 1, we also have $X \in (0, 1)$. As a result, the function

$$\phi(X) = \frac{1}{X} \tag{3.27}$$

is convex (since the reciprocal function is convex for positive arguments). Now we can apply Jensen's inequality,[9] which states that for any convex function ϕ,

$$\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)]. \tag{3.28}$$

Applying this to inequality 3.26, we arrive at

$$\mathbb{E}\left[\frac{1}{1 - \hat{I}}\right] \geq \frac{1}{\mathbb{E}[1 - \hat{I}]} \geq \frac{1}{1 - C}, \tag{3.29}$$

which is equivalent to corollary. □

[9]For a proof of Jensen's inequality, see for example p. 27 of Cover and Thomas [2006].

We can see from Equation 3.22 that the expected speedup is determined entirely by the population size n and the ratio of the standard deviation (which is related to $(b - a)$ by a constant factor) to its maximum value b. Moreover, the maximum attainable speedup is determined by the limit of the idle time as n grows, which we can express exactly (because, as we noted, Equation 3.19 is tight for large n):

$$\lim_{n\to\infty} \mathbb{E}[\hat{I}] = \frac{1}{2}\left(\frac{b-a}{b}\right). \tag{3.30}$$

This result indicates that the generational EA will never incur an expected idleness greater than 50% when evaluation times follow a uniform distribution, and consequently an asynchronous EA can never provide a throughput improvement of greater than 2. I note in passing, however, that for other distributions that permit more extreme values (such as the Gaussian or long-tail distributions), a speedup of much greater than 2 is possible.

3.3.2 Throughput Speedup Experiments

Theoretical bounds like Theorem 3.1 and its corollaries can be very informative, but it is important to contextualize analysis of this kind with empirical results. Experiments help to verify that analytical results are correctly derived, provide a view of how tight the bounds they provide are in practice, and (most importantly) how quickly the theoretical results become invalid when the assumptions that went into them are relaxed.[10]

Hypotheses

I performed[11] two sets of speedup experiments to explore these questions. The first examines how the expected speedup of the ASEA varies with the number of processors:

[10] I have published all of the results from this section in Scott and De Jong [2015].

[11] And here I bid you adieu, dear reader: returning to the first-person singular now that the math is behind us; and because this is a dissertation, and I am expected to emphasize my personal contributions (or so I am told!).

Hypothesis 3.5. *When the number of processors n equals the population size and evaluation times are i.i.d. and uniformly distributed, the* ***throughput speedup of an ASEA*** *over a generational EA* ***increases with*** *n in a fashion that closely follows the lower bound given by Theorem 3.2.*

The second experiment examines how speedup changes when I relax the assumption that n equals the population size. This assumption simplified our analytical approach to throughput above, but in practical applications the population size will often be significantly greater than the number of processors. When the population is larger than the number of processors, each processor in a generational algorithm will evaluate a subset of individuals in the population. The total evaluation time of the jobs executed by a given processor during a generation in this scenario, then, is a random variable given by the *sum* of the evaluation times of the individual jobs. In general (for any distribution), while the mean and variance of the sum of n independent variables grow linearly with n, the standard deviation of the sum grows with $\sqrt{n}$.[12] As a result, the total evaluation time per generation on each processor will be less prone to extreme values relative to the mean—meaning that cases where a processor takes much longer to finish its workload relative to the other processors will become rarer. For this reason, I expect the amount of idle time in the generational algorithm (and thus the speedup achieved with an ASEA) to decrease:

Hypothesis 3.6. *The* ***throughput speedup of an ASEA*** *over a generational EA* ***decreases*** *with the ratio of the population size to the number of processors.*

[12] This follows from Bienaymé's formula, which is a general result that applies to any sum distribution:

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{Var}(X_i) + \sum_{i \neq j}^{n} \mathrm{Cov}(X_i, X_j).$$

When the X_i are independent (as they are in the current context), the covariances reduce to zero, and the variance of the sum grows linearly with n.

Methods

To measure the impact of evaluation-time variance on the speedup of an asynchronous steady-state EA, I use a synthetic fitness evaluation scheme to test Hypotheses 3.5 and 3.6. The evaluation scheme is configured to wait a given amount of time (i.e., using a `sleep` system call) before returning a fitness value. Since I am only studying throughput speedup in this experiment, and since evaluation times are independent of fitness in this experiment, the fitness values themselves are irrelevant for our purposes and were configured arbitrarily.

For both of these experiments, I run generational and asynchronous algorithms on a single shared-memory machine using the ECJ evolutionary computation framework [Scott and Luke, 2019]. Since executing `sleep` system calls to the kernel to simulate fitness evaluation latency is not resource intensive, I was able to simulate arbitrary numbers of processors on a shared-memory machine that had just a few cores. Because the algorithm is still running on a real-world machine, however, effects such as process scheduling, algorithm overhead, etc. may affect the performance of the algorithms in ways that are not captured by the assumptions of our derivations in section 3.3.1.[13]

To approach Hypothesis 3.5, I keep the number of threads available to the algorithms fixed to the population size. Within this assumption, I studied two configurations of the evaluation-time distribution:

1. In the first configuration, individual evaluation times were sampled uniformly from the interval $[\frac{t}{4}, t]$, where t is a sufficiently long time (a few seconds) that the overhead of evolutionary operators is negligible compared to evaluation time.

2. In the second configuration, evaluation times were sampled from $[0, t]$—with the result that the ratio between any two evaluation times may be arbitrarily large.

For each of these configurations, and for each of the population sizes $|P| \in \{5, 10, 15, 20, 25, 30\}$, I ran both an ASEA and a generational EA each for 50 independent

[13]This is in contrast to other experiments I present in this chapter, in which a discrete-event simulation is used to perfectly simulate a zero-overhead evolutionary algorithm.

runs. Each run of both algorithms ran for $|P| \cdot \lfloor |P|/500 \rfloor$ fitness evaluations (i.e., for as close to 500 fitness evaluations as one can get without exceeding it, given that the generational algorithm evaluates exactly $|P|$ individuals at a time and can only execute an integral number of generations).

To approach Hypothesis 3.6, meanwhile, I performed additional runs with the $[0, t]$ evaluation-time configuration, but this time the number of processors was fixed to $n = 10$ while the population size varied ($|P| \in \{10, 20, 30, 40, 50\}$). Similarly to before, I performed 50 runs for each configuration, each consisting of $|P| \cdot \lfloor |P|/250 \rfloor$ fitness evaluations.

In both experiments, I measure the empirical speedup by taking the ratio

$$\bar{S} = \frac{T_{\text{generational}}}{T_{\text{asynchronous}}}, \tag{3.31}$$

where $T_{\text{generational}}$ is the wall-clock time measuring how long it took the generational algorithm to complete all fitness evaluations, and $T_{\text{asynchronous}}$ is the equivalent quantity for the ASEA.

Results

The results of both experiments are shown in Figure 3.5.

The left-hand figure tests Hypothesis 3.5, comparing the empirically observed speedup values to lower bounds (bold dark lines). These bounds were computed from Theorem 3.2 using the parameters of the uniform distributions for the two experimental configurations, $[\frac{t}{4}, t]$ and $[0, t]$. The thin error bars indicate the standard deviation in the speedup across the 50 runs. Tighter, wide error bars showing the 95% confident interval on the mean are barely visible in the figure, indicating that the estimate of the expected speedup is precise.

The results confirm that Theorem 3.2 provides a reasonably tight estimate of the throughput improvement. For large numbers of processors ($T = n > 20$), however, the prediction no longer serves as an accurate lower bound. I found that the degree to which the results

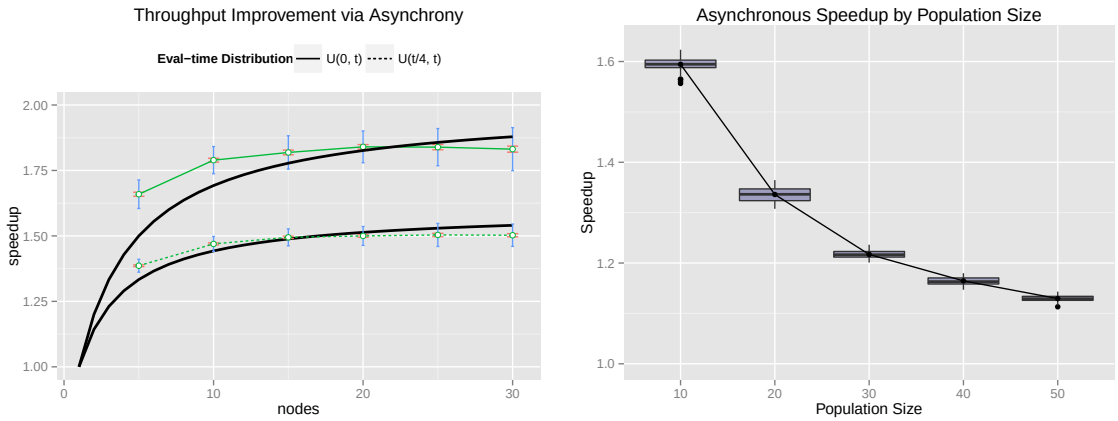

Figure 3.5: **Left**: Observed throughput improvement, shown along with theoretical lower bounds predicted by Theorem 3.2 (bold lines). Error bars show standard deviation (**blue** bars with narrow hats) and a 95% confidence interval on the mean (**red** bars with wide hats—these intervals are so small as to be barely visible). **Right**: Observed throughput improvement when the number of worker processors is fixed at 10 and the population size varies.

conform to the prediction vary somewhat depending on the architecture of the computer I run the experiments on. From this I surmise that the deviation from the prediction at high n is an artifact of my experimental setup, which simulates more processors than are actually available and thus invites some operating system overhead. I thus consider **Hypothesis 3.5 to be partly supported**: the theoretical bounds appear to be reasonably tight, but they begin to be violated as the overhead of executing many parallel evaluation threads grows. In particular, these results suggest that—while Equation 3.30 showed that the ASEA can attain a theoretical maximum speedup of 2 in the limit for uniform evaluation-time distributions—in practice speedups of greater than 1.8 or 1.9 may be difficult to achieve, due to classical considerations related to Amdahl's law [Amdahl, 1967].

The right-hand side of Figure 3.5 tests Hypothesis 3.6 by measuring the effect of holding the number of processors fixed at 10, and varying the population size while individual evaluation times are sampled non-heritably from $[0, t]$ (50 independent runs of 250 fitness evaluations each). As $|P|$ increases, each processor becomes responsible for evaluating a

larger share of the population, and the throughput improvement quickly decreases, as was predicted. This data **supports Hypothesis 3.6**.

In the experiments shown so far, the variance in throughput improvement from run to run is small. When evaluation time is a heritable trait, this may no longer be the case, as genetic drift and/or selection can significantly alter the distribution of evaluation times as evolution progresses. The amount of throughput improvement that the algorithm attains over the entire run depends heavily on how the magnitude and variation of evaluation times expands or shrinks over time. How that change occurs depends in turn on how the heritable component of evaluation time is related to an individual's fitness. I will return to this nuance as part of further experiments in section 3.3.3.

Conclusion of Throughput Experiments

In this section I have used empirical experiments to demonstrate and test the limits of the analytical model of ASEA processing throughput that we derived in section 3.3.1. These results serve to precisely illustrate and bound the main advantages that the ASEA offers to practitioners: asynchronous parallelization is especially advantageous over the generational EA when the number of worker processors is large and the ratio of processors to the population size is high. But decreasing returns appear to set in quickly, so it is important to have realistic expectations about how much of an advantage asynchrony will offer.

The main limitation of the results that I have presented in this section are that they deal only with evaluation times that are both uniformly distributed and non-heritable. It is reasonable to expect qualitatively similar results for other simple distributions when evaluation times are non-heritable. When evaluation times are systematic, however—in the sense of being a trait that carries over from parents to offspring—or when they exhibit some relationship to fitness (such as when faster-evaluating solutions tend to have better fitness), then the impact that asynchronous dynamics have on the performance and search trajectory of an EA may deviate in complex ways from what the results of this section would suggest.

I turn to these more challenging questions in the next section.

3.3.3 True Speedup Experiments

In the forgoing discussion, I have given an analytical and quantitative view of the speedup that an asynchronous EA offers in *throughput*. The throughput improvement an asynchronous EA offers over a generational alternative is an intuitively appealing metric, because one is inclined to believe that progress toward the solution can be measured by the number of fitness evaluations an algorithm has completed. But focusing on throughput alone can be misleading when considering optimization algorithm performance.

In his early analysis of asynchronous master-worker EAs, Kim [1994] called this the *requisite sample set hypothesis*, and used it to express the importance of throughput:

> Given an [algorithm] for which the invariance of the requisite sample set size holds, search speed is determined by the throughput of fitness evaluation. The greater the number of processors dedicated to the evaluation of individuals, and the higher the utilization of these processors, the faster the global optimum will be located.

When the number of samples an asynchronous EA needs to find a high quality solution on the given problem is equal to the number of samples a generational EA requires, then throughput improvement is an accurate predictor of true speedup. But this assumption is unlikely to hold in practice. In principle, on some problems the asynchronous EA could require so many more samples that it takes longer to converge than the generational EA, despite the increase in throughput. Computational throughput serves no evolutionary purpose unless it allows us to find better solutions than would otherwise be possible (given the same computational budget).

In this section, I continue to analyze the speedup the ASEA offers over a generational EA, but now I consider the *true speedup*—i.e., the speedup in terms of solving particular problems, rather than only the throughput speedup.[14] An ASEA's true speedup is determined partly by the throughput advantages it offers, and partly by the search trajectory it

[14] I have published all of the results from this section in Scott and De Jong [2015].

takes when sampling the solution space. Informally, the latter can be understood in terms of how the algorithm balances exploration with exploitation [Blum and Roli, 2003b]—or, more simply, by the degree to which its search trajectory exhibits *greediness*.

As with all evolutionary algorithms, the complex interactions between selection and variation operators in an ASEA ensure that the algorithm operates as a non-linear dynamical system [De Jong, 2006]. Because it is generally difficult to study these interactions analytically, in this section I present only empirical results.[15]

Hypotheses

Here I use a set of three qualitative properties to suggest hypotheses about how exploration and exploitation are exhibited in ASEAs: namely 1) the relative greediness of steady-state population models, 2) genetic drift, and 3) evaluation-time bias.

First, traditional steady-state evolutionary algorithms are known to have a more *greedy* search trajectory than generational algorithms. This arises because as new individuals replace existing individuals in the population, some individuals are replaced not long after they are born, and are thus given far more limited opportunities to produce offspring than they would be in a generational population model. The result is that a steady-state EA has a shorter delay in its evolutionary feedback loop than a generational EA does [De Jong, 2006, pp. 52–3]. While the ASEA is a generalization of the SSEA, all ASEAs share this basic property of a tightened feedback loop in common with the traditional SSEA. On highly multi-modal problems, this may cause the algorithm to be more prone to falling into a local optimum, delaying or preventing it from finding a good solution. But on simple, unimodal problems, greediness may be an advantage—reducing the number of fitness samples required to find a good solution:

[15] While there is a significant literature devoted to the computational complexity analysis of traditional EAs [Doerr and Neumann, 2020], sophisticated proof techniques are often needed for even very simple problems, and almost none of this work has examined the additional complications introduced by asynchronous EAs.

Hypothesis 3.7. *In general, the ASEA's search trajectory on simple problems (spheroid and Rastrigin) will be* ***greedier*** *than a generational EA, in terms of taking* ***fewer fitness evaluations*** *to approach the optimum.*

Second, when evaluation times are heritable but can vary independently of fitness, the degree of evaluation-time variance that the population exhibits at any given step is determined by competing pressures from mutation (which changes evaluation-time traits in individuals over time and can cause them to diverge) and selection (which, through genetic drift, has a tendency to cause diversity to be lost given sufficient time, in extreme cases fixating to a single uniform value) [De Jong, 2006, pp. 121–123]. If the former dominates over the latter (as is normally the case), then it will lead to higher variance in evaluation times within the population as the simulation proceeds.[16] For this reason, I hypothesize that the ASEA will exhibit higher variance in its performance in the heritable scenario:

Hypothesis 3.8. *When evaluation times are* ***heritable but independent of fitness****, the* ***variance*** *in the ASEA's throughput speedup and true speedups will both be considerably greater than in the other three scenarios.*

Third, it seems plausible that ASEAs may exhibit *evaluation-time bias*: because of the selection lag property that ASEAs exhibit, it is sometimes the case that while a single long-evaluating individual is evaluating on one processor, many fast-evaluating individuals can be born, evaluated by the remaining processors, and integrated into the population. This may lead fast-evaluating individuals to be preferred over time by evolution, separate from any consideration of fitness.

[16] This can be seen by considering the case of additive Gaussian mutation, in which case—disregarding selection—the evolution of a neutral trait in a single individual's lineage follows a Gaussian random walk. The overall change in a lineage's evaluation-time trait between generation 0 and generation n in this case can be modeled as the sum $\sum_{i=1}^{n} X_i$ of n i.i.d. random variables $X_i \sim \mathcal{N}(0, \sigma^2)$. But by Bienaymé's formula, the variance of this sum distribution grows linearly in n:

$$\sum_{i=1}^{n} X_i \sim \mathcal{N}(0, n\sigma^2)$$

So after several hundred generations, the variance in a population's evaluation times has the potential to grow very large.

In my experience, practitioners frequency cite the possibility of evaluation-time bias as a concern when they are considering applying an ASEA to a problem. In particular, the informal expectation is that when better solutions are faster, evaluation-time bias may accelerate an algorithm's trajectory toward these solutions—leading to a more exploitative algorithm:

Hypothesis 3.9. *When* ***better solutions are faster*** *(i.e., when evaluation times are positively correlated with fitness on a minimization problem) on simple problems (spheroid and Rastrigin), the ASEA will take* ***fewer fitness evaluations*** *to find the optimum than it does in the other three scenarios, and will exhibit a* ***greater true speedup*** *over a generational EA.*

Conversely, when better solutions are slower, the algorithm may be slower to move into good regions of the search space:

Hypothesis 3.10. *When* ***better solutions are slower*** *(i.e., when evaluation times are negatively correlated with fitness on a minimization problem), the ASEA will take* ***more fitness evaluations*** *to find the optimum than it does in the other three scenarios, and will exhibit* ***less true speedup*** *over a generational EA.*

The possibility of these dynamics is particularly concerning on difficult, multi-modal problems. Because of the greedier dynamics of steady-state population models, I expect the ASEA to be more prone to local optima than the generational EA overall:

Hypothesis 3.11. *In general, the ASEA will be* ***more prone to local optima*** *than a generational algorithm on highly multi-modal problems (namely the Hölder table).*

But when better solutions are faster-evaluating, it may be that evaluation-time bias exacerbates the problem of local optima further:

Hypothesis 3.12. *When **better solutions are faster** (i.e., when evaluation times are positively correlated with fitness on a minimization problem) on a highly multi-modal problem (Hölder table), the ASEA will be **more prone to local optima** than it is in the other three scenarios.*

A Remark on Metrics

As Kim's concept of a "requisite sample set" suggests [Kim, 1994], measuring the true speedup of an algorithm requires me to take the *number of fitness evaluations* into account that are needed to achieve a goal. This is in line with the overwhelming majority of evolutionary algorithms research, which considers the number of fitness evaluations needed to reach a goal (or, relatedly, the quality of the solution achieved after some fixed evaluation budget) as an objective way to evaluate algorithms' performance in a way that is comparable across different computer architectures (which may run the same algorithm at different speeds) [Luke and Panait, 2002].

When evaluation times vary, however, the raw count of fitness evaluations needed to achieve some goal (or, similarly, the quality achieved after some fixed evaluation budget) can be misleading. In asynchronous algorithm experiments, then, measuring performance in terms of time—time plotted on the x-axis, time to convergence, best solution found within a fixed budget of time, etc.—is often important. The trade-off is that the wall-clock time that passes while an algorithm runs varies considerably from machine to machine or implementation to implementation, making it difficult to compare results across experiments, papers, and labs.

In my experiments here, then, I focus primarily on wall-clock time metrics, since those are what give us the most meaningful window into the true speedup of an ASEA. I will additionally present some results in terms of evaluation-count metrics, to show how the search behavior of the ASEA differs from the control independently of time.

Methods

Following the four-part classification of problems based on their evaluation-time properties that I presented in section 2.1.2, I used four distinct simulated scenarios on each test function to see how the asynchronous EA performs on real-valued minimization problems. In all four scenarios, I represented individual genomes as vectors in $\mathbb{R}^l$, where l is the dimensionality of the particular problem at hand.

1. In the **Non-Heritable** scenario, individual evaluation times are uniformly sampled from the interval $[0,\ t_{\max}]$.

2. In the **Heritable**, fitness-independent scenario, I define a special gene to represent the individual's evaluation-time trait. The trait is randomly initialized on $[0,\ t_{\max}]$, and undergoes Gaussian mutation within these bounds with a standard deviation of $0.05 \cdot t_{\max}$. This gene is ignored during the calculation of fitness.

3. In the **Positive** fitness-correlated scenario, the evaluation time $t(\vec{x})$ of an individual $\vec{x}$ is a linear function of fitness with a positive slope m and zero intercept:

$$t(\vec{x}) = mf(\vec{x}) \tag{3.32}$$

   This simulates the case where evaluation becomes faster as a solution $\vec{x}$ approaches the optimum.

4. In the **Negative** fitness-correlated scenario, evaluation time is a linear function of fitness with a negative slope and a non-zero intercept:

$$t(\vec{x}) = \max\left(0, -mf(\vec{x}) + t_{\max}\right). \tag{3.33}$$

   In this case, evaluation becomes slower as the solution approaches the optimum (up to a maximum of $t_{\max}$ seconds).

I tested all four scenarios on the venerable 10-dimensional spheroid function, the 2-dimensional Rastrigin function, and the Hölder table function. The spheroid function[17] has a unimodal, quadratic macrostructure. The faster an algorithm approaches the optimum of a spheroid, the greedier its search behavior is. While the Rastrigin function has many local optima, it is linearly separable and has a quadratic macro-structure which makes the global optimum relatively easy to find:

$$f(\vec{x}) = 10l + \sum_{i=1}^{l} [x_i^2 - 10\cos(2\pi x_i)]. \tag{3.34}$$

The Hölder table function (Figure 3.6) is also highly multi-modal:

$$f(\vec{x}) = -\left|\sin(x_1)\cos(x_2)\exp\left(\left|1 - \frac{\sqrt{x_1^2 + x_2^2}}{\pi}\right|\right)\right| + 19.2085, \tag{3.35}$$

where I have added the non-traditional constant 19.2085 so that the global optima have a fitness of approximately zero. I select the Hölder table because it is moderately difficult, in the sense that the EAs I am studying sometimes converge on a local optimum, and fail to converge on a global optimum after several hundred generations.

Each algorithm used a Gaussian mutation operator with a per-gene mutation probability of 0.05, and used a 100% rate of two-point crossover. The population size was fixed at $n = 10$ in each case, and the number of worker processors was also $T = 10$. During both initialization and the application of reproductive operators, I bound each gene between -10 and 10 on all three objectives, and I set the standard deviation of the Gaussian mutation operator to 0.5. For the fitness-correlated scenarios, I set the parameter m to 1 for the sphere function and 5 on the Rastrigin and Hölder table. I ran each EA for 250 generations on the sphere function, 500 on the Hölder function, and 1,000 on the Rastrigin function.

[17] So-called not because it is a sphere, but because it is a paraboloid whose contours of equal fitness form hyper-spheres.

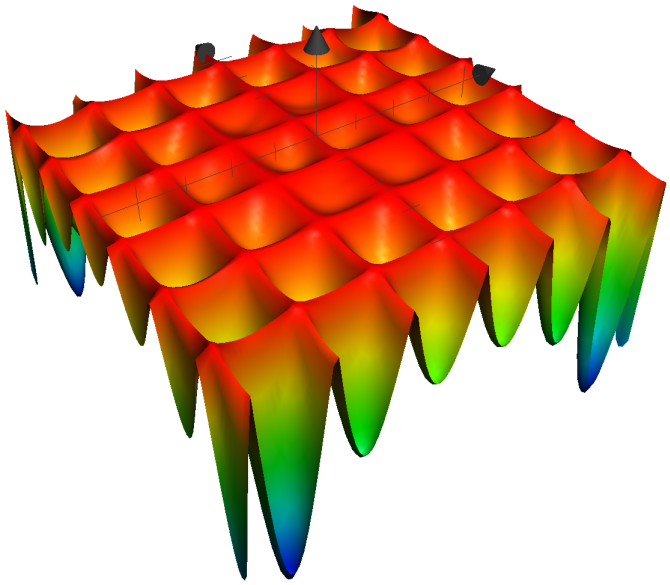

Figure 3.6: The Hölder table function.

Results on Sphere and Rastrigin Functions

Figures 3.7 and 3.8 show mean best-so-far fitness trajectories for 50 independent runs of the ASEA and generational EA on the spheroid and Rastrigin functions, respectively, for the case where evaluation times are non-heritable. The curves are indexed by generation on the left-hand side, and by wall-clock time on the right-hand side. For comparison, results from a plain (single-processor) steady-state EA are also shown in the by-generation plots.[18] For the non-generational algorithms (ASEA and SSEA), a "generation" here is taken to indicate a number of fitness evaluations equal to the population size.

The results on the spheroid and Rastrigin function indicate that the asynchronous EA is greedier than the generational EA. This is true in a fairly fundamental sense: because the evaluation-time distribution in this experiment is non-heritable, no systematic evaluation-time bias effect can be at work. This **supports Hypothesis 3.7**, indicating that the ASEA has greedier search behavior on a unimodal function (spheroid) and a function that

[18]Because the SSEA uses only a single processor, it takes far greater wall-clock time to solve these problems than the two multi-processor algorithms do. As such, I don't bother showing it in the wall-clock-time plots.

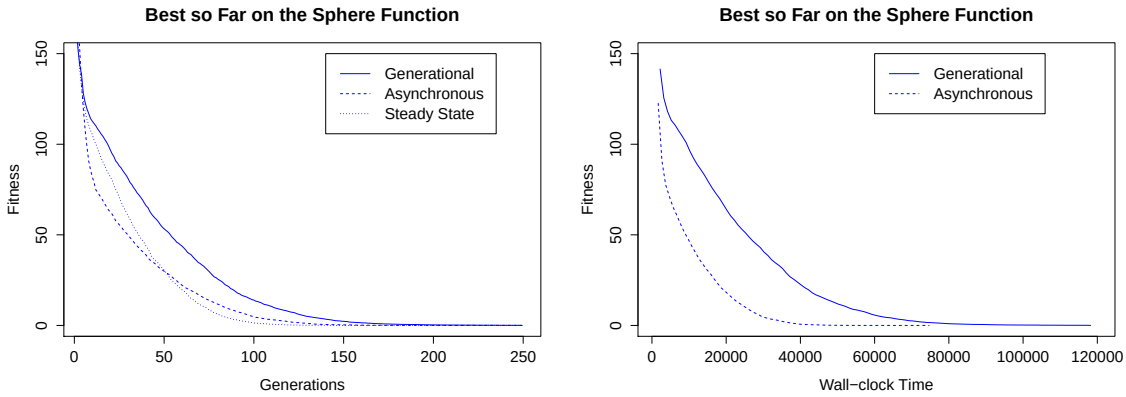

Figure 3.7: On the spheroid function, the simple asynchronous EA consistently takes fewer fitness evaluations on average to reach the optimum (**Left**). This contributes to its ability to find the solution in less wall-clock time than the generational EA (**Right**).

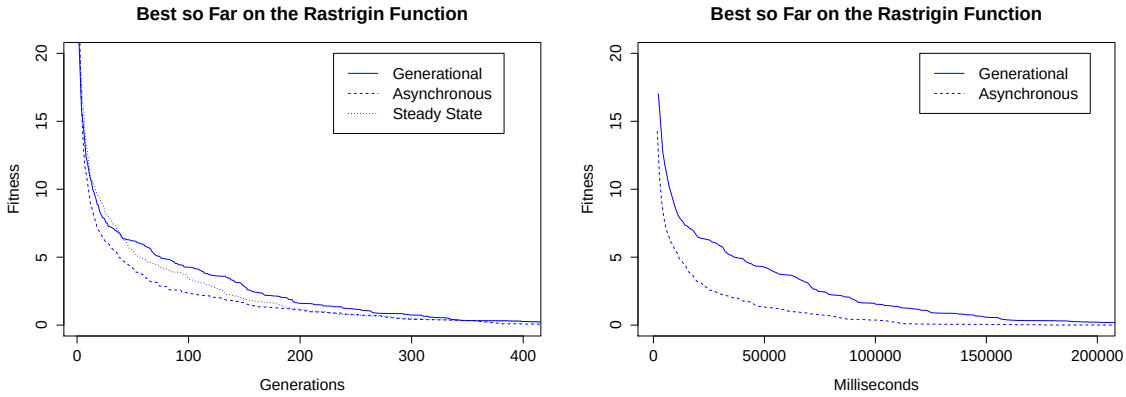

Figure 3.8: The asynchronous EA also performs well on the Rastrigin function both in terms of fitness evaluation (**Left**) and wall-clock time (**Right**).

is multimodal on a small scale, but has an overall convex macrostructure (Rastrigin).

Comparing just the ASEA performance to itself across the four scenarios, however, I observe no statistically significant ($p < 0.05$) difference in the number of fitness evaluations it takes for the asynchronous EA to converge on the spheroid function (not shown). This indicates that as Gaussian mutation leads the population to exploit a unimodal function, the effect of evaluation-time bias is negligible on average.

This surprising absence of an identifiable evaluation-time bias in the ASEA persists when I closely examine both the throughput speedup and true speedup in each scenario. Figures 3.9 and 3.10 show the true speedup and throughput improvement for each of the four scenarios on the spheroid and Rastrigin functions, respectively. The true speedup is measured by considering the amount of wall-clock time it takes each algorithm to reach a fitness value of less than the threshold value $\eta = 2$. Each scenario was tested by pairing 50 independent runs of each algorithm and measuring the resulting speedup distribution.

On both functions, I note that the "heritable" (independent of fitness) eval-time scenario exhibits higher variance than the other three scenarios. This **supports Hypothesis 3.8**, suggesting that in the absence of selection pressure on evaluation-time traits, genetic drift increases the unpredictability of the benefits of ASEAs.

Because the asynchronous EA tends to converge in less time than the generational EA in all four scenarios on both functions, and since all of them exhibit throughput improvements, their true speedups are greater than 1.0. On the Rastrigin, but not the sphere, the true speedups are remarkably high. All of them average over 2.0, which is the maximum-attainable expected value for throughput improvement for the non-heritable scenario as determined by Equation 3.30, and outliers for the non-heritable scenario are visible in Figure 3.10 with true speedup values in excess of 20 or 30. This is possible only because the asynchronous EA requires fewer fitness evaluations to solve the Rastrigin function than the generational EA.

On the spheroid function, most of the true speedup is more readily explained by an increase in throughput: I find at $p < 0.05$ (with a Bonferroni correction for testing four

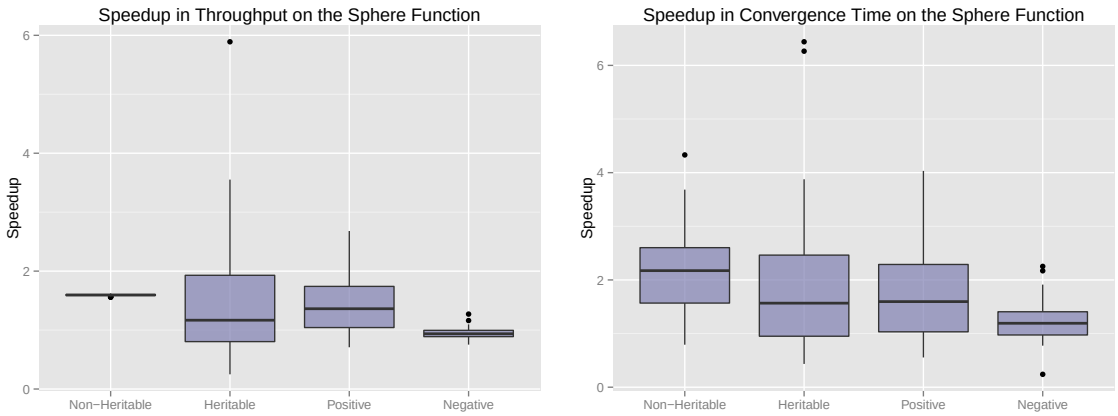

Figure 3.9: Speedup in throughput (**Left**) and convergence time (**Right**) of the asynchronous EA on the sphere function.

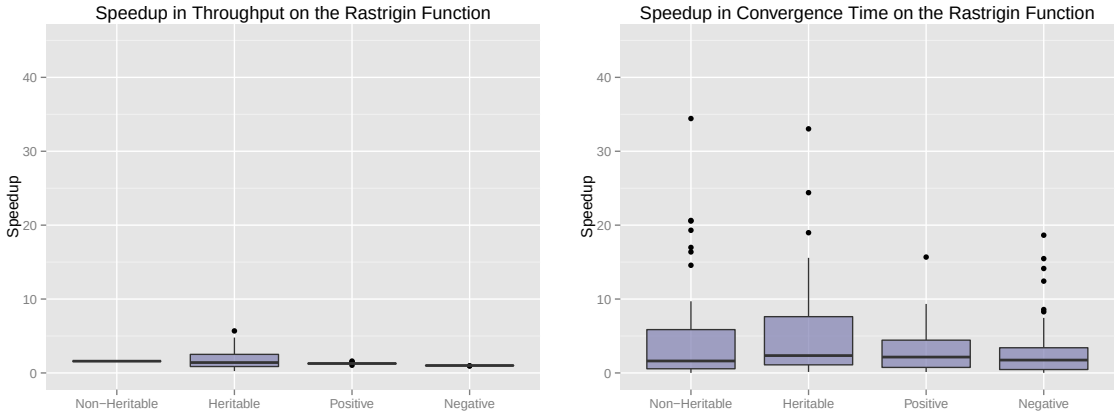

Figure 3.10: Speedup in throughput (**Left**) and convergence time (**Right**) of the asynchronous EA on the Rastrigin function.

simultaneous hypotheses) that there is no statistically significant difference between the means of the throughput speedup and true speedup in three of the four cases: the non-heritable, heritable, and positive scenarios. In these scenarios, all of the true speedup comes from throughput—there is no evidence that greater or fewer overall fitness evaluations are needed to reach the optimum. This result is **inconsistent with Hypothesis 3.9**, which predicted that in the positive ("better is faster") scenario, evaluation-time bias would cause the ASEA to require fewer fitness evaluations to reach the optimum than in the other scenarios.

In the negative ("better is slower") scenario, however, the same test (at $p < 0.05$ with Bonferroni correction) on the spheroid function rejects the hypothesis that the mean true speedup in the negative scenario is equal to the mean of its throughput improvement: the true speedup is *higher* on average. This indicates that the ASEA *did* require fewer fitness evaluations than a generational EA to solve the spheroid function in the negatively correlated scenario—despite the fact that this requires it to move more greedily into slow-evaluating regions of the space.

Specifically, on both the spheroid and the Rastrigin, the throughput speedup in the negative ("better is slower") scenario is very close to 1. It is easy to see why this is the case: when evaluation time is negatively correlated with fitness on a minimization problem, the majority of the processing time is spent in the later stages of the run, where the population consists primarily of high-quality individuals with long evaluation times. The late stages of the run are also where genetic variation is low in the population, so there is very little evaluation-time variance. As we saw analytically in Section 3.3.1, low evaluation-time variance translates to very little difference in throughput between the two algorithms—when there is no eval-time variance, there is no idle time to eliminate. As such, throughput improvement will approach 1 on any objective function provided that the algorithm is run for a sufficiently long period of time. And this is what I observe in the throughput results in Figures 3.9 and 3.10.

But the true speedup in the negative scenario is observed at an average of 1.23 +/-

0.05 (95% confidence interval) for the sphere function and 3.09 +/- 0.58 for the Rastrigin. The fact that the true speedup is greater than 1 for both functions indicates that it is the evolutionary trajectory of the asynchronous EA, not the elimination of idle time, that is producing the true speedup. These results **contradict Hypothesis 3.10**, which predicted that evaluation-time bias would cause the ASEA to require more fitness evaluations to reach the optimum.

Results on the Hölder Table Function

Both algorithms frequently fail to converge to a global minimum on the Hölder table function. As a result, it is not possible to fully describe the run-length distribution, which I used to compute the distribution of speedups on the other objectives. Instead, I characterize just part of the run-length distribution by measuring the *success ratio* of each algorithm after a fixed amount of resources has been expended (such as wall-clock time). I define the success ratio as the fraction of 50 independent runs that achieved a fitness of less than $\eta = 2$. See Bartz-Beielstein [2006] for a discussion of run-length distributions, success ratios and related measures.

In all four scenarios, the asynchronous EA is able to find a global optimum on the Hölder table function more frequently than the generational EA does. When both algorithms do find a solution, the asynchronous EA tends to find one at least as quickly as the generational EA does, as shown in Figure 3.11. The ASEA is thus less prone to local optima on this function than the generational EA. This surprising result **contradicts Hypothesis 3.11**.

In the positively correlated ("better is faster") scenario, however, the success rate of the ASEA is lower than in the other three scenarios. This **supports Hypothesis 3.12**, suggesting that under this evaluation-time scenario, the algorithm has a harder time escaping local optima relative to the other scenarios. The surprising caveat, however, is that the ASEA still outperforms the generational control in this scenario.

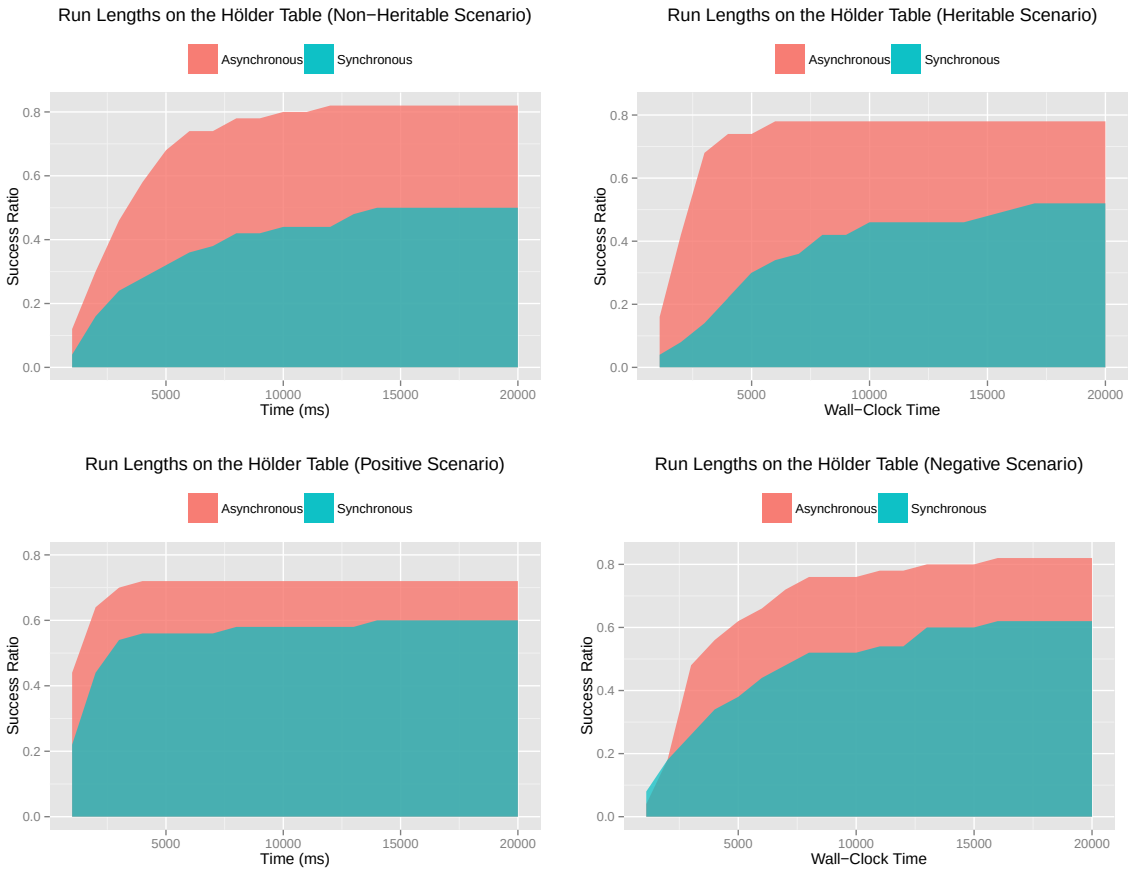

Figure 3.11: The asynchronous EA is able to find a global optimum on the Hölder table function more often than the generational EA in all four scenarios.

Conclusion of True Speedup Experiments

In this section I have shown that, like the single-processor SSEA that it generalizes, the ASEA tends to exhibit a greedier search trajectory on unimodal problems—but that this does not prevent it from doing a better job at solving a difficult, multi-modal problem (the Hölder table function) than a similarly configured generational EA. I have also shown that the variance in the true speedup it offers over a generational EA is considerably higher when evaluation time is systematic (heritable) but independent of fitness.

The question of evaluation-time bias, meanwhile, has proved to be considerably more subtle than I (or most practitioners I have spoken with) had believed before this study. Despite the fact that several experiments in this section were configured to have a perfect (100%) correlation between fitness and evaluation time, no clear, systematic effect emerged that would suggest ASEAs are clearly greedier when better solutions evaluate faster, or that they are less greedy when the opposite is true. Only one experiment suggests that the ASEA is a little more prone to getting stuck in a local optimum in the former case, and even then the ASEA outperforms a generational algorithm in this regard. The puzzle of evaluation-time bias is worth further investigation, and I will return to it with further experiments below in section 3.4.

Another salient observation that comes out of these experiments is that the benefits of asynchronous evolution are muted in cases where better solutions take longer to evaluate. There may exist simple modifications to the ASEA that can improve on this, making it more useful on problems of this kind. I will take up this proposal with additional experiments in section 3.5.

The results of this section are, of course, problem-dependent, and performance may vary on other objective functions. Some limitations of these results are that only three test functions were used, only linear relationships between fitness and evaluation times were considered, and most experiments assumed that the number of processors was equal to the

population size. I also did not perform any parameter tuning—I held each algorithm's parameters (such as population size and mutation width) constant, varying only the underlying algorithm. While this *ceteris paribus* approach is reasonable in these experiments, where the two algorithms share the same parameters and the different evolutionary models used by each algorithm are what I was primarily interested in studying, it is nonetheless possible that parameter choices and performance interact in complex ways (i.e., I did not test for the possibility that the generational EA performs best with a different mutation setting or population size than the ASEA).

Overall, on all three test functions and all four evaluation-time scenarios in the framework I have introduced, I have found that the asynchronous EA does not exhibit any particularly adverse performance that should worry practitioners. To the contrary, it does a good job of delivering the promised speedup in problem-solving ability, without introducing any overwhelming tendency to be biased by evaluation times or to fall into local optima.

3.4 Evaluation-Time Bias and Quasi-Generational Evolution

In section 3.3 I found that the evaluation-time properties of problems have less of an impact on the ASEA's search trajectory than one might expect. In particular, when better solutions are faster-evaluating, I found insufficient evidence to conclude that the ASEA moves more greedily toward better solutions—and conversely, I found insufficient evidence that it is less greedy when better solutions are slow-evaluating. This suggests that, while evaluation-time bias may still occur under certain conditions, it may arise in a complex way from the dynamical behavior of the ASEA, making it non-trivial to predict.

Evaluation-time bias is a serious concern among practitioners, who would like to be assured that an optimization algorithm is solving the objective they have defined for it. If an ASEA effectively solves an implicit objective—one that encourages it to minimize the evaluation time of candidate solutions, in addition to (or instead of) fitness—then it could lead to suboptimal or invalid results in some applications.

In this section, I perform a suite of experiments with both the ASEA and the quasi-generational EA (QGEA—which I introduced at a high level in section 2.1.3) that aim to understand evaluation-time bias, how and when it occurs, and how severe it tends to be. I begin this investigation in the simple setting of flat fitness landscapes (i.e., constant objective functions). In this scenario, selection based on fitness exhibits no preference for any solution over any other, and any empirically observed selection effect that remains can be attributed to evaluation-time bias in the algorithm. Second, I investigate simple multimodal landscapes, where I study how evaluation-time bias and fitness-based selection interact to alter the frequency with which each algorithm successfully finds the global optimum.

The experiments in this section are conducted on simple discrete- and real-valued optimization problems. These problems are not intended to form a representative benchmark with real-world features. The purpose of this kind of study is rather to gain targeted insight into aspects of the algorithms' behavior. In particular, I focus on how the evaluation-time properties of landscapes impact the behavior of each algorithm. Each synthetic problem is defined by a tuple of two functions (f, t), where $f(\vec{x})$ represents the fitness of individual $\vec{x}$ and $t(\vec{x})$ defines its evaluation time (in arbitrary units). Some of the experiments in this section use real evaluation times (with `sleep` calls to the kernel used to implement $t(\vec{x})$), and some use a discrete-event simulation to simulation the passage of time.

3.4.1 The Quasi-Generational EA

The QGEA uses an asynchronous evaluation scheme to minimize idle time, but as new individuals are evaluated, they are not incorporated directly into the parent population P in steady-state fashion [Durillo et al., 2008, Fonseca and Fleming, 1998]. Instead, they are added one-at-a-time to a separate child population P'. Once the child population becomes full, it replaces the parent population, and a new, empty child population is initialized. The key difference between the QGEA and the classical generational scheme is that the QGEA generates more than $|P|$ children from each set of parents, as a means of keeping all of the processors occupied (in this way, the QGEA's evolutionary loop has a lot in common

with the *extra* initialization strategy of Algorithm 6). Fonseca and Fleming argue that this approach retains dynamics that are similar to the generational EA, and that by doing so it will reduce the "bias towards individuals easy to evaluate." They also observe that it is possible to select all of the parents up front, rather than one-at-a-time—and that it is thus possible to use selection techniques that select a full population of parents all at once—such as stochastic universal sampling—in conjunction with the QGEA [Baker, 1987].

For simplicity, I interpret the QGEA as a variant of the general asynchronous EA given in Algorithm 2 in section 3.2. To implement generational dynamics, I define an alternative implementation of the INTEGRATE() procedure which inserts individuals into an offspring population until it is full—see Algorithm 7. Recall that Algorithm 2 calls INTEGRATE() once each time an individual completes evaluating, so Algorithm 7 incrementally collects individuals into a new population P', and uses it to replace the old population P only once P' becomes full ($|P'| = |P|$). Here I treat the offspring population P' as a global variable that is implicitly initialized to be the empty multiset.

Algorithm 7 Quasi-generational insertion into a population

```
1: function INTEGRATEGENERATIONAL(ind, P)
2:     P' ← P' ∪ {ind}
3:     if |P'| = |P| then
4:         P ← P'
5:         P' ← ∅
```

This integration method ensures that changes to the population occur in a generational way, in the sense that the population is completely replaced every μ steps.

3.4.2 Evaluation-Time Bias on Flat Landscapes

Evaluation-time bias can be understood as an implicit selection effect: at some point during a run, fast-evaluating individuals gain a reproductive advantage over slow ones—that is,

they are selected as parents more frequently than can be explained by other factors (such as fitness). The simplest and most direct way to study evaluation-time bias, then, is to ask what happens when the fitness of all individuals is equal.[19] This makes it possible to isolate any bias toward fast-evaluating genotypes from other evolutionary effects. In this setting, evolutionary change occurs under the influence of the well-studied problem of genetic drift in finite populations (see, for example, De Jong [2006]). Alleles are lost simply due to sampling variance occurring in the selection procedure.

Hypotheses

Naïvely, one might expect that the fast-evaluating subset of the population in an asynchronous EA is akin to the famously exponential growth of rabbits in 19th-century Australia: the faster individuals mature, the more children they will have, which will tend to be fast-evaluating themselves, and so forth—quickly coming to dominate the population. However, besides the fact that the total population size is fixed, there are at least two reasons that any reproductive advantage that fast individuals receive in an asynchronous EA is limited in a way that it is not for an exponential growth process like the Australian rabbits:

1. While a slow individual is evaluating, the CPU resource it is occupying is made unavailable. So, if half the processors are occupied with slow individuals, only half remain for fast-evaluating individuals to exploit.

2. Individuals do not reproduce as soon as they finish evaluating. Evaluations trigger a breeding event that produces a new child, but the parents are selected from the general population. Individuals only reproduce when they are selected.

So the dynamics that govern reproductive advantages for fast-evaluating individuals in an ASEA are somewhat subtle.

[19] With the exception of some minor refinements to the introductory arguments such as Equation 3.36, I have published all of the results from this section in Scott and De Jong [2015] and Scott and De Jong [2016a].

In fact, on closer analysis it seems that the evolutionary bias toward fast-evaluating phenotypes in the ASEA must be very small on average. Figure 3.12 illustrates an example in which there are three processors that receive a stream of individual jobs to process—and where `slow` jobs (S) take much longer to evaluate than `fast` ones (F). It is clear from this diagrammatic view that on the left half of the figure, any `fast` jobs that occur at the beginning of the subsequence assigned to each processor will complete evaluating long before any of the `slow` jobs have completed. The first few jobs to complete, then, will indeed be fast-evaluating, giving these phenotypes a "head start" at competing for a place in the population. The probability that a contiguous run of n `fast` jobs occurs at the beginning of each processor's queue of jobs, however, *decreases exponentially* with n. In fact, on average the length of any run S of contiguous `fast` jobs is equal to the *odds* of a single job being `fast`:[20]

$$\mathbb{E}[|S|] = \frac{p}{1-p}. \tag{3.36}$$

In the case where `fast` and `slow` values are equally probable, $p = \frac{1}{2}$, and we therefore expect $\mathbb{E}[|S|] = 1$. So in this example, only *one* `fast` individual will evaluate on average on each processor before the next `slow` individual arrives! In general, the initial burst of `fast`-evaluating individuals is limited to $\mathbb{E}[|S|]$ times the number of processors: or in this illustration, *just 3 individuals*.

What this implies is that—while `fast` jobs may have an initial transient advantage—it is not long before each processor is occupied by a `slow` job, at which point the system must wait until one of these complete before continuing. By this informal reasoning, I expect

[20] Specifically, if the probability that a given individual is of `fast`-type is p, then the chance that a sequence S of `fast`-type events has length at least n is the conjunction of n independent Bernoulli events, giving us the following exponential function:

$$P(|S| \geq n) = p^n.$$

Because $|S|$ takes values on $\mathbb{N} \cup \{0\}$, we can use the tail-sum formula from probability theory to compute the expectation for an infinite sequence of jobs:

$$\mathbb{E}[|S|] = \sum_{n=1}^{\infty} P(|S| \geq n) = \sum_{n=1}^{\infty} p^n = \frac{p}{1-p},$$

where the last step is the closed-form solution of the infinite geometric series (which holds for any $p < 1$).

Figure 3.12: An illustration of how a stream of jobs is processed by three processors, when each job has an equal probability of being either `fast`-type (F) or `slow`-type (S).

that evaluation-time bias will tend to be a short-lived, transient phenomenon in general (including in cases where evaluation times can vary continuously):

Hypothesis 3.13. *On flat fitness landscapes, the ASEA and QGEA will both exhibit a short-lived bias toward fast-evaluating individuals near the beginning of the run.*

Outside of this initialization effect, over long time scales each processor should complete an equal number of jobs in equal time on average. In the example of the two-valued evaluation-time distribution of Figure 3.12, the proportion of `fast` and `slow` jobs that complete evaluating on each processor is thus governed by the probability of each type of job being born (i.e., p and $1 - p$). Generalizing from this, I hypothesize that ASEAs will show no systematic bias toward fast-evaluating individuals over long time scales:

Hypothesis 3.14. *On flat fitness landscapes, the ASEA and QGEA will both exhibit no significant evaluation-time bias (that is, drift toward faster-evaluating genotypes) over time* after initialization.

Methods

I test these hypotheses about evaluation-time bias on flat landscapes in two settings: a simple problem where genotypes can take one of two discrete values, and on real-valued

search spaces. In the first domain I consider the case of no mutation (so, population change occurs only through genetic drift), while in the second I do include a mutation operator.

For the first domain, I study only the ASEA, and I define a heritable runtime trait by a gene that can take on one of two alleles: `fast` and `slow`, where `slow` takes 10 times longer to evaluate. I further assume that there is no reproductive variation (offspring are generated only by cloning), and that the fitness landscape is flat—i.e., every individual has an equal chance of being selected as a parent. Each cloned offspring replaces a random individual in the population with 100% probability. I seeded the initial population with random genotypes that were drawn with equal probability from both `fast` and `slow` alleles, so that the initial `fast`-type frequency averaged $f = 0.5$. I then ran the asynchronous EA with 10 worker processors and a population size of 100 for a total of 50 independent runs. In this scenario, any systematic change in genotype frequency can only be due to some implicit selection acting on the evaluation-time trait. In this experiment I measure the mean frequency of `slow`-type individuals in the population at each evolutionary step across 50 independent runs.

For this first set of experiments, I use the same implementation approach as in section 3.3: namely using an ASEA implemented in ECJ [Scott and Luke, 2019], with `sleep` system calls used to simulate artificially configured evaluation times.

In the second domain, I study both the ASEA and the QGEA, representing solutions as vectors in $\mathbb{R}$, and breed single children by applying Gaussian mutation with hard bounds and a standard deviation of 0.5, mutating each gene individually with 100% probability. I do not use crossover, and I consider only the case where the number of worker processors T is equal to the population size. I conduct experiment with population sizes of 10 and 100, respectively. Initial individuals are generated randomly from a uniform distribution across the genotype space, $\mathcal{U}(-10, 10)$, and they are evaluated on a task where the fitness function is flat ($f(x) = 1$). I investigate two different configurations of the evaluation-time function: in one scheme, evaluation times are defined by a 1-dimensional parabola centered on the origin ($t(x) = x^2$)—so individuals close to 0 are very fast-evaluating. In the other,

evaluation times are defined by a mixture of two Gaussian functions, one of which defines a fast-evaluating region and one that defines a slow-evaluating region. These design decisions clearly limit the generality of the results, but they are sufficient for my current purpose, which is to isolate the effect of evaluation-time bias, rather than to study its interactions with particular representations or reproductive operators. In this experiment, I divide the population's genotype values into bins, taking a histogram at each step. I will use the collected data below to plot heatmaps of the population's genotypic distribution over time, and to test its deviation from a uniform distribution at the end of the runs.

For this second set of experiments, I introduce a new discrete-event simulation (DES) methodology for testing hypotheses about ASEAs. Because this research is focused on the case where fitness evaluation dwarfs other EA overhead, experiments with parallel EAs can be quite time consuming to run if one wishes to obtain statistically significant results. To facilitate doing a large number of runs on artificial problems as well as complete control over the experimental conditions, I implemented a version of both algorithms that is based on a discrete-event simulation, in which each individual $\vec{x}$ is assigned an evaluation time from an artificially-imposed evaluation-time function $t(\vec{x})$ that is defined as part of the experiment. A priority queue is used to instantly jump to the next completed evaluation event when nextEvaluatedIndividual() is called (see Algorithm 2). Because there is always some small amount of variation in the time a function takes to execute an algorithm in vivo on a computer (thanks to process scheduling effects if nothing else), I always add a small amount of Gaussian noise to the simulated evaluation time $t(\vec{x})$, so that this value is not perfectly deterministic in simulation.

Taking this simulation approach introduces some infidelity to real-world conditions: I effectively ignore any effect from algorithmic overhead besides fitness evaluation, from process scheduling in the operating system, etc. When fitness evaluation cost truly dwarfs other factors, there is little threat that this will affect the ability of my conclusions to be replicated on real problems. But it may introduce some deviations in cases where overhead still makes up a small (but nontrivial) fraction of an EA's computational cost.

Table 3.2: Summary of the three main experimental configurations that I use to study evaluation-time bias on flat fitness landscapes.

Domain	Eval-Time	Algorithms	Variation	Environment
$x \in \{\texttt{fast}, \texttt{slow}\}$	$t(x) = \begin{cases} k & \text{if } x = \texttt{fast} \\ 10k & \text{if } x = \texttt{slow} \end{cases}$	ASEA	Cloning only	Multiprocessing with `sleep()`
$x \in (-10, 10)$	$t(x) = x^2$	ASEA, QGEA	Gaussian mutation	Discrete-event simulation
$x \in (-10, 10)$	$t(x) = 1 + e^{-\frac{1}{2}\left(\frac{x+5}{\sigma}\right)^2} - e^{-\frac{1}{2}\left(\frac{x-5}{\sigma}\right)^2}$	ASEA, QGEA	Gaussian mutation	Discrete-event simulation

The experiments that I have just described are summarized in Table 3.2.

Results

In the discrete domain of `slow`- and `fast`-type genotypes, I find that apart from a very short transient period at initialization, the average genotype of the population in the ASEA stays very close to 0.5 throughout the run—as shown on the left-hand plot of Figure 3.13. For most of the run, 95% confidence intervals on the mean genotype straddle 0.5. The only exception appears to be the first few evolutionary steps immediately after initialization: here there appears to be a slight bias toward `fast`-type individuals. I found similar results for population sizes of 10 and 500 (not shown), and also for similar experiment in which genotypes can vary continuously along an interval under genetic drift (not shown—other results on continuous landscapes are reported below). This first result is **consistent with both Hypotheses 3.13 and 3.14**: after a brief initial evaluation-time bias, there is no further detectable bias toward `fast` genotypes, and genetic drift leads the population's genotypes to diverge randomly.

While these results lead to strong predictions about the behavior of algorithms in the expectation, this insight is useful only for characterizing the ensemble averages of a sufficiently large number of finite-population models: not the behavior of individual runs. Notice that in Figure 3.13 the variance in genotype frequency over the 50 independent runs continues to increase over time, even as the mean is roughly constant. This is an indication that there are in fact significant changes in the `slow`/`fast` ratios on individual runs.

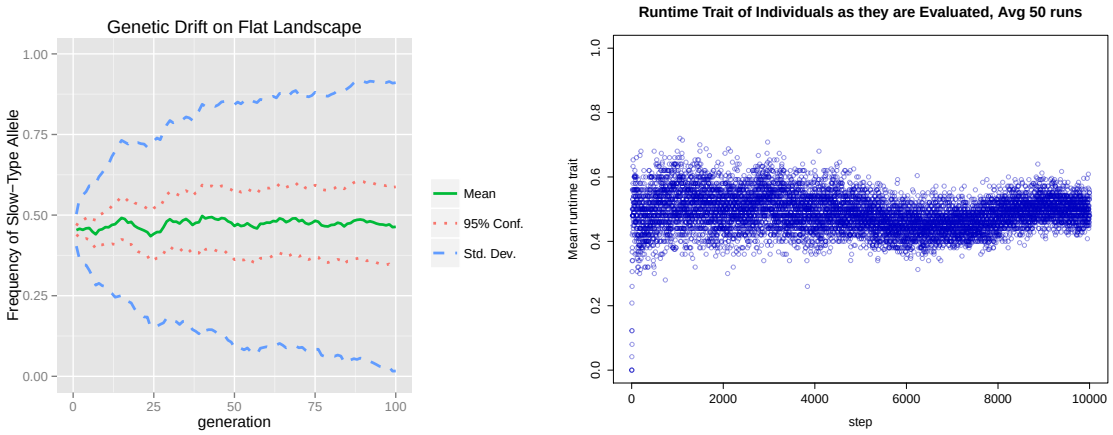

Figure 3.13: **Left**: Frequency of `slow`-type individuals on a flat fitness landscape, averaged over 50 runs. No systematic drift toward `fast` alleles is observed. **Right**: Each point represents the fraction of individuals out of the same 50 independent runs that finished evaluating with `slow` type at that step. Since the fraction hovers around 0.5, the frequency of each genotype does not change very much.

The right side of Figure 3.13 shows an alternative view of the same experiment: this scatter plot shows the fraction of individuals that completed evaluation at step i that are of `slow`-type. In all but just the first few steps, these values consistently hover around 0.5 without any systematic trend up or down. This supports the claim that that inspired Hypothesis 3.14: the system completed evaluating an approximately equal number of `fast`- and `slow`-type jobs in equal time over long time periods.

The rest of the experiments in this section—involving real-valued genotypes in a discrete-event simulation of the ASEA and QGEA—yield similar conclusions. Figure 3.14 shows the mean distribution of genotypes in the ASEA population as a function of the number of evolutionary steps for the experiment in which the evaluation times are configured according to the parabola $t(x) = x^2$, averaged over 500 runs. The top heatmap shows data from runs with a population size of 100, while the bottom shows a population size of 10. In each plot, the dashed line represents the end of the initialization phase—i.e., the step at which the population is first completely filled. Plots for similar experiments with the QGEA are not shown, but were qualitatively similar.

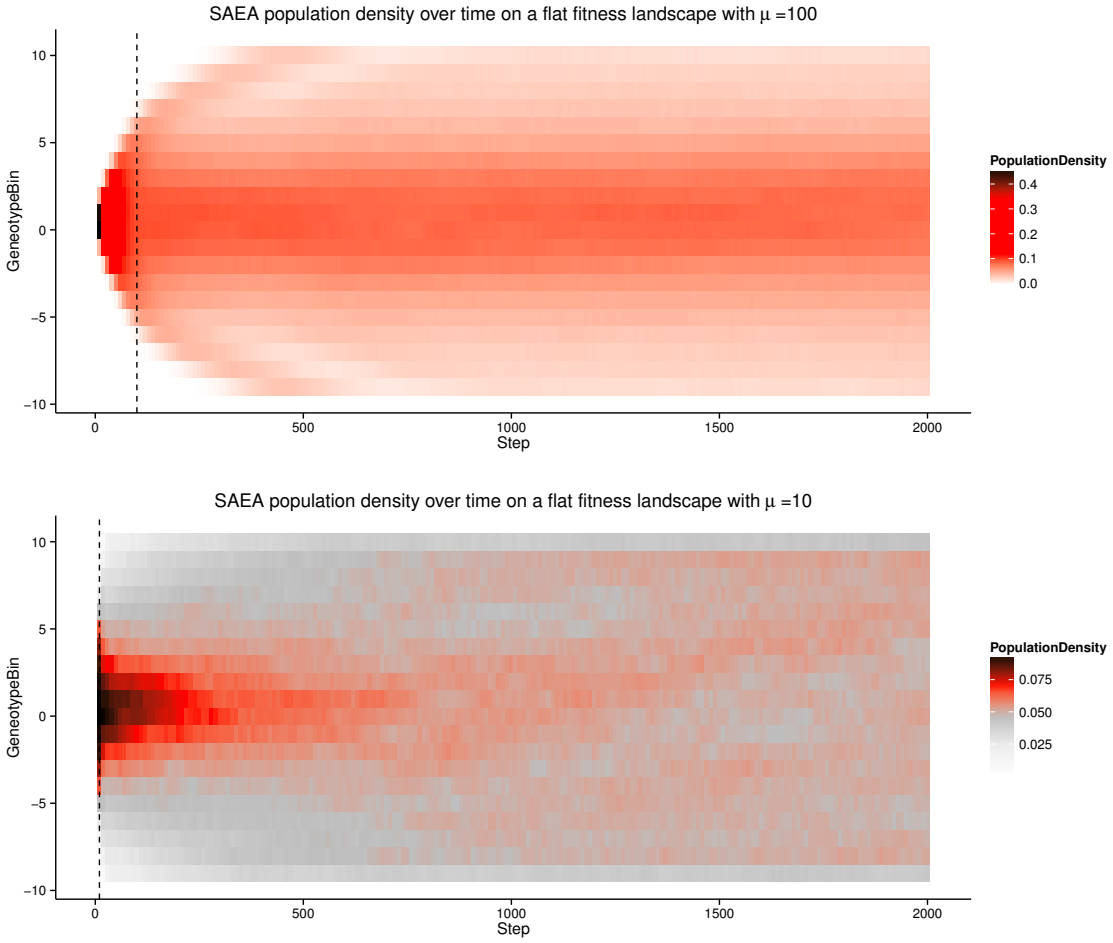

Figure 3.14: Distribution of genotypes over time in an ASEA on a one-dimensional flat fitness landscape, where individual evaluation times are equal to the square of the genotype. **Top**: Population size of $\mu = 100$. **Bottom**: $\mu = 10$.

At the beginning of the run, a clear pattern of initialization bias emerges: individuals with very short evaluation times (i.e., $x \approx 0$) enter the population first, while the slowest individuals do not begin to enter the population until around step 400 (in the `pop_size` = 100 case). When the population is large, I observe that the initial overrepresentation of fast individuals persists and undergoes negligible change later in the run. This can be explained by the well known property of evolution systems, in which genetic drift is weak for large population sizes, leading genotype frequencies to change very little over time.

When the population is small, however (10), the effect of mutation and drift are strong enough to quickly erase the effects of initial evaluation-time bias. In fact, by step 2,000, the average distribution of the population for both the ASEA and the QGEA (not shown) is statistically indistinguishable from a uniform distribution (using a Kolmogorov-Smirnov test with $p > 0.05$).

I obtained similar results with the two-Gaussian form of the evaluation-time function $t(x)$ (this function is depicted on the left-hand side of Figure 3.15). At the end of 2,000 iterations, the distribution of genotypes in the population over many independent runs was statistically indistinguishable from a uniform distribution. The right-hand side of Figure 3.15 shows the observed final distributions of the ASEA and the QGEA, along with a similar dataset sampled from a purely uniform distribution (for visual comparison).

I observed similar results (not shown) for cases where evaluation times followed a linear form ($t(x) = x - 10$), and with a Rastrigin function (i.e., following Equation 3.34). Overall, the results of this section strongly support the conclusion that any evaluation-time bias that exists on flat fitness landscapes after the initialization period is so weak that, in both the ASEA and the QGEA, it can be entirely overpowered by the disruptive effects of mutation and drift. This **supports both Hypotheses 3.13 and 3.14**.

Conclusions on Flat Landscapes

It is clear from these experiments that evaluation-time bias only occurs in special circumstances. Overall, the apparent advantage that fast-evaluating individuals may obtain while

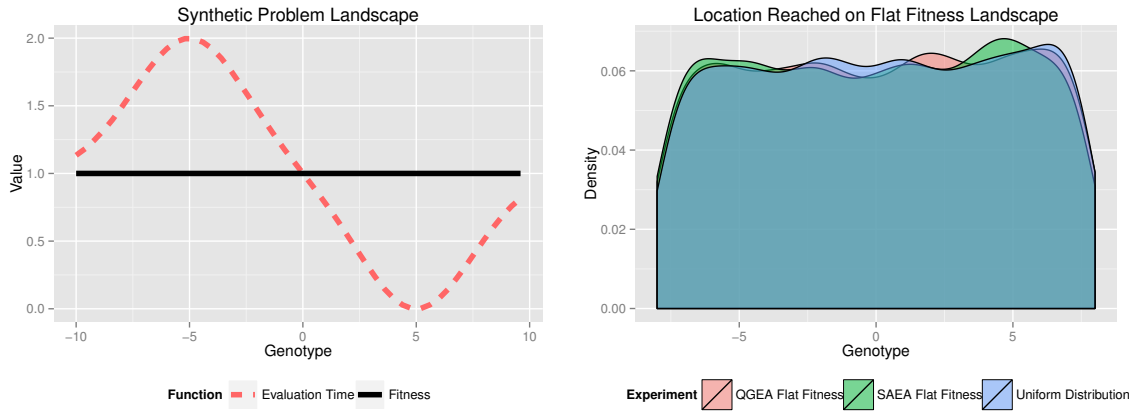

Figure 3.15: **Left**: A synthetic evaluation-time function over a 1-dimensional genotype space, and a flat fitness function. **Right**: Distribution of individuals in the final population after 2,000 evolutionary steps.

long-evaluating individuals evaluate on some processors seems to be completely balanced out by the fact that long sequences of consecutive fast-evaluating jobs are unlikely to be assigned to the same processor.

This does not, however, rule out an evaluation-time bias that emerges from a combination of reproduction, variation and selection, so my results do not contradict the bias that authors such as Yagoubi et al. [2011] have observed in practice. In the next section, I will turn my attention to non-flat fitness landscapes.

3.4.3 Evaluation-Time Bias on Multimodal Problems

Recall that in section 3.3.3 I was surprised to find only very limited evidence that evaluation-time bias could lead the ASEA to behave more greedily or to fall into local optima on simple real-valued optimization problems. In the section 3.4.2 I have now shown that no significant evaluation-time bias appears on flat landscapes in the ASEA or QGEA after a brief initial transient period.

In this section I test the degree to which this observation generalizes to non-flat fitness landscapes, and in particular how evaluation-time bias can divert search away from a promising optimum and toward a sub-optimal one. For this analysis, I focus on an artificial problem with two large basins of attraction—one around a local optimum and one around a global optimum—to design an experiment that allows me to detect even very weak biases. This provides a more direct window into understand evaluation-time bias on multimodal problems than the more general experiments I presented in section 3.3.3.

Hypotheses

I continue to study both the ASEA and the QGEA. Both may exhibit some evaluation-time bias:

Hypothesis 3.15. *Let the quasi-generational EA and the steady-state asynchronous EA be run on a fitness landscape that has two basins of attraction, and let one basin of attraction have fast evaluation times, and the other have long evaluation times. Both algorithms will be more likely to converge to the fast basin than the slow one.*

But the QGEA was introduced specifically to reduce this bias—raising a second hypothesis:

Hypothesis 3.16. *The ASEA will exhibit a stronger preference for the fast basin than the QGEA does.*

If supported, this would provide evidence in favor of the conjecture of Fonseca and Fleming [1998] that the QGEA has a reduced "bias towards individuals easy to evaluate" compared to the ASEA.

Experiments targeting these two hypothesis may show that bias occurs at some point during the run, but not when. But I know from section 3.4.2 that both asynchronous EAs are primarily biased at initialization. Is initialization bias by itself sufficient to explain observations of evaluation-time bias on non-flat landscapes?

Hypothesis 3.17. *Most of the evaluation-time bias in the QGEA and ASEA on a non-flat fitness landscape occurs during the initialization stage of the run.*

Methods

I test the above hypothesis on an objective function defined over $\mathbb{R}^n$ that has two Gaussian basins of attraction centered on local minima A_0 and B_0:

$$\begin{aligned} f_{a,b}(\vec{x}) = & \max(|a|, |b|) \\ & - a \exp\left(-\frac{1}{2\sigma^2} \sum (x_i - 2\sigma)^2\right) \\ & - b \exp\left(-\frac{1}{2\sigma^2} \sum (x_i + 2\sigma)^2\right). \end{aligned} \tag{3.37}$$

I set $\sigma = 2.5$ and use bounds of $(-10, 10)$ when initializing and mutating each gene. When the depth parameters a and b are equal, the two basins are identical from a fitness perspective. Because of the symmetry in the objective function, most evolutionary algorithms will converge to either of the two optima with equal probability.

In the experiment in this section, $a = b$, so the two basins form a symmetric objective function. I introduce an asymmetry into the problem, however, by assigning each individual $\vec{x}$ a simulated evaluation time $t(\vec{x})$, where the eval-time function $t : \mathbb{R}^n \mapsto \mathbb{R}$ is given by:

$$t(\vec{x}) = f_{a,-b}(\vec{x}). \tag{3.38}$$

Evaluation-time is thus equal to fitness, except that evaluation-time basin surrounding optimum B_0 is inverted. This ensures that A_0 is surrounded by fast-evaluating solutions, and B_0 is surrounded by slow-evaluating solutions.

In the first set of experiments, I ran $N = 5,000$ independent runs of both asynchronous EAs (the ASEA and the QGEA) out to 2,000 steps—long enough so that all N runs converged. The population size is 10, and I use 10 simulated worker processors in a discrete-event simulation (just as described in section 3.4.2), and the search space has two dimensions. The

result of each run can be represented by the random variable

$$R_i = \begin{cases} 1 & \text{if } \|\vec{x}' - A_0\| \leq \tau \\ \\ 0 & \text{if } \|\vec{x}' - B_0\| \leq \tau \end{cases}, \tag{3.39}$$

where $\vec{x}'$ is the best individual discovered in the run, A_0 and B_0 are the locations of the two optima, and τ is a small number that serves as a convergence threshold. Now, the number of runs that converge to basin A is $\sum_{i=1}^{N} R_i$, which follows a binomial distribution with proportion parameter r equal to $P(R = 1)$. I use the Wilson method [Brown et al., 2001] to compute 95% confidence intervals around the binomial proportion r. If the asymmetry in evaluation times has no effect on the basin the algorithm coverages to, then r will be equal to 0.5.

I also perform a second set of experiments to target initialization bias in particular: in these, I configured both the fitness function $f(\vec{x})$ and the evaluation-time function $t(\vec{x})$ to have a *constant* value for the first 100 steps of each evolutionary run. After 100 steps, the functions revert to their original values of $f(\vec{x}) = f_{a,b}(\vec{x})$ and $t(\vec{x}) = f_{a,-b}(\vec{x})$ (with $a = b$). This way there is no evaluation-time bias during initialization. Any bias that is still measurable is due to the dynamic interaction between the fitness landscape and evaluation time.

Results

The empirical results of the first two-basin experiment are shown in Figure 3.16. Both the QGEA and the ASEA display a statistically significant preference for the faster basin, **confirming Hypothesis 3.15**. However, I find that the QGEA displays a *stronger* evaluation-time bias than the ASEA. The difference is small, but statistically significant at $p < 0.05$.

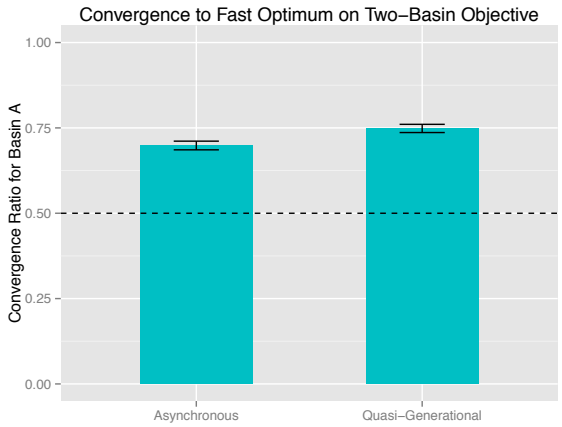

Figure 3.16: Ratio of runs that converge to the faster of two basins of attraction.

On these grounds, **Hypothesis 3.16 is unsupported**. Under the conditions of the two-basin experiment, it turns out that the QGEA is in fact quite biased toward fast solutions, and more so than the ASEA.

Recall that the second two-basin experiment examined initialization bias by artificially holding evaluation times constant for the first 100 steps. The results of this experiment—again as estimates of the probability of each algorithm converging to optimum A_0—are shown in Figure 3.17. Removing initialization bias caused a significant reduction in this measure of evaluation-time bias, and this **supports Hypothesis 3.17**. Both algorithms still display a statistically significant preference for optimum A_0, however. This indicates that, *unlike flat fitness landscapes*, I detect an evaluation-time bias on non-flat fitness functions that occurs after initialization.

3.4.4 Slow Evolution in the QGEA

I close my sequence of experiments with the QGEA with a small but important observation: the evolutionary trajectory of the QGEA does not mimic the generational EA closely, but instead takes substantially longer to converge on simple problems.

I show this with an empirical observation of the QGEA's behavior: the QGEA, it turns

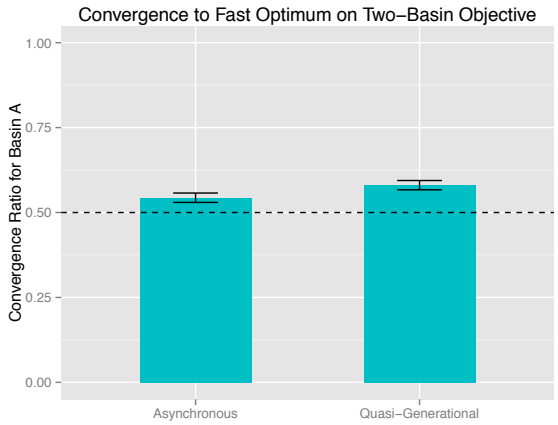

Figure 3.17: Ratio of runs that converge to the faster of two basins of attraction when initialization bias is controlled for.

out, is an algorithm that converges very slowly on unimodal functions. Figure 3.18 shows the number of steps that it takes the generational EA, the ASEA, and the QGEA to reach a threshold fitness value of $\tau = 0.5$ on spheroid functions of differing dimensionality. In this experiment, evaluation times are held constant. Each bar depicts the convergence time of $N = 500$ independent runs.

The quasi-generational EA consistently takes longer to converge than the generational EA. This is a consequence of the asynchronous dynamics of the QGEA, which cause it to generate more than μ children at each generation.

3.4.5 Discussion of Evaluation-Time Bias

Asynchronous evolutionary algorithms are able to make efficient use of parallel processing resources when fitness evaluation is expensive enough to make the master-worker model viable—but asynchronous EAs bring evaluation-time bias with them as side effect. Evaluation-time bias may hinder the results of some applications, have no impact on other applications, and in some cases it may even be beneficial. The current state of the literature does not provide sufficient theoretical insight to make accurate predictions about how

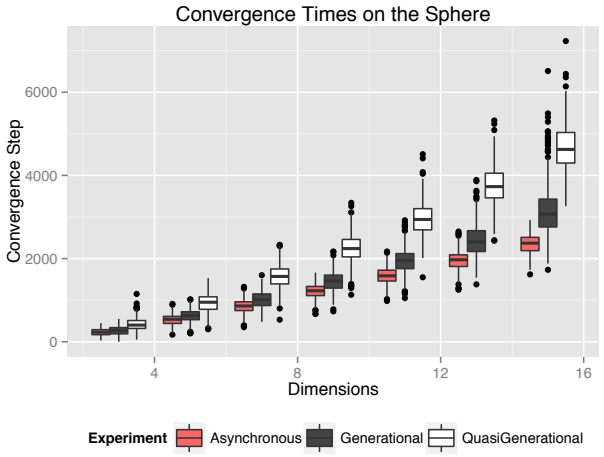

Figure 3.18: Time-to-convergence for master-worker EAs on the spheroid function (Note that the left-most box is a red "asynchronous" box—the color is hard to see because it is so small).

evaluation-time bias will impact EA performance on a particular problem.

The work in this section marks the first attempt that I know of to analyze the nature of evaluation-time bias in asynchronous evolutionary algorithms, especially by distinguishing between bias that occurs at initialization and bias that occurs later in the run. Despite the strong intuitive case to be made that evaluation-time bias can significantly affect the trajectory of an EA under the right conditions (I began my preliminary investigation of evaluation-time bias in section 3.3.3 on the belief that it would have a profound impact on the behaviors of asynchronous evolutionary algorithms), the empirical evidence seems to suggest that the effect is relatively mild after initialization.

This is good news for practitioners, who may be more willing to trust ASEAs with their problems, knowing that they rarely exhibit strong undesired biases.

Following the suggestion of Fonseca and Fleming [1998], I also initiated a study of the quasi-generational EA on the belief that it can serve as a behavioral intermediate to the classical, generational EA and the steady-state asynchronous EA. I found, however, that this is not the case. While both the ASEA and the QGEA display little or no evaluation

time bias on flat landscapes after initialization (section 3.4.2), in the simplified scenarios I used to measure evaluation-time bias, the QGEA has a slightly *stronger* preference for fast-evaluating regions of the search pace than the ASEA (section 3.4.3). The QGEA also shows much slower progress in general while solving simple, unimodal optimization problems (section 3.4.4).

Overall, these results suggest that the QGEA does not have the particular benefits that it has been conjectured to offer.

My experiments here made a number of assumptions that could affect their validity as general conclusions—in particular, I used a fixed mutation width and no crossover, and I assumed that the number of processors was equal to the population size. In my experience, the representation and operators do not have a major effect on the kind of results that I have presented here—I observed some similar results (not presented here), for instance, with higher-dimensional search spaces, with crossover enabled, and with binary representations on simple pseudo-Boolean functions. It may be that these kinds of parameters have a substantial impact on the magnitude of evaluation-time bias in certain circumstances, however—the evidence I present here does not rule this out. Furthermore, the simple two-basin objective that I used to measure evaluation-time bias, while a useful instrument for my purposes here, may not capture the information that is most useful for predicting the behavior of an EA on new problems.

Many open questions still remain about evaluation-time bias in asynchronous EAs that fall outside the scope of this study. Future empirical studies could move closer to a predictive theory of evaluation-time bias by investigating more complex landscapes, particular classes of application domains, or by considering how a population moves asynchronously along smooth fitness gradients. The community also lacks a solid theoretical understanding of asynchronous algorithm behavior—and particularly how the re-ordering effect that induces selection lag interacts with evaluation times to bias the algorithm. My theoretical analysis here was limited to brief observations like Equation 3.36 and Figure 3.12—future work using mathematical tools like drift analysis [Lengler, 2020] might be able to prove more satisfying

and general statements about ASEA behavior, if these techniques can be adapted to handle asynchronous re-ordering effects.

With the most essential questions about evaluation-time bias now answered, in the next section I move on to looking at different aspects of ASEA behavior: namely, how can the benefits of asynchronous fitness evaluation be maximized in cases where better individuals take longer to evaluate than worse ones?

3.5 Selection While Evaluating (SWEET)

In this section, I turn to a specific concern that arises when there is a correlation between evaluation time and fitness. Take the case of evolving a controller for a task such as a cart-pole or vehicle control problem, for example. Here, better-performing individuals may "survive" longer (i.e., don't drop the pole, or don't crash) and thus take longer to evaluate, resulting in fitness evaluation times being positively correlated with fitness scores. In scenarios of this kind, an asynchronous EA can still recover some idle processing resources that a generational EA would incur. The degree to which this additional computation is "useful," however, might be called into question. To make progress on such a problem, the algorithm essentially needs to wait for long-evaluating individuals to complete processing, so that it receives feedback that it can use to continue moving the population toward high-quality solutions.

So it seems that the true speedup over a generational algorithm will be limited in applications like this, where "slower is better" (even if the processing throughput is improved).

In this section, I propose to mitigate this problem of *excess work* in asynchronous evolution with a simple mechanism: I allow parents to be selected randomly from not just the population, but also from among individuals that have *begun evaluation* on a processor but have not yet completed. This gives long-evaluating individuals a chance to reproduce before their fitness has been evaluated, potentially accelerating the propagation of the genetic material that they encode. Stated differently, this approach offers a simple strategy for

neutralizing a large portion of the evaluation-time bias I studied in section 3.4 at its source: by mitigating the tendency for slow-evaluating individuals to be disfavored by selection.

For convenience, I refer to this method as Selection WhilE EvaluaTing (SWEET). My core research question in this section is to ask whether SWEET improves the convergence behavior of ASEAs in the case where "better is slower," and, conversely, whether it has an impact on ASEAs for the opposite class of applications, where "better is faster."

3.5.1 The SWEET Operator

In Chapter 2, I discussed how steady-state evolutionary models admit of two different kinds of selection—namely parent selection and survival selection—and noted that is it common in practice to configure one of these selection stages to use random uniform selection (see section 2.1.4). In terms of the ASEA I study in this chapter, which is defined by parameterizing the general asynchronous EA of Algorithm 2 with the steady-state insertion procedure of Algorithm 3, parent selection occurs as part of the breedOne() procedure, and survival selection is implemented by the selectOne() procedure.

This tradition of using a random selection operator opens up an opportunity that I exploit in this section. Because random uniform selection makes no use of an individual's fitness when choosing an individual, it becomes possible to apply selection to a set that includes individuals that have not yet been evaluated—that is, to Select WhilE EvaluaTing (SWEET). In particular, random selection (and thus SWEET) fits in naturally as a strategy for *parent selection*. This is because survival selection centers on selecting competitors to replace in the current population (which has already had its fitnesses evaluated).

In this section, I obtain a SWEET-based ASEA implementation by using a traditional selection operator—namely binary tournament selection—for survival selection,[21] while using SWEET for the parent selection operator.

In the experiments described below, I evaluate the merits of SWEET by comparing two

[21]This contrasts with the other ASEA configurations earlier in this chapter, which typically use random uniform selection for survival selection, and binary tournament selection for parent selection.

Table 3.3: Overview of all of the benchmark problems used in this study. All use a simulated parallelization environment to model the impact that evaluation-time variance has on the algorithm's behavior.

Problem	Parallelization Type	Domain
Takeover Times	Simulated	N/A
Exponential (Correlated)	Simulated	Real-Valued Optimization
Exponential (Anti-Correlated)	Simulated	Real-Valued Optimization
Exponential (Uncorrelated)	Simulated	Real-Valued Optimization
Two-Basin (Correlated)	Simulated	Real-Valued Optimization
Two-Basin (Anti-Correlated)	Simulated	Real-Valued Optimization
Two-Basin (Uncorrelated)	Simulated	Real-Valued Optimization

ASEA variations:

1. the **basic asynchronous** (basic ASEA) selection strategy, which implements parent selection by randomly choosing and individual from the population P_p, and

2. the **SWEET** strategy (SWEET ASEA), which implements parent selection by applying random selection to the union $P_p + E$ of the population and *currently evaluating individuals*.

To evaluate the performance of SWEET in different asynchronous evaluation environments I use a combination of take-over time analysis and several real-valued optimization problems, which I study using the discrete-event simulation approach that I introduced in section 3.4.2. These benchmark optimization problems are summarized in Table 3.3, and will be described in more detail below. In this section I describe the asynchronous algorithm under study, and the specific hypotheses that I will test in terms of takeover times and performance.

All of the algorithms and simulations in this section were implemented using the Library for Evolutionary Algorithms in Python (LEAP)—a general-purpose toolkit for evolutionary computation research which I have created in collaboration with colleagues, and which has built-in support for ASEAs [Coletti et al., 2020].

3.5.2 Takeover Time Analysis

Takeover times are a traditional methodology for evaluating the rate at which selection operators increase the frequency of high-performing genotypes in an evolutionary metaheuristic [De Jong, 2006]. Takeover time is typically defined as the earliest point at which the genotype of the single best individual in the population completely replaces all other genotypes, reaching a frequency of 100% (from its initial frequency of 1/`population_size`). The purpose of takeover times is to study the evolutionary dynamics of a selection operator specifically, in isolation from other aspects of evolution such as reproductive operators. That is, I separate out the two fundamental sources of evolutionary change—selection and variation—so as to understand the behavior of the former.

In this section, I use takeover times to measure the impact of asynchronous selection strategies on the frequency of a good-fitness, slow-evaluating genotype.[22]

Hypothesis

The basic motivation behind SWEET is to give long-evaluating individuals a chance to reproduce earlier, allowing them to increase their representation in the population sooner than would otherwise be possible.

To test this, I measure the effect that the SWEET strategy—selecting from the combination of the population with the pool of currently-evaluating individual ($P + E$)—has on the rate at which slow-evaluating individuals take over the population:

Hypothesis 3.18. *Given an initial population that contains a* ***single good-fitness, slow-evaluating individual*** *that evolves under cloning and selection alone, the SWEET strategy will lead to significantly faster* takeover times *than the basic asynchronous selection strategy.*

[22] I have published all of the results from this section in Scott et al. [2021] and Scott et al. [in press].

Methods

Like in section 3.4.2, the experiments in this section run an ASEA using a cluster with processors and job-evaluation times simulated by a discrete-event simulation (DES). To facilitate the study of the takeover time of a particular genotype of interest, the initial population is configured to contain exactly one individual of `HIGH` genotype (good-fitness, slow-evaluating). The rest of the population is initialized to have `LOW` genotype (poor-fitness, fast-evaluating). The exact fitness value that is assigned to these genotypes is irrelevant, as I use binary tournament selection for survival selection,[23] and tournament selection considers only the relative rank of individuals (rather than absolute differences in fitness values). I fix the evaluation-time differential between the two genotypes to a ratio of 100 to 1 (i.e., a fast individual evaluates 100x faster than slow ones).

Because the survival selection operator in the ASEA used here uses tournament selection to choose competitors, with zero chance of keeping the poorer individual, the initial high-fitness individual will always successfully compete for a place in the population, and the fraction of `HIGH`-type individuals will grow monotonically under selection and cloning. To formally test Hypothesis 3.18, I report the simulated time (and also number of births) required for each of 50 runs to converge to a population of 100% `HIGH` genotype (i.e., the takeover time).

Asynchronous EA behavior is affected by a variety of parameters, most notably the relation between the size of the population and the number of processors. Here I focus on the simple case where the population size is equal to the number of processors, and I report results for a fixed population size of 50. I note in passing that qualitatively similar results can be obtained for lower and higher population sizes. In these experiments the initialization strategy for both algorithms (see section 3.2.2) is the *immediate* strategy of Algorithm 4—which, because the population size equals the number of processors, is equivalent in this

[23] More specifically, when selection is used to find a *competitor* for survival selection, as in `selectOne()` in Algorithm 3, I use (in this case) binary tournament selection in reverse: the algorithm chooses two individuals randomly, and the *worse* of the two is kept. This is because competitor selection aims to choose a *poor* individual to replace in the population (rather than a good individual to keep).

Table 3.4: Configuration parameters used in the **takeover times** experiment. Reproduction here is via cloning, so the evolutionary dynamics serve only to increase the frequency of high-fitness genotypes, without modifying them.

Parameter	Value
`HIGH` genotype evaluation time	100 simulated seconds
`LOW` genotype evaluation time	1 simulated seconds
Initialization strategy	*immediate*/*until-finished* (Algorithm 4)
Parent selection operator	Random-uniform selection (with or without SWEET, respectively)
Survival selection	Inverted binary tournament selection
Reproduction operators	Cloning only (no variation)
Stopping criterion	Run for 10,000 simulated seconds
Population size	50
# simulated processors	50 (= population size)
# independent runs	50

case to the *until-finished* strategy of Algorithm 5. For the sake of reproducibility, all of the parameters and design decisions I use when testing Hypothesis 3.18 are summarized in Table 3.4.

The notion of a "faster" takeover time can be defined either in terms of births (fitness evaluations) or time. In this experiment I report both metrics—but because time is a simulated variable in our cluster DES methodology, I report "simulated seconds" rather than true wall-clock time.

Results

The results of the DES experiment with takeover times are shown in Figure 3.19. The SWEET strategy exhibits a considerably shorter (and statistically significant via Wilcoxon rank-sum at $p \approx 10^{-9}$) median takeover time in terms of simulated seconds (505.5s vs. 621s). Plotting the mean takeover curves shows that the behavior of the two algorithms differs qualitatively as well, with SWEET assuming a sort of pseudo-generational trajectory, as groups of `HIGH`-type individuals repeatedly complete evaluating at nearly the same time.

Overall, these results **confirm Hypothesis 3.18** and suggest that the SWEET strategy allows high-fitness genetic material from long-running individuals to more rapidly increase its frequency within a population. It is worth noting here, however, that the majority of the difference in takeover times with SWEET occurs at the beginning of the run.

3.5.3 Synthetic Real-Valued Optimization Problems

The takeover-time analysis described in the previous section is useful for understanding how selection operators behave in isolation. But—as I noted in section 3.3.3—because the components of an evolutionary algorithm can interact with each other in complex ways, its performance is often best evaluated empirically by running a complete algorithm on various benchmark tasks.[24]

In this section I examine the behavior of a SWEET-based ASEA as it solves several simple real-valued optimization problems. The aim here is to understand SWEET's performance advantages in simple scenarios, on the anticipation that these observations will generalize to real-world applications. While I do not treat any such application here, I have worked elsewhere with colleagues to apply SWEET to some problems in robotics and adversarial machine learning—those results are further reported in Scott et al. [in press].

Hypotheses

Again following the claim that SWEET gives long-evaluating individuals an improved chance at reproducing, I expect that the ASEA with SWEET will perform better at approaching local optima on landscapes when better solution take more time to evaluate:

Hypothesis 3.19. *On **unimodal problems**, when evaluation time is heritable and **positively** correlated with fitness on a maximization problem, an ASEA that uses the SWEET strategy will find the global optimum significantly **faster** than a basic asynchronous strategy on average.*

[24] I have published all of the results from this section in Scott et al. [in press].

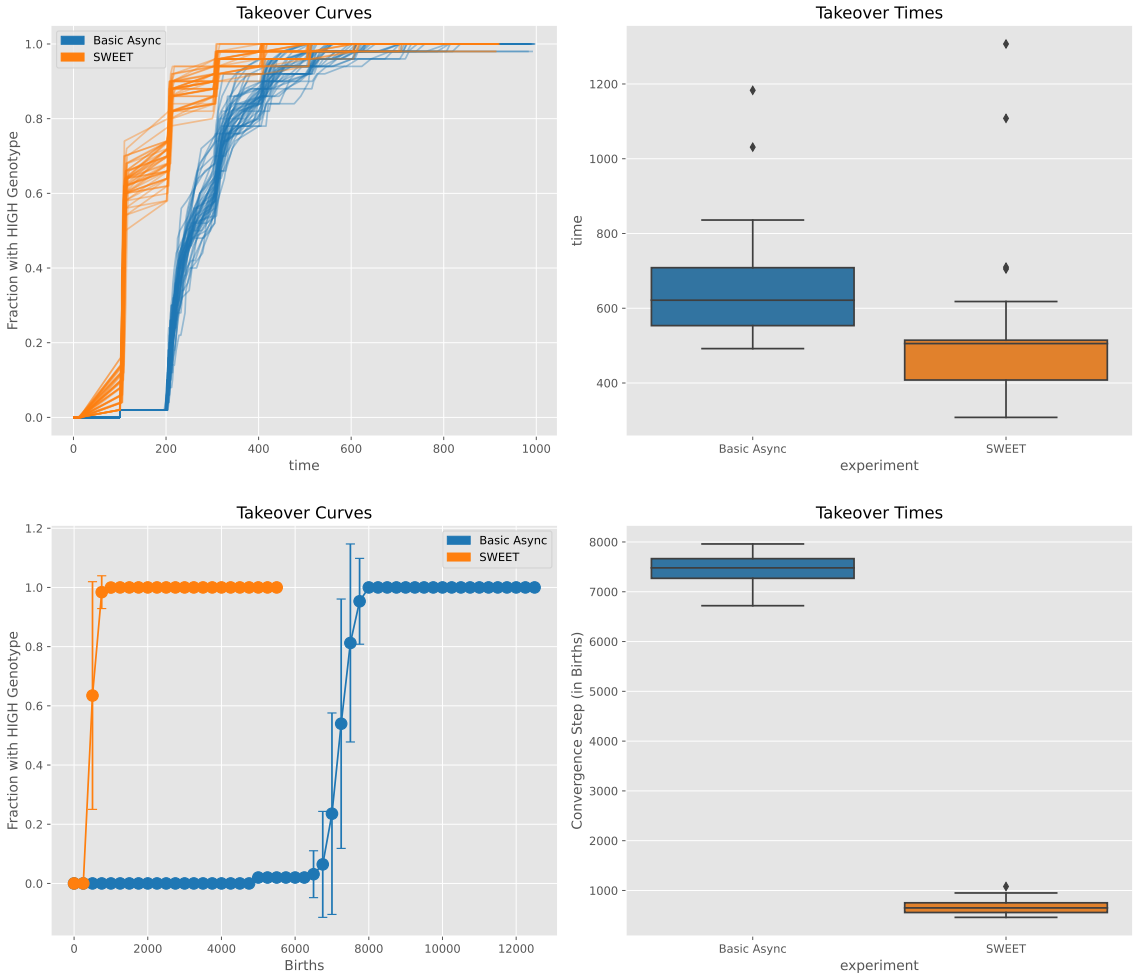

Figure 3.19: Takeover-time experiments for 50 runs under selection & cloning, seeded with exactly 1 `HIGH`-type individual and 49 `LOW`-type. Left: individual `HIGH` frequency curves for each run. Right: distribution of the time until convergence to a `HIGH` frequency of 1.0. The Top plots report metrics in terms of simulated seconds, while the Bottom shows the same data in terms of the number of births.

Hypothesis 3.19 covers the case where SWEET is used on the kind of problem it was designed for. In practice, however, the evaluation-time properties of many problems are not fully understood in advance—or it could be the case that different parts of the search space display different patterns of correlation between evaluation time and fitness. There is risk, then, that a SWEET-based ASEA may be used on problems where better solutions sometimes (or always) evaluate more quickly (rather than more slowly) than poorer ones. In this scenario, then (depending on the state of the population at each step), SWEET may reduce the probability of choosing high-quality parents. On unimodal problems, then (where exploration has little benefit compared to exploitation), using SWEET in an ASEA when "better is faster" may have an adverse effect on performance:

Hypothesis 3.20. *On **unimodal problems**, when evaluation time is heritable and is **anti-correlated** with fitness on a maximization problem, an ASEA that uses the SWEET strategy will the global optimum significantly **less quickly** than a basic asynchronous strategy on average.*

The hypotheses stated so far concern only unimodal functions. A more complex question is how SWEET will affect an asynchronous EA's ability to cope with local optima. I formulate this question in the following hypotheses:

Hypothesis 3.21. *On **multimodal problems**, when evaluation time is heritable and **positively** correlated with fitness on a maximization problem, an ASEA that uses the SWEET strategy will achieve good fitness values **more quickly** on average than a basic asynchronous strategy.*

Hypothesis 3.22. *On **multimodal problems**, when evaluation time is heritable and **negatively** correlated with fitness on a maximization problem, an ASEA that uses the SWEET strategy will achieve good fitness solutions **less quickly** on average than a basic asynchronous strategy.*

Since the algorithms may not always find the global optimum in the multimodal case, measuring convergence times may yield infinite values and thus is not feasible as a measure of performance. Instead, I will operationalize the notion of achieving "good fitness values more quickly on average" by considering the *area under the best-so-far curve* (AUC). This metric takes into account both the final quality of the solution the algorithm finds *and* the speed with which it finds it, and will also serve as a suitable metric for evaluating the unimodal problems referenced by Hypotheses 3.19 and 3.20.

Methods

To provide a concrete test of these hypotheses about the performance of SWEET, I define a benchmark of real-valued optimization problems. Much like in the experiments in other sections of this chapter, each of these problems is defined by a tuple $(f(\cdot), \tau(\cdot))$ made up of a fitness function $f(\cdot)$ and an evaluation-time function $\tau(\cdot)$. Both functions are defined over the domain of genotypes. In this section I focus on three cases from the four-part classification in section 3.3.3: when $f = \tau$, there is perfect, heritable correlation between fitness and evaluation time; when $f = -\tau$, there is perfect, heritable anti-correlation; and if the value of $\tau(\mathbf{x})$ is independent of the genome $\mathbf{x}$, then evaluation time is non-heritable. Specifically, for the non-heritable case in this section, I consider evaluation times that are stochastic and drawn from a uniform distribution on $[0, 1]$.

Specifically, for the benchmark in this section, I construct six different problems in this way by choosing the forms of $f(\cdot)$ and $\tau(\cdot)$ from five different functional forms (both summarized in Table 3.5). In all cases, ten-dimensional versions of the problems are constructed ($n = 10$), and the search space is bounded within the hypercube $[-2, 2]^{10}$. These problems can be divided into two categories, corresponding to unimodal cases (Hypotheses 3.19 and 3.20) and multimodal cases (Hypotheses 3.21 and 3.22, respectively):

1. The **"exponential" problems** are unimodal, and are designed to introduce exaggerated differences in fitness values/evaluation times among individuals as evolution

Table 3.5: **Top**: the five functions I use as building blocks to create the synthetic problems used in this section. **Bottom**: the six synthetic problems that I define by choosing different functions to define their fitness landscapes and evaluation-time properties.

Function Name	Expression
Exponential-growth	$f(\mathbf{x}) = \exp\left(\sum_i^n x_i\right)$
Exponential-decay	$f(\mathbf{x}) = \exp\left(-\sum_i^n x_i\right)$
Two-basin (A higher)	$f(\mathbf{x}) = 10 \cdot \exp\left(-\sum_i^n \left(\frac{x_i - \mathbf{x}_a}{w}\right)^2\right) + \exp\left(-\sum_i^n \left(\frac{x_i - \mathbf{x}_b}{w}\right)^2\right)$
Two-basin (B higher)	$f(\mathbf{x}) = \exp\left(-\sum_i^n \left(\frac{x_i - \mathbf{x}_a}{w}\right)^2\right) + 10 \cdot \exp\left(-\sum_i^n \left(\frac{x_i - \mathbf{x}_b}{w}\right)^2\right)$
Random-uniform	$f(\mathbf{x}) \sim \mathcal{U}(0, 1)$

Problem Name	Fitness Function $f(\cdot)$	Evaluation-Time Function $\tau(\cdot)$
Exponential (Correlated)	Exponential-growth	Exponential-growth
Exponential (Anti-Correlated)	Exponential-growth	Exponential-decay
Exponential (Uncorrelated)	Exponential-growth	Random-uniform
Two-Basin	Two-Basin (A higher)	Two-Basin (A higher)
Two-Basin (Anti-Correlated)	Two-Basin (A higher)	Two-Basin (B higher)
Two-Basin (Uncorrelated)	Two-Basin (A higher)	Random-uniform

progresses. This is valuable because differences in evaluation time are what create the complex dynamics of asynchronous algorithms, and this gives us a simple environment within which to test those dynamics.

2. The **"two-basin" problems** are constructed (similarly to section 3.4.3) by summing two Gaussian functions centered over different regions of the search space, one of which has an amplitude ten times higher than the other. These problems provide a scenario in which evolutionary algorithms are prone to getting stuck in a local optimum, from which it is difficult to escape. For the "two-basin" functions, the offset of basin A is a constant $\mathbf{x}_a = [-1, \cdots, -1]^\top$, while the offset of the other basin (B) is $\mathbf{x}_b = [1, \cdots, 1]^\top$.

As for the algorithm used in our synthetic experiments, it is similar to the one described above for the takeover times experiment, except that I introduce an additive Gaussian mutation operator which mutates genes by adding a value randomly sampled from a Gaussian

Table 3.6: Configuration parameters used in the **synthetic real-valued optimization problems** experiment.

Parameter	Value
Genome representation	Real-valued vectors
Search space dimensions n	10
Search space bounds	$[-2, 2]^n$
Initialization strategy	*immediate*/*until-finished* (Algorithm 4)
Parent selection	Random-uniform selection (with or without SWEET, respectively)
Survival selection	Inverted binary tournament selection
Reproduction operators	Additive Gaussian mutation only (no crossover)
Mutation σ	1.5
Mutation probability	$1/n = 0.1$
Stopping criterion	Run for 5,000 births
Population size	50
# simulated processors	50 (= population size)
# independent runs	100

distribution: $g' = g + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma)$. For this analysis, I do not include crossover, and I use a fixed mutation width of $\sigma = 1.5$. As in the takeover-time experiments in the preceding section, I used an *immediate* initialization strategy for both algorithms (see section 3.2.2), which is equivalent to the *until-finished* strategy of Algorithm 5 since the population size equals the number of processors here.

Table 3.6 summarizes the design parameters for the synthetic real-valued optimization experiments with SWEET. In many contexts, it would be important to tune these parameters via a suitable hyperparameter-tuning algorithm before comparing algorithms against each other on real-world problems. In this study, however, I am narrowly concerned with the effect of modifying asynchronous selection strategies in particular; having highly tuned mutation parameters, population size, etc., therefore is unlikely to modify my conclusions.

Results

The results of the DES experiments with SWEET on real-valued optimization problems come in two parts: the exponential (unimodal) problems that test Hypotheses 3.19 and 3.20, and the two-basin (multimodal) problems that test Hypotheses 3.21 and 3.22.

The results on the exponential functions are depicted in Figures 3.20, along with visualizations made from 2-D projections of the functions $f(\cdot)$ and $\tau(\cdot)$ for reference (though recall that the full problems themselves are 10-dimensional). Recall that in the correlated, anti-correlated, and uncorrelated cases, $f = \tau$, $f = -\tau$, and $\tau \sim \mathcal{U}(0, 1)$, respectively. I report the results here in terms of births (i.e., evolutionary steps). In the correlated case (where "better is slower"), SWEET clearly outperforms the control, consistently obtaining a higher AUC and **confirming Hypothesis 3.19**. In the remaining cases, the performance of the two algorithms shows little difference. The basic ASEA does take fewer steps to converge (and have a lower AUC) in both the anti-correlated and the uncorrelated experiments, and the difference in the median AUC values under these conditions is statistically significant (Wilcoxon rand-sum test, $p < 0.0005$). But the effect size here is very small (as can be seen by examining the box plots in the center and bottom rows of Figure 3.20, which show considerable overlap). So I consider **Hypothesis 3.20 to be partially supported**: SWEET does appear to take slightly more steps to converge on unimodal functions, but the difference is so small that it may be safe to ignore in applications.

The results on the two-basin functions are depicted in Figures 3.21, again with 2-D projections to visualize these 10-dimensional function surfaces. These results are more nuanced than the unimodal case: I do observe that when fitness and evaluation time are directly correlated, SWEET converges more rapidly on a solution, achieving a superior AUC score in the median ($p < 0.0005$). SWEET also exhibits higher variance in its fitness curve, however, suggesting that it achieves faster convergence at the cost of having a higher probability of falling into a local optimum. So while I consider **Hypothesis 3.21 to be mostly supported** by this data, it comes with the simple observation that faster convergence may not always lead

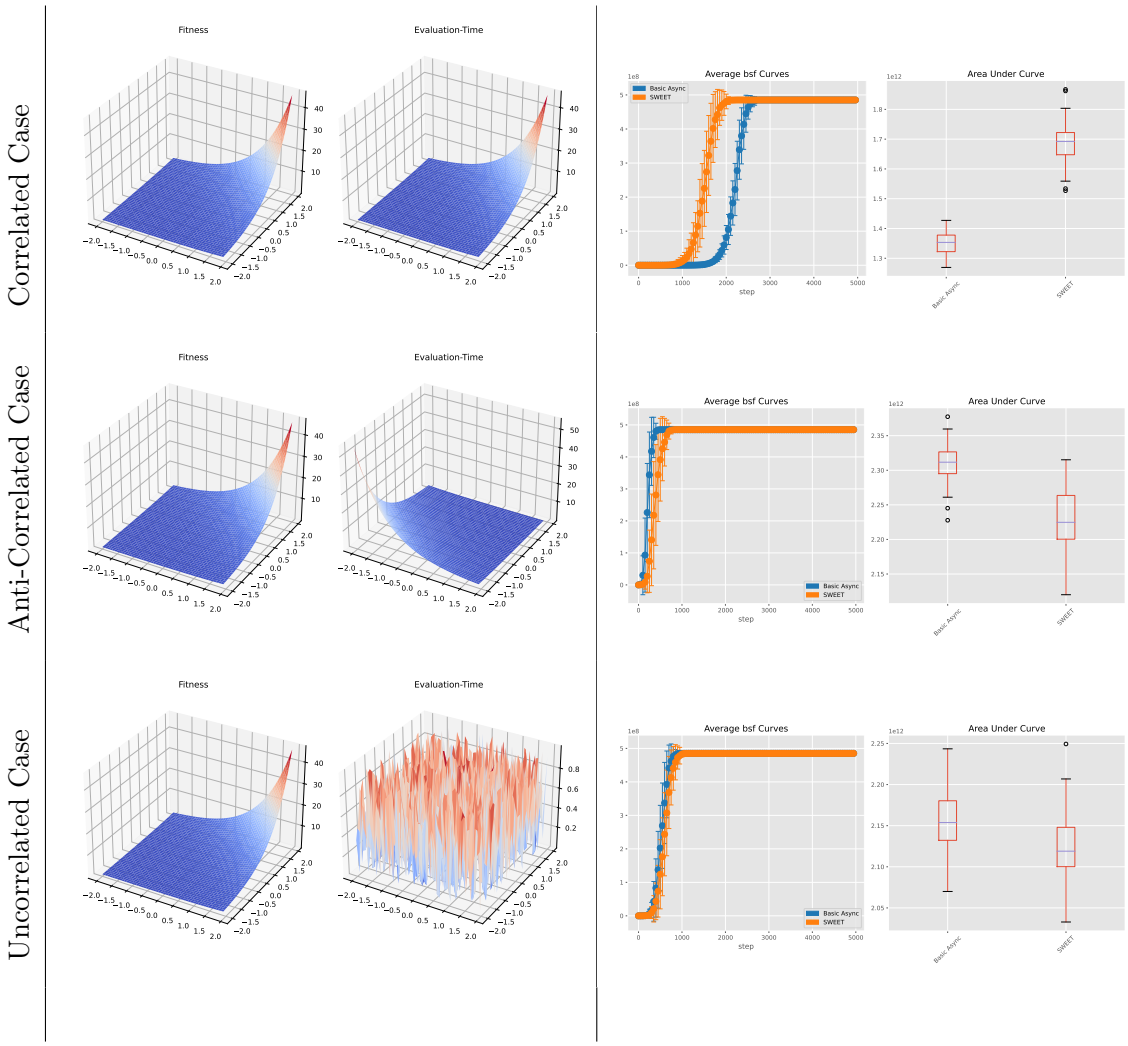

Figure 3.20: Mean best-so-far fitness curves and AUC metrics for SWEET and the Basic Async algorithm applied to the exponential benchmark problems. Shown on the left are 2-D projected visualizations of each 10-D synthetic problem's properties. Error bars on the best-so-far curves indicate the standard deviation across 100 independent runs.

Table 3.7: Median area-under-curve (AUC) values for 100 runs of the two algorithms on each problem. Bold numbers are statistically significantly the best-performer in each row by a Wilcoxon rank-sum test, and the triple-asterisks (***) indicate $p < 0.0005$.

Problem	Evaluation Times	Basic ASEA	SWEET
Exponential	Correlated	1.35e+12	**1.69e+12*** **
Exponential	Anti-correlated	**2.31e+12*** **	2.22e+12
Exponential	Uncorrelated	**2.15e+12*** **	2.12e+12
Two-Basin	Correlated	2.41e+4	**3.11e+4*** **
Two-Basin	Anti-Correlated	2.67e+4	**3.39e+4*** **
Two-Basin	Uncorrelated	**4.16e+4*** **	3.97e+4

to better solutions (this is in line with classic arguments based on an exploration-exploitation trade off). The anti-correlated case offers an additional surprise: although SWEET was introduced in order to solve correlated ("better is slower") problems, and I expected it to have a harmful performance effect on anti-correlated ("better is faster") problems, here I observe that SWEET offers a clear and non-trivial improvement over the basic ASEA on a moderately anti-correlated multi-modal problem. This **disconfirms Hypothesis 3.22**, which posited that no improvement would be seen in the anti-correlated case. In the uncorrelated case, there is a small, statistically significant difference in median behavior between the two algorithms, favoring the basic ASEA, but the difference is so small as to be practically negligible. I also observe high variance in both algorithms in this scenario—suggesting that the noisy evaluation-time landscape significantly increases the chance of falling into a local optimum.

The median AUC values measure for SWEET and the basic control on all six benchmark problems are summarized in Table 3.7, along with asterisks indicating that all differences were statistically significant at $p < 0.005$.

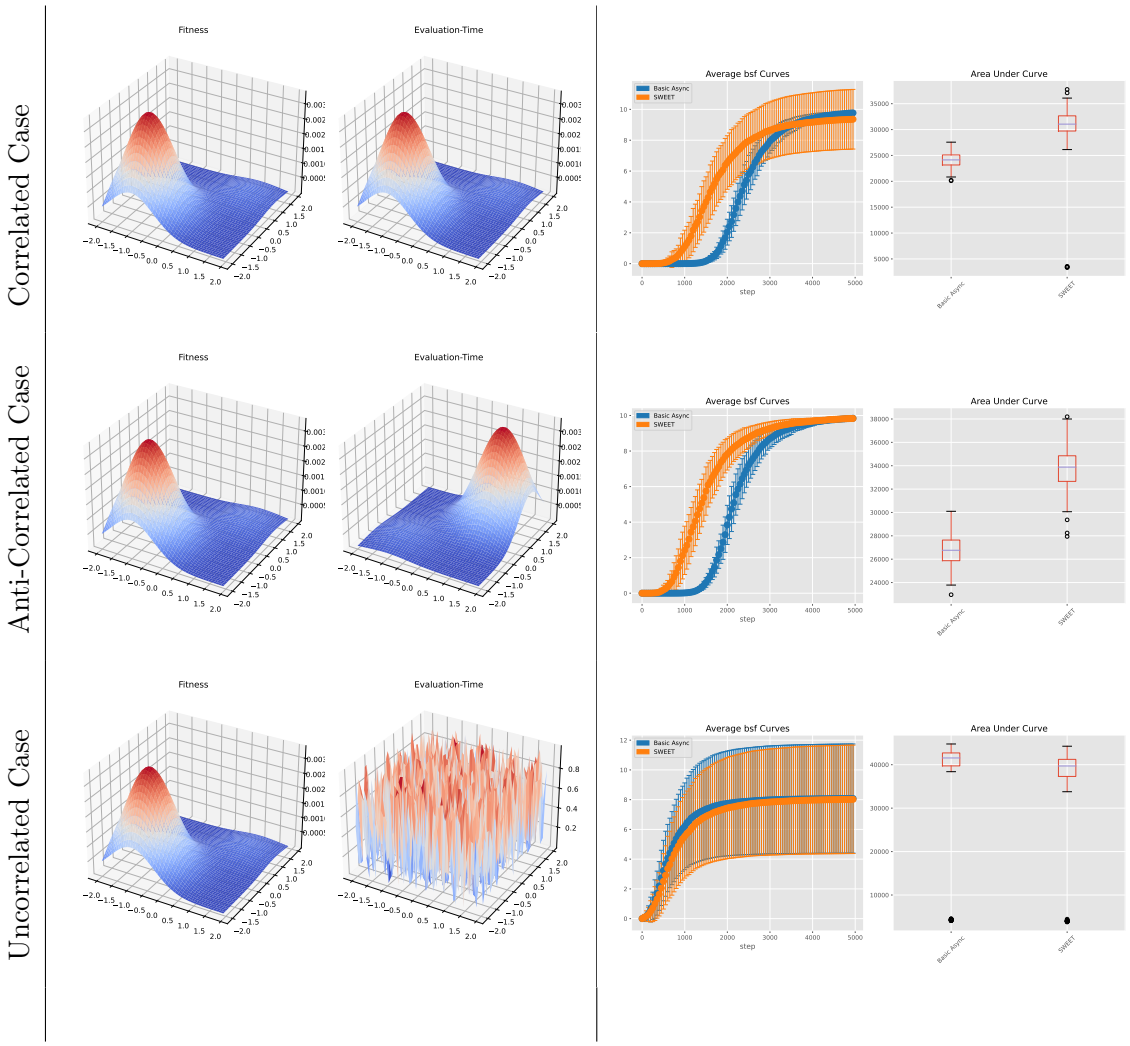

Figure 3.21: Mean best-so-far fitness curves and AUC metrics for SWEET and the Basic Async algorithm applied to the two-basin benchmark problems. Shown on the left are 2-D projected visualizations of each 10-D synthetic problem's properties. Error bars on the best-so-far curves indicate the standard deviation across 100 independent runs.

3.5.4 Discussion of SWEET

I have proposed a simple steady-state parent selection strategy, SWEET ("Selection WhilE EvaluaTing"), which is intended to offset the tendency that asynchronous steady-state EAs have to inefficiently sample an excess of fast-evaluating solutions at the expense of higher-quality, slow-evaluating ones. Simulated experiments with takeover times suggest that this method is able to more quickly increase the frequency of high-performing, long-evaluating alleles. And results on synthetic real-valued optimization problems suggest that SWEET is effective at boosting problem-solving performance in the case of a positive fitness-eval-time correlation.

The experiments in this section are limited in that I used only a few, simple real-valued landscapes to test the algorithms' behavior, and I considered only a few fixed parameter values (for example, assuming that the population size is equal to the number of processors). Future work will need to investigate the practical impact of SWEET more thoroughly on a variety of real-world applications to see if the conclusions drawn here generalize effectively to practical settings. I have begun this research in collaboration with colleagues [Scott et al., 2021, in press]: we have applied SWEET to some initial applications that include evolving spiking neural networks for autonomous vehicle control, and to adversarial methods that use an evolutionary algorithm to improve the robustness of a reinforcement learner to different environmental conditions.

Overall, the experiments presented here suggest that SWEET is a promising optimization to ASEAs, and that in some applications it can have a moderate impact on the efficiency of an algorithm's ability to find high-quality solutions with a limited budget. In particular, its tendency to adversely impact performance when evaluation times in fact decrease with quality (rather than increase) appears to be significantly smaller than I believed when beginning this research. As with many meta-heuristic enhancements, however, it remains difficult to predict what problem properties might cause this ASEA strategy to lead to improved performance.

One factor that I observed in the simulation of takeover times is that SWEET tended to improve performance early, but as longer evaluating individuals take over the population, it has much less of an impact. On useful avenue for future research may be to investigate the use of SWEET as only during the initial portion of an ASEA run, pivoting to a traditional selection approach later in the run.

3.6 Conclusions & Discussion

In Table 3.8 I summarize all of the hypotheses that I have tested in this chapter and their conclusions. As indicated by the structure of the table, my research questions from section 2.1.5 have led me to six different lines of experimental inquiry in total.

3.6.1 Research Question 1: Speedup

Research Question 1 led me to a multi-faceted investigation of the performance of ASEAs in terms of speedup. My first contribution in this area has been a novel analysis of initialization strategies in ASEAs in Hypotheses 3.1–3.4—which I have shown can have an outsized impact on the behavior and performance of an algorithm, and should thus be approached with more care than they traditionally are in other areas of evolutionary computation. In particular, when the number of processors T is significantly less than the population size μ, the *immediate* initialization strategy suffers from queueing effects that significantly slow the fundamental feedback loop that drives evolution. In practice, knowing that initialization strategies should be treated with care is an important lesson. This has had an immediate effect of the code that my colleagues and I have written for present and future projects—as for years we were unaware of the pitfalls of asynchronous initialization. A simpler alternative practice, however, may be to hybridize an ASEA by simply using *synchronous initialization*—bypassing the subtleties of asynchronous initialization altogether, at the cost of a brief period of idle time. I have not recommended this approach as the best solution, however, because I have not yet studied this strategy to ensure that it doesn't lead to unexpected

Table 3.8: Summary of the hypotheses tested in Chapter 3.

	Initialization Strategy Experiments	
Hyp. 3.1	*Extra* and *until-finished* initialization behave similarly when $T << \mu$	Supported
Hyp. 3.2	ASEA with *Immediate* initialization converges slowly when $T << \mu$	Supported
Hyp. 3.3	*Immediate* and *until-finished* initialization the same when $T = \mu$	Supported
Hyp. 3.4	*Extra* initialization behaves differently when $T = \mu$	Supported
	Throughput Speedup Experiments	
Hyp. 3.5	ASEA throughput speedup obeys the analytic lower bounds given by Theorem 3.2	Mostly supported
Hyp. 3.6	ASEA throughput speedup decreases with population size	Supported
	True Speedup Experiments	
Hyp. 3.7	ASEA greedier overall than generational on mostly unimodal problems	Supported
Hyp. 3.8	ASEA speedup has high variance when eval-time heritable but fitness-neutral	Supported
Hyp. 3.9	ASEA more greedy when better is faster on mostly unimodal problems	Unsupported
Hyp. 3.10	ASEA less greedy when better is slower on mostly unimodal problems	Unsupported
Hyp. 3.11	ASEA more prone to local optima overall on highly multimodal problem	Unsupported
Hyp. 3.12	ASEA more prone to local optima when better is faster on highly multimodal problem	Mostly supported
	Evaluation-Time Bias on Flat Landscapes	
Hyp. 3.13	ASEA and QGEA exhibit evaluation-time bias at initialization on flat landscapes	Supported
Hyp. 3.14	ASEA and QGEA exhibit no evaluation-time bias after initialization on flat landscapes	Supported
	Evaluation-Time Bias on Multimodal Problems	
Hyp. 3.15	ASEA and QGEA exhibit evaluation-time bias on two-basin problems	Supported
Hyp. 3.16	QGEA is less biased than ASEA on two-basin problems	Unsupported
Hyp. 3.17	Most evaluation-time bias in the ASEA & QGEA is at initialization	Supported
	Selection While Evaluating (SWEET)	
Hyp. 3.18	SWEET leads to faster takeover times in the ASEA	Supported
Hyp. 3.19	SWEET outperforms a basic ASEA on unimodal problems when better is slower	Supported
Hyp. 3.20	A basic ASEA outperforms SWEET on unimodal problems when better is faster	Partly supported
Hyp. 3.21	SWEET outperforms a basic ASEA on multimodal problems when better is slower	Mostly supported
Hyp. 3.22	A basic ASEA outperforms SWEET on multimodal problems when better is faster	Unsupported

effects during the transition from synchronous to asynchronous evaluation. Some applications, moreover, exhibit the most evaluation-time variance early in the run, in which case synchronous initialization of the population may still lead to unacceptably high levels of idle resources. Nevertheless, this hybrid approach is one avenue that I hope to pursue in future research.

Next I turned to the problem of quantifying the idle time that generational EAs incur in Hypotheses 3.5 and 3.6, and I have proved a general result in Theorem 3.1 and a specific algebraic bound in Theorem 3.22 that very closely quantifies the degree of throughput speedup that an ASEA exhibits—at least in the case where evaluation times follow a uniform distribution. These results mostly serve to limn with precision the intuitive point that the idle time properties of generational EAs are driven primarily by the maximum evaluation time incurred by a given population. Having analytic results on the uniform distribution is useful in that it allows an easy algebraic rule for predicting what kind of inefficiencies to expect in a given application, and of quantifying the benefits of an ASEA.[25] In practice, Theorem 3.22 will be of fairly narrow use to practitioners, because few applications exhibit uniform evaluation-time distributions in the objective functions. I am hopeful that these results can be extended in the future to provide similar theoretical analysis of other distributions that often arise—such as Gaussian, exponential, and long-tail distributions. In the future, I hope that more analytic solutions (or at least empirical results) can be collected so that organizations and tech companies who use distributed clusters for optimization have an easy way to understand the benefits that asynchronous search algorithms will provide them in their specific context.

The third thread of Research Question 1 that I have investigated is the difficult and application-specific question of true speedup in Hypotheses 3.7–3.12—that is, the impact that asynchronous evolution has on the ability to find acceptable solutions to problems in less "wall-clock time" than would otherwise be possible. The reason this question is

[25]This is exactly why algebraic solutions have been so important around the world since antiquity: they provide a simple, straightforward way to calculate values that otherwise would be very difficult to divine.

not obvious it that while the ASEA does increase an algorithm's ability to make more fitness queries in less time (to refer back to my original motivating diagram for evolutionary algorithm efficiency in Chapter 1—Figure 1.1), it also follows a different kind of feedback loop with different search dynamics than traditional genetic algorithms do. So the question of how much true speedup it offers will depend to some degree on the application, and on the interaction between the fitness landscape and the asynchronous dynamical system at the algorithm's heart. Here, I mostly tested a number of commonsense assumptions that practitioners (including myself) have often had when approaching ASEAs: beliefs like "an ASEA tends to be a greedier algorithm," and "an ASEA will tend to be drawn more quickly toward fast-evaluating regions of the search space." Here I found a number of surprises: the ASEA is indeed greedier on unimodal problems, but overall it proved quite good at avoiding local optima on more complex problems. And it has turned out—to my surprise—that no clear, systematic evaluation-time bias emerged from these experiments. The biggest practical thrust of these results it that there appears to be no glaring "gotcha" that affects the asynchronous steady-state EA: it is a viable optimization strategy that can handle a variety of different kinds of complex optimization problems—and no general cautions against it (such as "don't use it on complex problems when better solutions are slower-evaluating") seem to apply. Personally, these results have greatly emboldened me to recommend the ASEA to my colleagues as a practical solution to applied problems.

3.6.2 Research Question 2: Evaluation-Time Bias

These results led me directly to Research Question 2, in which I have investigated the conditions that give rise to evaluation-time bias and how it might be mitigated. Using flat fitness landscapes (Hypotheses 3.13 and 3.14) and a specially constructed landscape with two configurable basins of attraction (Hypotheses 3.15–3.17), what has surprised me in this study has been the revelation that most evaluation-time bias in ASEAs occurs during asynchronous initialization of the population. Besides that, it seems to be very weak (this is another reason that studying a hybrid, synchronous initialization approach for ASEAs

may be fruitful). The practical impact of this work has been an immediate boost in the confidence with which I apply asynchronous EAs in my work: these results have led me to believe that "evaluation-time bias is no big deal."

This general principle may not generalize to every application, however, and my understanding of evaluation-time bias remains empirical. I am frustrated that we (the EC community) as of yet have no solid analytical understanding of what causes evaluation-time bias: the dynamical feedback system in the ASEA is just complex enough that it is not obvious how to fit it into any of the existing methodologies of Markov chain analysis, drift analysis, or other tools that the EC community has traditionally used to understand algorithm dynamics. I hope that future research can settle this question and give apodictic credence to the conclusions that my experiments here have pointed to. Recent work by Harada [2019] may offer the first step in this direction.

3.6.3 Research Question 3: Excess Computation & SWEET

My final research question was motivated by applications that my colleagues have been engaged in, in which an optimization algorithm tends to encounter more computationally expensive objective evaluations as it narrows in on higher-quality solutions. For example, in the F1TENTH competition,[26], entrants compete to autonomously control a one-tenth scale Formula One car in racing around a track. A common approach to synthesizing autonomous vehicle controllers for this task would be to first implement a simulation in which a controller uses simulated sensors to navigate around a track while, minimally, avoiding crashing into the walls [Scott et al., in press]. But in this example, low-quality solutions will lead to vehicle trajectories that quickly collide with a wall, ending the simulation—whereas higher-quality solutions will be able to successfully navigate for a longer period of time. This variance in evaluation times will create idle time on distributed processors.

An asynchronous EA can reduce the idle time incurred by an evolutionary learner in this situation. But it seems that in such a case, the extra computational resources will serve

[26] `https://f1tenth.org/`

no benefit—the algorithm may simply trade off idle resources for *excess computation*. This is because in order to make progress on optimization, evolution must receive feedback from increasingly high-quality individuals. But if those individuals are always the long-evaluating ones in the population, then generating additional offspring while the algorithm waits for them to complete evaluating may yield little benefit. In a word, the ASEA reduces idle time, but it cannot always speed up the *feedback loop* that drives evolutionary progress.

My results testing Hypotheses 3.18–3.22 shows that the SWEET (Selection WhilE EvaluaTing) strategy is a promising approach to mitigate this pathology—providing an asynchronous algorithm that is able to maintain a tighter feedback loop. An encouraging result that came out of this study is the observation that SWEET seems to have no significant negative impact on applications that are the opposite of what it was design for—reducing the fear that using it might lead to poor results on problems that don't satisfy the simple assumption "longer is always better." In practice, my colleagues have found that the positive benefit of SWEET is fairly small on their applications so far (to the point of being difficult to detect statistically) [Scott et al., 2021, in press]. This suggests to me that the use case for SWEET may be to consider it as one narrow algorithmic tool that can be combined with others to create efficient algorithms for various applications. Having a fully-featured software framework that makes these tools available to practitioners (without require in-depth research to implement myriad small operators and features) is essential for making results of this kind of research useful. My work as a coauthor of the LEAP framework partly aims to help move niche features of these kind to a higher technology readiness level—providing ready-build examples and easily installable Python packages for deploying distributed optimization strategies [Coletti et al., 2020].

Chapter 4: Transferability in Instance-Based Evolutionary Knowledge Transfer

> In a rugged field of this character, selection will easily carry the species to the nearest peak, but there may be innumerable other peaks which are higher but which are separated by 'valleys.' The problem of evolution as I see it is that of a mechanism by which the species may continually find its way from lower to higher peaks in such a field.
>
> —Sewall Wright [1932]

4.1 Research Plan

In this chapter I present a series of results that concern transferability: the conditions under which knowledge transfer between two or more tasks can be exploited. More specifically, the results in this chapter address the first two "ingredients of successful EKT" that I discussed in Chapter 2: transferability within problem classes and source selection. The experiments in this chapter all use simple population-seeding approaches to sequential EKT (I postpone treatment of the other strategies and representations that can be used to Chapter 5).

In section 4.2 I pose and prove no-free-lunch theorems for transfer optimization, showing that in the class of all possible problems (fitness functions) defined over some search space, no EKT approach outperforms any other on average. I follow this up with some empirical experiments on problem classes whose definition involves permutation, but which slightly relax the assumptions of the NFLTs—confirming the prediction that population-seeding EKT is ineffective in these domains. Then in section 4.3 I give examples of narrower problem classes in which knowledge transfer is considerably beneficial—proving a simple asymptotic

complexity result for population seeding on generalized leading-ones problems, for example, and performing experiments with generalized max-ones and leading-one problems whose target patterns vary in a modular way, and with real-valued optimization problems. Together, section 4.2 and section 4.3 test the two parts of Research Question 4.

In section 4.4.1 I present a very simple analysis of how properties of real-valued optimization problem pairs—namely landscape correlation and global optimum distance—correlate with transferability between problem instances. This preliminary analysis suggests that future research on landscape properties may be a promising approach to source selection to mitigate negative transfer in EKT (Research Question 5).

Finally, in section 4.4.2 I study many-source population seeding on real-valued optimization problems, performing an experiment in which population seeding proves effective at doing implicit source-selection—avoiding negative transfer and succeeding at positive transfer when using many source tasks (even though only a subset of the source tasks are useful sources for any given target task). This result satisfies Research Question 6.

4.2 No Free Lunch for Transfer

Because NFLTs are an important theoretical landmark for understanding problem-solving in search and optimization, I begin my investigation of evolutionary knowledge transfer with a formal observation that there can be no "free lunch" with transfer: when averaging over all possible source-target pairs, no optimization algorithm that uses knowledge transfer will perform better than any other. This implies that, in general, no strategy for identifying and transferring bits of information from one task to another is better than any other: any transfer strategy that performs well on one subset of problems performs poorly on another subset of problems.

In section 2.2.4, I explained that the classic no-free-lunch theorems (NFLTs) for optimization are not general enough to apply directly to evolutionary knowledge transfer. In

this section I will analytically prove novel no-free-lunch theorems for optimization covering the special cases of pairwise sequential transfer, multi-task transfer, and heterogeneous sequential transfer (i.e., transfer where the domains of the source and target tasks differ). These results invite the usual interpretation of NFLTs as theorems that are theoretically foundational, but which have little-to-no direct implications for practice: in general, NFLTs only hold over classes of problems that are closed under permutation of the functions that define them [Schumacher et al., 2001, Whitley and Watson, 2005], and these classes of functions are so large as to contain a great many arbitrary functions that have no relationship to real-world problems. At the end of this section, however, I will present some empirical results that show evidence that the no-free-lunch view seems to hold even when the strict assumptions made by NFLTs are relaxed somewhat.

4.2.1 Proof of NFLTs for Transfer Optimization

Traditional proofs of the no-free-lunch theorems for search and optimization proceed by showing that all algorithms (that is, all rational strategies for sampling solutions from a solution space $\mathcal{X}$ and returning the best solution found) perform identically when aggregated over all possible problems $\mathcal{F}$. This is usually done by showing that the sum (and thus average) of an algorithm's performance values across all problems are equal for all algorithms regardless of what performance metric is chosen [Adam et al., 2019, Whitley and Watson, 2005].

Preliminary Lemmas

One clear and effective way to approach this kind of proof is to define a *fundamental matrix* (or *P-matrix*) of performance values [Ho and Pepyne, 2002a,b].

Table 4.1: Illustration of a *fundamental matrix* for single-task optimization in the special case where $Y = \{0, 1\}$ and $X_s = \{x_0, x_1, x_2, x_3\}$.

$\mathbf{f}_s =$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}
$\mathbf{x}_0$	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
$\mathbf{x}_1$	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
$\mathbf{x}_2$	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
$\mathbf{x}_3$	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Definition 4.1. ***(Fundamental Matrix)***. *Let $\mathcal{X}$ be a solution space, and $\mathcal{Y}$ be a set of performance values. We can assume that both sets are finite, since even with floating-point representations all values used by digital computers are discrete. Further, let $\mathcal{F}$ be the set of all possible mappings $f : \mathcal{X} \mapsto \mathcal{Y}$. Now, the* ***fundamental matrix*** *M is constructed by letting the columns correspond to each problem $f_j \in \mathcal{F}$, and the rows correspond to each solution $x_j \in \mathcal{X}$. Specifically, the elements of M are made up of performance values $y_{ij} \in \mathcal{Y}$, chosen such that $M_{ij} = y_{ij} = f_j(x_i)$.*

An example of a fundamental matrix for a binary performance set and a solution space of size $|\mathcal{X}| = 4$ is illustrated in Table 4.1.

The fundamental matrix is a tool for thinking about the degree to which values sampled from $\mathcal{X}$ give an algorithm *information* about which function $f \in \mathcal{F}$ the algorithm is solving. If the algorithm can learn about the function it is solving in a way that allows it to predict the performance value of unseen samples, then it can be more efficient about choosing future samples—and algorithms that are better at this kind of learning can out-perform algorithms that are worse at it. The key conclusion we can draw from a fundamental matrix, however, is that no such learning is possible when considering the space of all possible problems $\mathcal{F}$: learning the performance values associated with a set of samples provides no information about the performance values of the remaining (unseen) samples, and thus does nothing to reduce the algorithm's uncertainty about what function it is solving.

These insights follow analytically from the basic observation that the columns of a fundamental matrix are unique. This implies the two following useful lemmas, which are given by Ho and Pepyne [2002b]:

Lemma 4.1. ***(Counting Lemma for Fundamental Matrices).*** *For a fundamental matrix M associated with some $\mathcal{X}$ and $\mathcal{Y}$, each $y \in \mathcal{Y}$ appears exactly $|\mathcal{Y}|^{|\mathcal{X}|-1}$ times in each row.*

Proof. Briefly, the lemma can be proven by showing that the uniqueness of the columns of M requires that each possible value $y \in \mathcal{Y}$ appears exactly $|\mathcal{Y}|^{|\mathcal{X}|-1}$ times in each row. A full proof along these lines is given by Ho and Pepyne [2002b]. □

Lemma 4.2. ***(Sub-matrix Lemma for Fundamental Matrices).*** *Consider a fundamental matrix M, and pick any row i and any $y \in \mathcal{Y}$. Now create a sub-matrix M' by eliminating from M row i and all columns j such that $M_{ij} \neq y$. M' is itself then fundamental matrix.*

Proof. The idea of the proof is to observe that M' must have $|\mathcal{X}| - 1$ rows and that, by Lemma 4.1, it must have $|\mathcal{Y}|^{|\mathcal{X}|-1}$ columns. Then observing that the columns of M' are unique shows that it is a fundamental matrix. □

These lemmas can be used to execute proofs that all algorithms that solve problems by sampling solutions in $\mathcal{X}$ will have the same performance on average across all problems.

Results

I will proceed to prove an NFL for transfer across any source-target task pair by constructing a fundamental matrix in which each column corresponds to a task pair (f_s, f_t). As a bonus, I will not assume that f_s and f_t are defined over the same domains. This will allow the results here to generalize to *heterogeneous transfer* algorithms.

Following the notation of Wolpert and Macready [1997], let $d_m^x = \{d_m^x(1), \cdots, d_m^x(m)\}$ be the set of distinct points that an algorithm samples from a solution space at each of

m steps, and let $d_m^y = \{d_m^y(1), \cdots, d_m^y(m)\} = \{f(d_m^x(1)), \cdots, f(d_m^x(m))\}$ be the set of values of $\mathcal{Y}$ that are obtained by evaluating those samples. Let $\mathcal{X}_s = \{x_1, \ldots, x_q\}$ and $\mathcal{X}_t = \{x'_1, \ldots, x'_r\}$ be two sets of *solutions* (a.k.a. two *domains*) with finite cardinality, and let $\mathcal{Y} = \{y_1, \ldots, y_s\}$ be a set *performance values*, also with finite cardinality. Then consider the set of all unique mappings $\mathcal{F}_s = \{f_j : \mathcal{X}_s \mapsto \mathcal{Y}\}$ that map a finite solution set $\mathcal{X}_s$ to a finite set of performance values $\mathcal{Y}$. Each of these mappings can be considered a different *problem*, under the criterion that two problems are distinct if and only if they exhibit different performance values $y = f(x)$ for some x. Likewise, define the set of all unique target problems $\mathcal{F}_t = \{f_k : \mathcal{X}_t \mapsto \mathcal{Y}\}$.

I claim that no pairwise (i.e., two-task) knowledge transfer algorithm will perform better than any other on average across all possible source, target pairs $(f_s, f_t) \in \mathcal{F}_s \times \mathcal{F}_t$. This can be established via the following theorem:

Theorem 4.1. ***(No Free Lunch Theorem for Pairwise Transfer Optimization on Discrete Domains)*** *For any pair of algorithms a_1 and a_2,*

$$\sum_{f_s \in \mathcal{F}_s} \sum_{f_t \in \mathcal{F}_t} P(d_m^y | f_s, f_t, m, a_1) = \sum_{f_s \in \mathcal{F}} \sum_{f_t \in \mathcal{F}_t} P(d_m^y | f_s, f_t, m, a_2) \tag{4.1}$$

Proof. We proceed by defining $\mathcal{G} \equiv \mathcal{F}_s \times \mathcal{F}_t$ as the cross product of the source and target problems $\mathcal{F}_s$ and $\mathcal{F}_t$. We interpret the elements $(f_s, f_t) \in \mathcal{G}$ as the enumeration of all possible (source, target) pairs that can be used for pairwise knowledge transfer.

Now, construct a fundamental matrix M for knowledge transfer problems as follows. For every pair (f_s, f_t), create a column of length $|\mathcal{X}_s| + |\mathcal{X}_t|$ of performance values taken from Y, such that the first $|X_s|$ values are equal to $f_s(x_i)$ for each $x_i \in \mathcal{X}_s$, and the remaining $|X_t|$ values are set to $f_t(x'_i)$ for each $x'_i \in \mathcal{X}_t$. The matrix formed by these columns thus has $|\mathcal{X}_s| + |\mathcal{X}_t|$ rows and $|\mathcal{G}| = |\mathcal{F}_s| \cdot |\mathcal{F}_t| = |Y|^{|\mathcal{X}_s|+|\mathcal{X}_t|}$ columns. Note furthermore that because the mappings in each of $\mathcal{F}_s$ and $\mathcal{F}_t$ are unique, the pairs of mappings in $\mathcal{G}$ are also unique,

and thus all columns of the matrix are unique. An example of such a fundamental matrix for two tasks is shown in Table 4.1 for the special case where $Y = \{0, 1\}$, $\mathcal{X}_s = \{x_1, x_2, x_3\}$, and $\mathcal{X}_t = \{x'_1, x'_2, x'_3\}$.

Next consider an optimization algorithm a that proceeds by sequentially selecting m unique samples from the combined search space $\mathcal{X}_s \cup \mathcal{X}_t$. We will prove the theorem by induction on m.

- **Base step**: When the first sample d_1^x is taken, it corresponds to a row i in the fundamental matrix $M = M_1$, and we obtain a performance value $d_1^y = f(d_1^x)$ that corresponds to the column of the function being solved. Because M_1 is a fundamental matrix, by Lemma 4.1 we know that all performance values appear in the row an equal number of times. Therefore, for any value of y, and under the assumption that all problems are equally likely to be the one we are solving,

$$\sum_{f_s, f_t} P(d_1^y = y | f_1, f_2, m = 1, a) = \frac{|\mathcal{G}|}{|\mathcal{Y}|} = |\mathcal{Y}|^{|\mathcal{X}_s|+|\mathcal{X}_t|-1}. \tag{4.2}$$

  The set of unvisited sample points and their possible values, moreover, is now described by a sub-matrix M_2 of M with the ith row removed and all columns j removed where $M_{ij} = d_1^y$. By Lemma 4.2, M_2 is also a fundamental matrix, and it has $|\mathcal{X}_s| + |\mathcal{X}_t| - 1$ rows.

- **Inductive step**: Assume as the hypothesis that for some value of $m \in \mathbb{N} \cup 0$, we have

$$\sum_{f_s, f_t} P(d_m^y = y | f_1, f_2, m, a) = |\mathcal{Y}|^{|\mathcal{X}_s|+|\mathcal{X}_t|-m-1}, \tag{4.3}$$

  and the set of unvisited sample points is described by a fundamental matrix M_{m+1}

Table 4.2: Illustration of a *fundamental matrix* for single-source transfer in the special case where $Y = \{0, 1\}$, $X_s = \{x_1, x_2, x_3\}$, and $X_t = \{x'_1, x'_2, x'_3\}$. The upper half of the matrix's rows represent search points that can be sampled on the source task. The lower half are search points that can be sampled on the target task. Because the columns enumerate all possible joint mappings from $X_s \cup X_t$ to Y, the mean of all the rows are equal by Lemma 4.1. This observation is a key step in proving the no free lunch theorem for single-source transfer.

$\mathbf{f}_s =$	f_1	f_1	f_1	f_1	f_1	f_1	f_1	f_1	$\cdots$	f_8	f_8	f_8	f_8	f_8	f_8	f_8	f_8
$\mathbf{f}_t =$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	$\cdots$	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
$\mathbf{x}_0$	0	0	0	0	0	0	0	0	$\cdots$	1	1	1	1	1	1	1	1
$\mathbf{x}_1$	0	0	0	0	0	0	0	0	$\cdots$	1	1	1	1	1	1	1	1
$\mathbf{x}_2$	0	0	0	0	0	0	0	0	$\cdots$	1	1	1	1	1	1	1	1
$\mathbf{x}'_0$	0	0	0	0	1	1	1	1	$\cdots$	0	0	0	0	1	1	1	1
$\mathbf{x}'_1$	0	0	1	1	0	0	1	1	$\cdots$	0	0	1	1	0	0	1	1
$\mathbf{x}'_2$	0	1	0	1	0	1	0	1	$\cdots$	0	1	0	1	0	1	0	1

with $|\mathcal{X}_s| + |\mathcal{X}_t| - m - 1$ rows. Then we have:

$$\sum_{f_s, f_t} P(d_m^y = y | f_1, f_2, m+1, a) = |\mathcal{Y}|^{|\mathcal{X}_s|+|\mathcal{X}_t|-(m+1)-1}. \tag{4.4}$$

Therefore, by induction, Equation 4.4 holds for any value of m. But this means that the value of the sum does not depend on the algorithm a, and the theorem follows. □

Corollary 4.1. *No knowledge transfer algorithm outperforms single-task optimization when averaged over all possible problems.*

Proof. Single-task optimization can be seen as a knowledge transfer algorithm that simply ignores all information from the source task. Fixing a_1 to any single-task algorithm in Theorem 4.1 proves the corollary. □

Corollary 4.2. *There is no free lunch for* ***pairwise multi-task optimization****.*

Proof. The proof of Theorem 4.1 makes no assumption about the order in which samples are drawn from the two domains. It thus includes multi-task algorithms—that is, algorithms that alternate arbitrarily between taking samples from X_s and X_t—and the corollary follows. □

Corollary 4.3. *There is no free lunch for pairwise sequential transfer (or multi-task learning) on* ***heterogeneous discrete domains****.*

Proof. Theorem 4.1 makes no assumption that the two domains X_s and X_t are equal. If we set $X_s \neq X_t$, then the corollary follows. □

4.2.2 Experiments on Permuted Max-Ones and Multi-Peaks Problem Classes

In section 2.2.5, I discussed how it is important that a class of problems that EKT is applied to displays some non-negligible probability of positive transfer occurring between an arbitrary pair of tasks sampled from it. This can be important both to make it easier for a human to select useful source tasks *a priori*, and/or to make it easier for many-source EKT algorithms to sift useful material out from non-useful material (thus avoiding the memory swamping problem).

In this section, I apply a population-seeding EKT algorithm to pairs of tasks that are randomly sampled from two different classes of pseudo-Boolean optimization problem—namely a permuted version of the max-ones problem and a set of "multi-peak" problems defined by Watson and Jansen [2007]. These problem classes are defined by problem generators that make liberal use of permutation and/or random-sampling procedures as part of their operation, yielding very large and fairly unrestricted problem spaces.

These problem classes are not so large as to include "all possible problems," so they do not satisfy the assumptions of the NFLTs that I presented above (nor are they fully closed under permutation of their fitness values—so sharpened NFLTs in the vein of Schumacher et al. [2001] also would not apply). I show nevertheless that positive transferability is extremely

rare in these problem classes when population seeding is used. These results serve as a lightweight test of how to interpret the NFLTs: if the probability of transferability grows rapidly as I relax the no-free-lunch assumptions, then EKT would be very easy to exploit. If not, however, then this would be evidence for the claim that specialized knowledge-transfer algorithms (or knowledge transfer methods) are needed for different problem classes—or perhaps that knowledge transfer is not a viable approach at all for some problem classes.

Hypotheses

The point of the experiments in this section is to demonstrate the limitations of evolutionary knowledge transfer. To do so, I study the "permuted max-ones" and "multi-peaks" problem classes (each of which I will define below). Both of these classes are fairly unrestricted, and functions sampled from them have a tendency to be rugged with many local optima.

From an intuitive perspective, population-based seeding is likely to be useful when a solution transferred from a source task is located in a part of the search space that is either close to a high-quality optimum on the target task, or located on a gradient that makes a high-quality optimum easy for evolution to find. Rugged functions frustrate transfer of this kind in at least two ways. First, when optima are located arbitrarily in the space, the probability that a given source task has a global or high-quality optimum that shares a basin of attraction with a similar optimum on the target landscape may be very low. Second, even if two functions share similar global optima, the ruggedness of the source function makes it difficult to find the global optimum—and different runs of evolution on the same source task may arrive at very different solutions (and thus transferred individuals).

For these reasons, instance-based EKT on these complex fitness landscapes may only be useful in rare cases:

Hypothesis 4.1. *Instance-based transfer will rarely be useful on the permuted max-ones problem class.*

Hypothesis 4.2. *Instance-based transfer will rarely be useful on the multi-peaks problem class.*

Methods

The *max-ones* problem is the best-studied benchmark function in the theory of evolutionary computation and randomized search heuristics [Doerr, 2020]. This simple Boolean optimization problem is defined by a function that assigns to each genotype $\mathbf{x} \in \mathbb{B}^n$ an integer equal to the sum of bits in the genotype with a value of 1 (as opposed to 0):

$$f(\mathbf{x}) = \sum_{i=1}^{n} [x_i = 1], \tag{4.5}$$

where the Iverson bracket $[x_i = 1]$ is equal to 1 if $x_i = 1$ and 0 otherwise. The max-ones problem is very simple as far as optimization problems go, because the fitness contribution of each gene is completely independent of the values of all of the other genes: the resulting function is unimodal, and it is well-known that the computational complexity of a simple $(1 + 1)$-style EA on max-ones (and linear problems more generally [Droste et al., 2002]) is $O(n \log n)$ [Mühlenbein, 1992].[1]

The *permuted max-ones* problem generator I introduce here creates complex, multimodal optimization problems by applying a permutation to the value of the sum expression in equation 4.6:

$$f_\phi(\mathbf{x}) = \phi \left(\sum_{i=1}^{n} [x_i = 1] \right). \tag{4.6}$$

That is, while in the original max-ones a bitstring such as 001101 would receive a fitness value of $f(001101) = 3$, in the permuted version the value 3 is mapped via the permutation $\phi(\cdot)$ to a new value, so that $f_\phi(001101) = \phi(3)$. Examples of randomly permuted max-ones

[1] In fact, this was probably the first formal runtime result to be proved in evolutionary computation, according to Jansen [2020].

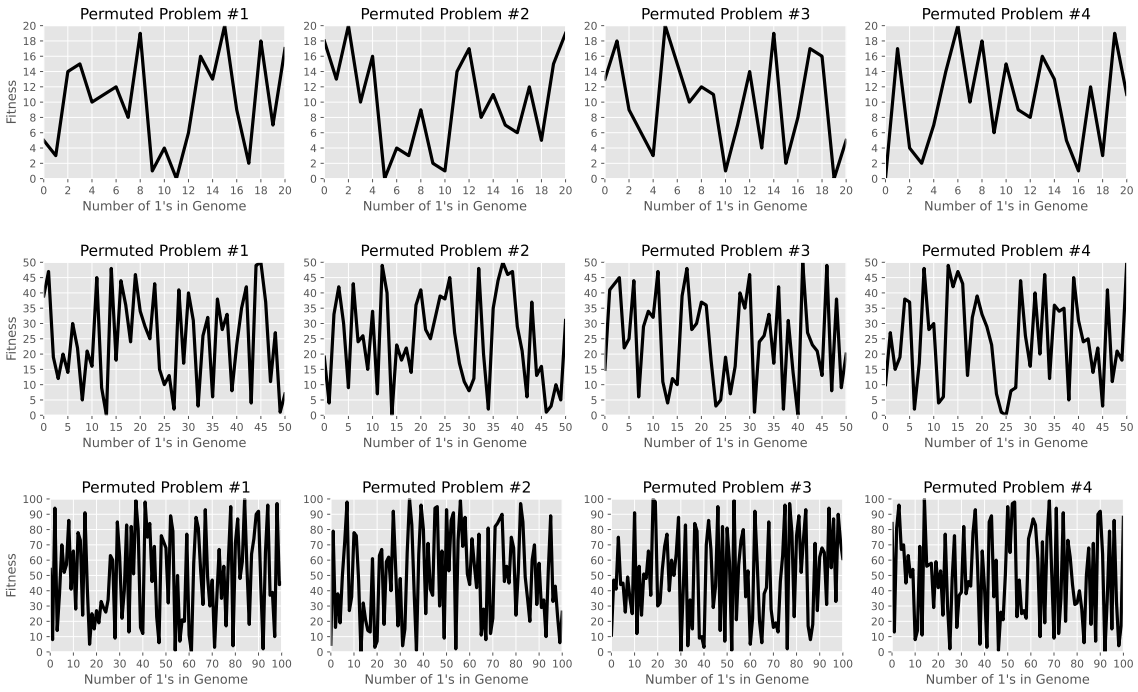

Figure 4.1: Examples of permuted max-ones problems that were generated with 20 dimensions (top), 50 dimensions (middle), and 100 dimensions (bottom).

problems that are generated in this way are visualized in Figure 4.1 for bitstrings of 20, 50, and 100 dimensions. In these visuals, the x-axis corresponds to the number of ones in the genotype (i.e., $f(\mathbf{x})$), and the y-axis plots a permuted version of the function (according to some randomly selected permutation ϕ).

The second problem generator I use in this section is the *multi-peak* generator. Watson and Jansen [2007] introduced this class of functions in the context of studies on evolutionary crossover (as multi-peak functions can be combined to build more complex functions that are asymptotically easier to solve with crossover than they are without). A multi-peak problem is made up of a number of binary *target patterns* $\{\mathbf{t}_1, \mathbf{t}_2, \ldots, \mathbf{t}_T\}$, each surrounded by a basin of attraction whose fitness decreases as the genotype vector's Hamming distance from the

target pattern increases. Specifically, the multi-peak function is defined as follows:

$$f_{mp}(\mathbf{x}) = \sum_{j=1}^{T} c(\mathbf{x}, t_j), \tag{4.7}$$

where the individual terms are defined by

$$c(\mathbf{x}, \mathbf{t}_j) = \begin{cases} w_j & \text{if } |\mathbf{x} - \mathbf{t}_j| = 0, \\ (1 + |\mathbf{x} - \mathbf{t}_j|)^{-1} & \text{otherwise.} \end{cases} \tag{4.8}$$

Here, w_j is the fitness contribution made by each target pattern $\mathbf{t}_j$ when the genome $\mathbf{x}$ matches that target pattern exactly.

I use Equation 4.7 to implement a multi-peaks problem generator as follows. I assume the genome dimensionality $|\mathbf{x}| = 20$, and I initialize $T = 4$ target strings by randomly generating 20-dimensional binary vectors, with the first m values set to one and the rest set to zero, where m is a random integer between 0 and $|\mathbf{x}|$ (inclusive). Finally, the w_j values are sampled uniformly on $(0, 2)$ (as in Watson and Jansen [2007]). Figure 4.2 shows a 1-dimensional cross-section of the fitness landscape that results from some of the generated functions. It is evident that these multi-peaks problem have fewer local optima than the most of the permuted max-ones problems do, each with a relatively wide basin of attraction.

On both of these problem classes, I run a single-task EA and a population-seeding EKT algorithm. Both algorithms use a bitstring representation for genotypes, and apply standard bit-flip mutation (that is, mutation in which each gene in a genotype is mutated with some probability) with a probability of $p = 1/L$, where L is the length of the genome. Neither algorithm uses crossover. The EKT algorithm here is run first on a source task and then on a target task. It saves the highest-fitness individual that was seen during the target task, and uses it to replace a single individual in the initial population when solving the target

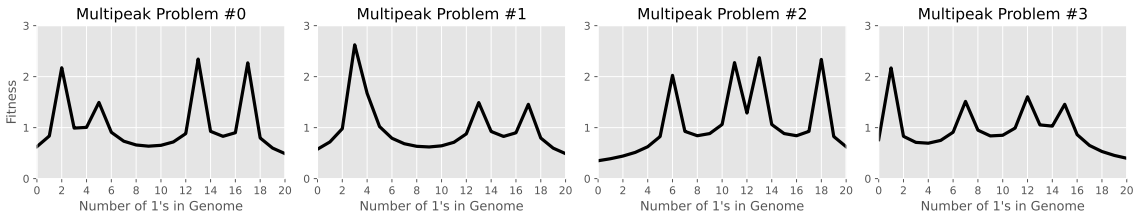

Figure 4.2: Fitness landscape cross-sections of problems generated with the form of Equation 4.7. At each point along the x axis, I fed a genome into the function with the indicated number of leading ones, and the remaining genes fixed to 0.

Table 4.3: Parameters used for the control and population-seeding algorithms on the permuted-max-ones and multi-peaks problems.

Parameter	Permuted Max-Ones	Multi-Peaks
Genome length	$L \in \{20, 50, 100\}$	$L = 20$
Parent selection	Binary tournament	
Mutation type	Flip each gene with probability p	
Mutation probability	$p = 1/L$	
Crossover probability	0 (no crossover)	
Population size	$\mu = 10$	
Generations for the single-task control	100	200
Generations for transfer during the training phase	100	200
Generations for transfer during the testing phase	100	200

task (the remaining $\mu - 1$ individuals in the initial population are uniformly initialized as they would be in a single-task algorithm). The single-task algorithm, meanwhile, is only run on a target task. The configuration and parameters used on both problem classes are summarized in Table 4.3.

On both problem classes, I generate 1,000 target tasks $\mathcal{T}$ up front, each with L dimensions. For each target task t, I run the single-task control algorithm for 100 generations, and measure the fitness of the best solution found $\mathbf{x}_c^*(t)$. Then for each target task t, I generate an additional source task s, and run the EKT algorithm, training on s and testing on t,

again measuring the fitness of the best solution found on the target task $\mathbf{x}_T^*(t)$. The experiment's output, then, is a dataset of paired fitness observations $(\mathbf{x}_c^*(t), \mathbf{x}_T^*(t))$, with one pair for each of the 1,000 target tasks. This data gives us a statistical view of how performance on these tasks with transfer differs from the single-task control—aggregated over a sample taken from the problem class.

On the permuted-max-ones problems, I run three separate experiments, each for problems of a different dimensionality ($L = 20$, $L = 50$, and $L = 100$). On the multi-peaks problems, I run just one experiment on multi-peaks with $L = 20$.

Results

For the permuted-max-ones experiments, the mean values of the control results $\mathbf{x}_c^*(t)$ and the transfer results $\mathbf{x}_T^*(t)$, respectively, are shown as a bar plot in Figure 4.3 with standard deviation bars in the left-hand plots and 95% confidence intervals on the right-hand plots. It is clear from the latter that the difference between the mean control and transfer experiments on these benchmarks is not statistically significant. Histograms of the paired *distances* between the two algorithms on permuted max-ones are also shown in Figure 4.4. From these it is clear that positive (and negative) transfer events are very rare in this data: the overwhelming majority of experimental runs show negligible difference in the performance of the two algorithms. Overall, there is no evidence that transfer helps (or hinders) on permuted max-ones problems on average when aggregating over a large number of source-task pairs—**confirming Hypothesis 4.1**.

On the multi-peak problem class, similar bar-plots showing the mean values of the control results $\mathbf{x}_c^*(t)$ and the transfer results $\mathbf{x}_T^*(t)$, respectively, are shown in Figure 4.5. Once again there is no significant trend toward positive transfer, **confirming Hypothesis 4.2**. On this problem class (unlike on the permuted max-ones problems), however, it seems that transfer hinders more often than it helps on average, albeit by a small margin. Some more detailed histograms of this data are shown in Figure 4.6, where the distribution of results for the single-task control slightly outperforms the one for transfer, and "peeks out" from behind the

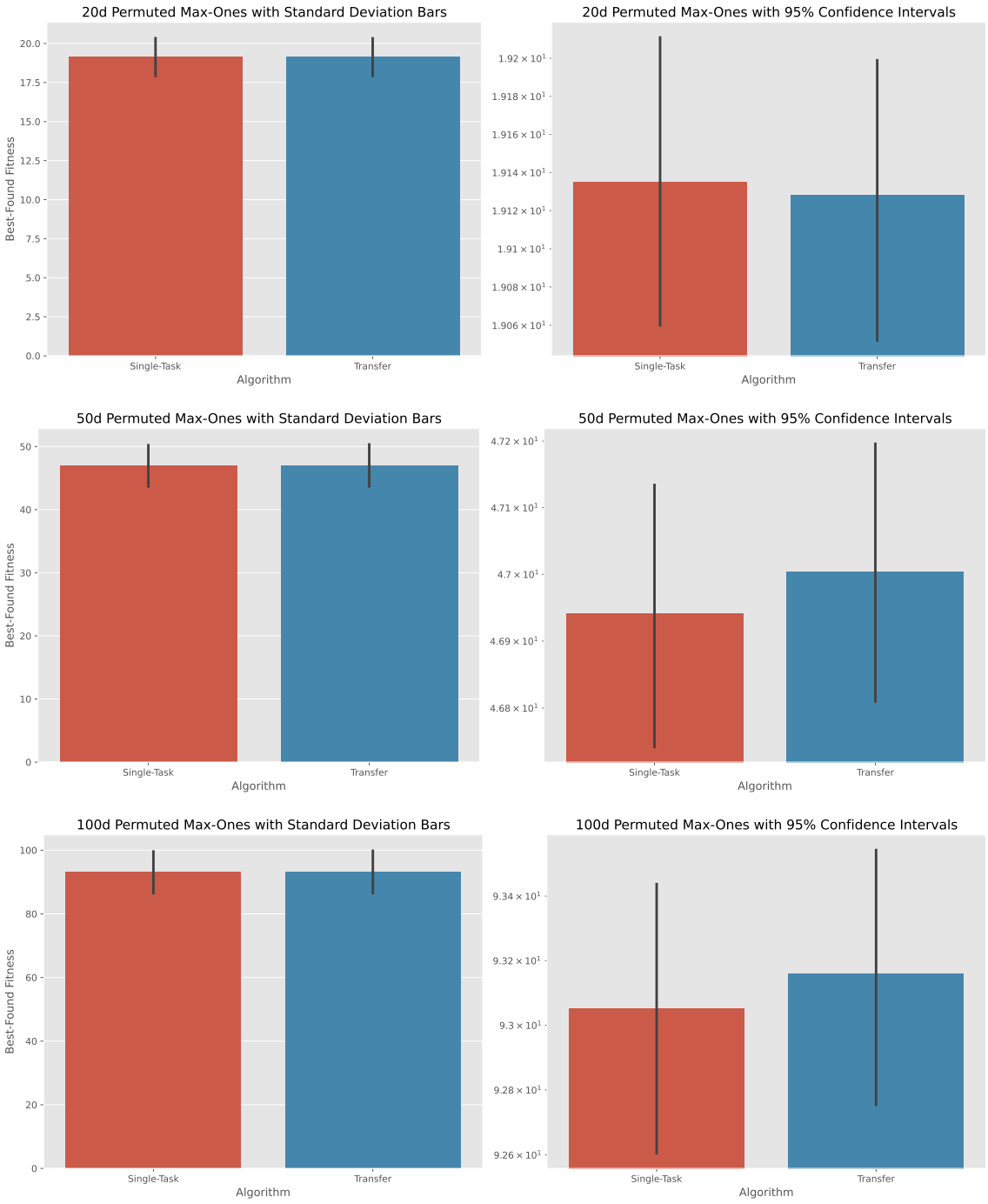

Figure 4.3: Mean best-found fitness results for a single-task control (**red**) and transfer from randomly generated source tasks (**blue**). The right-hand plots are zoomed-in versions of the left-hand plots. Shown are results for 20-dimensional tasks (top), 50-dimensional tasks (middle), and 100-dimensional tasks (right). Standard deviation is shown in the plots on the left, while 95% confidence intervals on the mean are shown on the right (and zoomed in to show small differences).

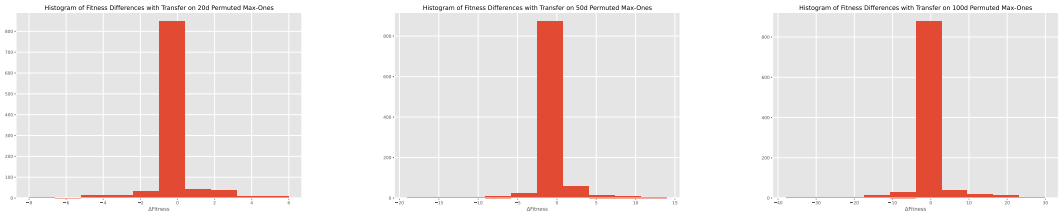

Figure 4.4: Histograms of fitness differences between transfer and the control on different permuted max-ones problems. The difference is computed as $\Delta f = t - c$, where t is the best solution found with transfer from a randomly generated source task, and c is the best solution found with a single-task control. In most cases, transfer makes no difference. In some rare cases, transfer works better or worse.

latter. A Wilcoxon signed-rank test [Wilcoxon, 1945] rejects the hypothesis that difference $\Delta f = t - c$ between the best transfer solution t and best control solution c is symmetric around zero ($p = 8.05 \cdot 10^{-12}$)—further indicating that the observed effect is statistically significant.

Conclusion on Permuted Max-Ones and Multi-Peaks

In this section I have tested two simple classes of problems for "transferability," in the sense of measuring the probability that any two randomly chosen instances of these classes are able to display a positive transfer relationship.

On the surface, it is reasonable to expect that even with the complex, randomized landscapes that are generated by the permuted max-ones and multi-peaks generators, positive transferability might occur with some non-negligible potential. A population-seeding algorithm, moreover, is not strongly bound to rely on the information that it receives from transfer: if the transferred individuals are not useful early in the run, they will simply be discarded by selection and search will focus on the lineages that descent from other, randomly sampled individuals. For that reason, I did not expect to see any significant evidence of negative transfer occurring in the average on these problem classes.

In fact, however, I find that there is no statistically significant mean trend toward positive

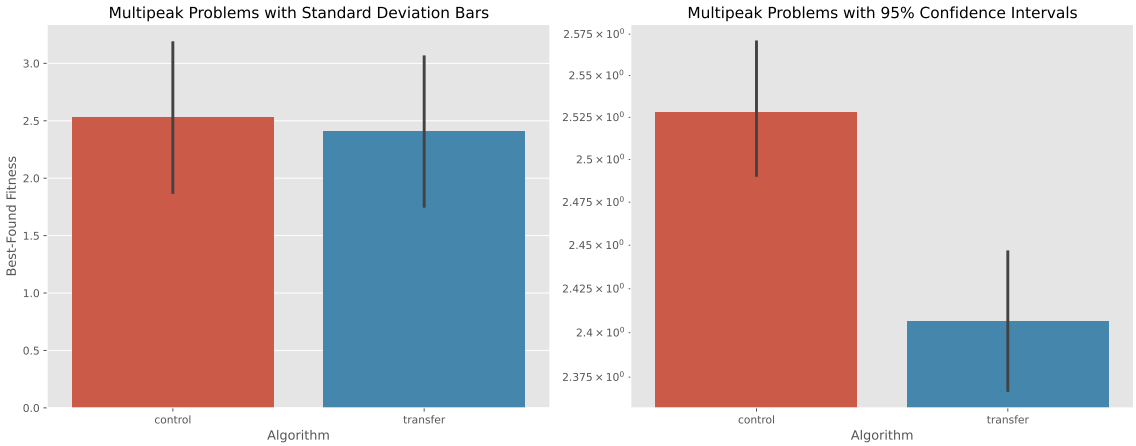

Figure 4.5: Mean best-found fitness results on generated **multi-peak tasks** for a single-task control (**red**) and transfer from randomly generated source tasks (**blue**). Standard deviation is shown in the plot on the left, while 95% confidence intervals on the mean are shown on the right (and zoomed in to show small differences).

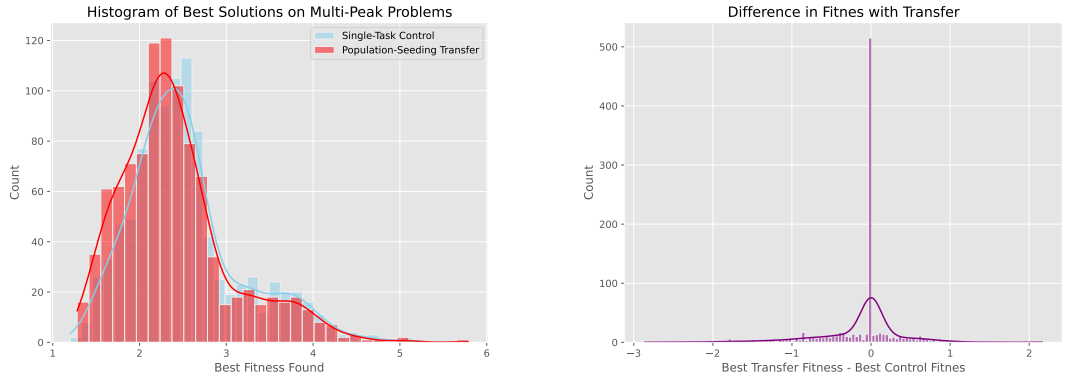

Figure 4.6: Histogram of best fitnesses (**left**) and the difference in best fitness between transfer and the control (**right**) on **multi-peak tasks**. The difference is computed as $\Delta f = t - c$, where t is the best solution found with transfer from a randomly generated source task, and c is the best solution found with a single-task control. In most cases, transfer makes no difference. In some rare cases, transfer works better or worse.

transfer in either problem class (as predicted by Hypotheses 4.1 and 4.2), and furthermore that positive transfer is extremely rare in both cases. To the contrary, on the multi-peaks class, I instead find a significant (if small) mean trend toward negative transfer. Considering how population-seeding transfer works, this negative effect is likely caused by a loss of useful diversity early in the run: in the single-task algorithm, $\mu = 10$ initial random individuals are available to the algorithm as a diverse initial sample of the search space. In the transfer algorithm, however, only $\mu - 1 = 9$ random individuals are available. If the remaining transferred individual from the source task is neither useful (in the sense of carrying information helpful for solving the target task in particular) nor randomly distributed in the solution space, then the effect is that the algorithm is less explorative overall. I believe this explains the slight negative impact on performance that I observe on the multi-peaks problems—which, behind highly multi-modal, require a considerable amount of exploration in order to find high-quality solutions.

Overall I have suggested that these results support the no-free-lunch theorems, and that they suggest that the proposition that there is "no free lunch for transfer" does hold to some degree in complex problem classes as the assumptions of the NFL theorems are relaxed. This argument is informal, however. At the very least, the results here suggest that *designing benchmarks for evolutionary transfer algorithms* is non-trivial: in order to demonstrate and test the performance of EKT methods across large numbers of tasks, problem classes are needed that exhibit stronger similarities on average than can be obtained from simple approaches to generating randomized problem instances. This is the primary lesson that I have drawn from my experience applying EKT algorithms to various randomized problem classes (some of which I have presented here, some of which I have not published).

4.2.3 Discussion of Free Lunches

I have proved some NFLTs for knowledge transfer algorithms that solve optimization problems, and shown some simple experimental settings that seem to confirm the idea that knowledge transfer does not yield free lunches.

Whitley and Watson [2005] suggest that there are "two general reactions" that practitioners tend to have when interpreting NFLTs. The first is to take the NFL principle to imply that problem- or domain-specific knowledge is essential to specializing algorithms so that they can be effective in solving real-world problem classes. In this view, the NFLTs bolster the narrative (which I discussed in section 2.2.2—Newell's "cliche of AI" [Newell, 1982]) that heuristic algorithms should be rigorously combined with expert insight in an effort to match algorithms to problems in specific ways. The second reaction, however, is to emphasize the very strong assumptions that NFLTs require in order to hold (namely averaging over the space of all functions—most of which are in fact random and incompressible). McDermott [2020], for example, has recently provided an extensive argument that the NFLTs are routinely and widely misunderstood by practitioners as having stronger implications than they actually do, and that in many cases NFLTs can be ignored for practical purposes because many general-purpose algorithms are "already specialized to an appropriate subset of problems, potentially escaping NFL."

This interpretive difficulty applies equally to the specific NFLTs that I have presented here. Personally, I take the results of this section as a caution that knowledge transfer by itself is not guaranteed to be a magic bullet—in the sense of offering us a way to systematically circumvent the limitations of single-task algorithms. But, following the arguments collected by McDermott [2020], this need not imply that contemporary efforts to design EKT algorithms that perform better than others at isolating, representing, and transferring useful knowledge are misguided.

4.3 Transferability in Instance-Based EKT

In the previous section I covered the pessimistic side of knowledge transfer, making clear that information reuse can only be useful on optimization problems when similar tasks are readily available that can be used as sources. Now I turn to examples of more constrained problem classes where positive transfer is common and relatively easy to exploit.

In this section I will first prove analytical conditions under which the leading-ones pseudo-Boolean optimization problem becomes asymptotically easier to solve with the help of instance-based transfer. Then I present experiments on three sets of benchmark problems—two of which are based on modularly varying binary patterns, and one of which is based on real-valued optimization problems that commonly appear in the multi-task optimization literature. In all three cases, I show that positive transfer occurs fairly frequently.

4.3.1 Asymptotic Complexity of Leading Ones with Transfer

Population-seeding is the simplest kind of instance-based knowledge transfer, and its benefits can arguably be understood via a simple intuition: if the transferred solution (i.e., the optimum) of the source task is "close to" to a high-quality (or optimal) solution to the target task (or, more specifically, if the latter is easily reachable from the former via successive mutations), then positive transfer is likely to occur. If the transferred solution (or optimum of the source task) is effectively randomly located in the search space, however (or otherwise not related to the target task in a useful way), then transfer is likely to have no effect or a negative one. In some cases the story may be more complicated than this (for example, population-seeding may provide a partial solution that, when combined with other solutions via crossover, allows a new basin of attraction to be reached), but in many applications this simple story is sufficient to explain the benefits of rudimentary transfer strategies—and in any event, the principle that random knowledge sources lead to little benefit still applies.

In this subsection I prove that on the class of generalized leading-ones problems, random local search (RLS) with transfer from a random leading-ones problem to a target task has no effect on the asymptotic complexity of leading-ones: it remains $O(n^2)$ with or without transfer. If we require, however, that the source and target task meet a certain similarity criterion—specifically, if the global optimum of the source task is within a particular distance bound from the target task—then I show that the asymptotic complexity of leading-ones can be reduced to $O(n \log n)$. The intuition behind the proof is straightforward. If an algorithm

requires n steps of improvement to solve a leading-ones instance, but I transfer an individual that has already solved $n - \log n$ of them, then only $\log n$ steps remain. On leading-ones with RLS, each iteration has a $1/n$ chance of improvement, so we thus go from $O(n^2)$ down to $O(n \log n)$ with the help of transfer.

This sketch of the idea is not rigorous, however, because it overlooks some complex aspects of the problem behavior—namely the fact that while RLS does achieve an improvement with probability $1/n$ on leading-ones, the change in fitness that results from one successful mutation can sometimes be greater than one (because a flipped bit may connect a "run" of leading-ones). So it is worth pushing through details like this to provide a formal argument about the conditions that lead to positive transfer. Here I will formalize this results with some tools from the complexity analysis of evolutionary algorithms, and in particular the additive drift theorem [Lengler, 2020], which makes it easy to make arguments of this kind in a rigorous way.

Preliminary Lemma

The *leading-ones* problem is, like max-ones, one of the most heavily studied problems in the theory of evolutionary computation. In its generalized form Doerr [2020], the leading-ones function counts the number of consecutive values, starting from the beginning of a bitstring $\mathbf{x}$, that match the values at the same position of a reference string $\mathbf{z}$:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \prod_{j=1}^{i} [x_i = z_i]. \tag{4.9}$$

A single non-matching bit in the bitstring, then, neutralizes the fitness contribution of any matches that occur later in the string. This dependency among variables makes leading-ones more challenging to solve than one-max—as there is only one bit (and exactly one bit) that can be flipped to yield a fitness improvement at any given step in optimization.

Random local search (RLS) is a very simple $(\mu + \lambda)$-style search algorithm that uses

a single search point ($\mu = 1$), generates a single offspring by mutating exactly one gene ($\lambda = 1$), and keeps whichever is better. RLS differs from most evolutionary algorithms in its mutation operator, but is easier to study analytically as a result. On an n-dimensional leading-ones problem, the additive drift theorem[2] can be used to show that the expected number of steps $\mathbb{E}[T]$ needed to find the global optimum is bounded as follows:

Lemma 4.3. ***(RLS Bounds on Leading-Ones).*** *When random local search is run on leading-ones, its expected running time is bounded by*

$$\frac{n\mathbb{E}[X_0]}{2} \leq \mathbb{E}[T] \leq n\mathbb{E}[X_0], \tag{4.10}$$

where X_0 is a potential function *defined over the initial population—specifically, $X_0 = n - f(x_0)$, where $f(x_0)$ is the fitness of the initial solution.*

Proof. The idea of the proof is to recognize that a single-bit mutation only improves fitness on leading-ones if exactly the next bit after the current sequence of leading ones is flipped. Such a mutation may increase fitness by more than 1, however, if the flipped bit "connects" the leading ones to a sequence of ones that appear later in the genome. This argument leads to an expression for drift that contains a finite geometric series, which can be bounded via an infinite series to obtain δ values of $\frac{1}{n} \leq \Delta_t \leq \frac{2}{n}$. See Lengler [2020] for a full proof. □

In a single-task setting, where the initial solution is randomly initialized on leading-ones, $n - 1 \leq \mathbb{E}[X_0] \leq n$,[3] leading to the well-known asymptotic bound of $\Theta(n^2)$ for RLS on

[2] A basic result in the area of drift analysis, one of the leading mathematical approaches to EA complexity analysis today. The additive drift theorem states that if X_t is a function defined over an algorithm's state such that $X_t \geq 0$ and $X_t = 0$ when the global optimum is first reached, then

$$\Delta_t = \mathbb{E}[X_t - X_{t+1} | X_t \neq 0] \geq \delta \Rightarrow \mathbb{E}[T] \leq \frac{\mathbb{E}[X_0]}{\delta},$$

and

$$\Delta_t = \mathbb{E}[X_t - X_{t+1} | X_t \neq 0] \leq \delta \Rightarrow \mathbb{E}[T] \geq \frac{\mathbb{E}[X_0]}{\delta}.$$

That is, if we can bound the one-step change in potential—the *drift* term Δ_t—on either side by a constant δ for all $t \neq 0$, then bounds on the expected running time directly follow.

[3] Because $0 \leq \mathbb{E}[f(x_0)] \leq 1$. This can be shown on leading-ones with the tail-sum formula for probability, which leads to a finite geometric series.

leading-ones.

Results

Now consider an RLS algorithm that uses pairwise sequential transfer to initialize the first search point x_0 based on the best-found solution to some source task. Depending on the relationship between the source and target task, transfer may have a neutral or positive effect, as I show in the following theorem:

Theorem 4.2. ***(Population-Seeding Transfer on Leading-Ones)***. *Let random local search be run on leading-ones (denoted by $f(\cdot)$) with the initial search point $x_0 = x_s^*$ seeded with the solution of some source task s.*

a) ***Neutral Transfer****: if x_s^* is uniformly distributed on $\mathbb{B}^n$, then $\mathbb{E}[T] \in \Theta(n^2)$.*

b) ***Positive Transfer****: if $\mathbb{E}[f(x_s^*)] \geq n - \log n$, then $\mathbb{E}[T] \in O(n \log n)$.*

That is, if the global optimum of the source task is randomly located in the search space, then we expect no performance improvement on the target task. But if the global optimum of the source task shares at least the first $(n - \log n)$ leading-one bits with the target task, then we will see an asymptotic improvement in performance on leading-ones.

Proof. Part (a): If the source task is completely random (for example, perhaps the source task is drawn uniformly from the set of all possible n-dimensional pseudo-Boolean functions), then the expected value of $x_0 = x_s^*$ is uniform. But in this case, $\mathbb{E}[X_0]$ is the same as in the single task case (namely $n - 1 \leq \mathbb{E}[X_0] \leq n$), and by Lemma 4.3 we have

$$\frac{n(n-1)}{2} \leq \mathbb{E}[T] \leq n^2, \tag{4.11}$$

and therefore $\mathbb{E}[T] \in \Theta(n^2)$.

Part (b): Recall that the potential X_0 is defined as $n - \mathbb{E}[f(x_0)]$. So given that $\mathbb{E}[f(x_s^*)] \geq$

$n - \log n$, we have $\mathbb{E}[X_0] = n - \mathbb{E}[f(x_0)] \leq \cancel{n} - (\cancel{n} - \log n) = \log n$. Then by Lemma 4.3,

$$\mathbb{E}[T] \leq n \log n, \tag{4.12}$$

and $\mathbb{E}[T] \in O(n \log n)$. □

4.3.2 Population Seeding on Modularly Varying Goals

In this section I demonstrate two simple benchmark problem classes where positive knowledge transfer is possible with high probability. These classes are constructed use the concept of *modularly varying goals* (MVGs), which was introduced by Kashtan et al. [2007] and Parter et al. [2008] as part of their simulated efforts to understand how biological evolution and developmental processes can reuse knowledge from past experience to accelerate evolution in new environments. I devise MVG benchmarks that are built atop the dynamics of one-max and leading-ones problems, respectively.

Hypotheses

Instance-based knowledge transfer on one-max-style problems is straightforward to interpret: I expect population-seeding transfer to be beneficial whenever a transferred solution has fitness high enough that it contributes more to solving the target task than random initial individuals will. Because one-max problems have binary genes and the fitness contribution of each gene is independent, the average value $\mathbb{E}[f(x_0)]$ of a random individual's fitness will be $1/2$ the length of the bitstring.

With this in mind, it seems that transfer will be beneficial on one-max-style problems when the overlap in the target patterns for the source and target tasks is greater than 50%.

Hypothesis 4.3. *On modularly varying max-ones, positive transfer will occur in the average on all source-target task pairs where target patterns share at least 50% overlap.*

On leading-ones problems, by contrast, the average initial fitness $\mathbb{E}[f(x_0)]$ of random

individuals is less than or equal to 1.[4] This suggests that transfer will happen frequently on leading-ones-style problems, so long as the source and target task goal patterns overlap in at least the *first two values*. In general, however, transfer in this context will only be beneficial to the extent that the two patterns exactly match for a consecutive run of values beginning from the beginning of the string. This will only occur when two tasks have a high degree of similarity.

Hypothesis 4.4. *On modularly varying leading-ones, a positive transfer will occur proportionally to the initial overlap in target patterns.*

Methods

The two pseudo-Boolean MVG benchmarks I use in these experiments are based on a set 16 binary target patterns which are used to define two sets of pseudo-Boolean functions—shown in Figure 4.7. These specific patterns are borrowed from Watson and Szathmáry [2016], who used them to demonstrate the ability of an adaptive developmental encoding of genotypes to learn to generate compositional patterns. In my version here, each pattern is made up of 400 individual pixels.

I then construct a *max-ones MVG* benchmark by treating each pattern $\mathbf{z}$ as the target bitstring of a generalized max-ones problem. That is, I construct 16 benchmark functions,

[4] Because the probability of an initial string of 1's being generated during initialization is given by the distribution $P(f(x_0) \geq 1) = 1/2, P(f(x_0) \geq 2) = (1/2)^2, P(f(x_0) \geq 3) = (1/2)^3, \ldots,$ this leads to a geometric tail-sum expression for the expected initial fitness:

$$\mathbb{E}[f(x_0)] = \sum_{k=1}^{n} P(f(x_0) \geq k) \tag{4.13}$$

$$= \sum_{k=1}^{n} \left(\frac{1}{2}\right)^k = -1 + \sum_{k=0}^{n} \left(\frac{1}{2}\right)^k \tag{4.14}$$

$$= -1 + \frac{1 - \left(\frac{1}{2}\right)^{n+1}}{1 - \frac{1}{2}} = -1 + 2(1 - 2^{-n-1}) = 1 - 2^{-n}. \tag{4.15}$$

But since n is a positive integer, we have

$$0 \leq \mathbb{E}[f(x_0)] \leq 1. \tag{4.16}$$

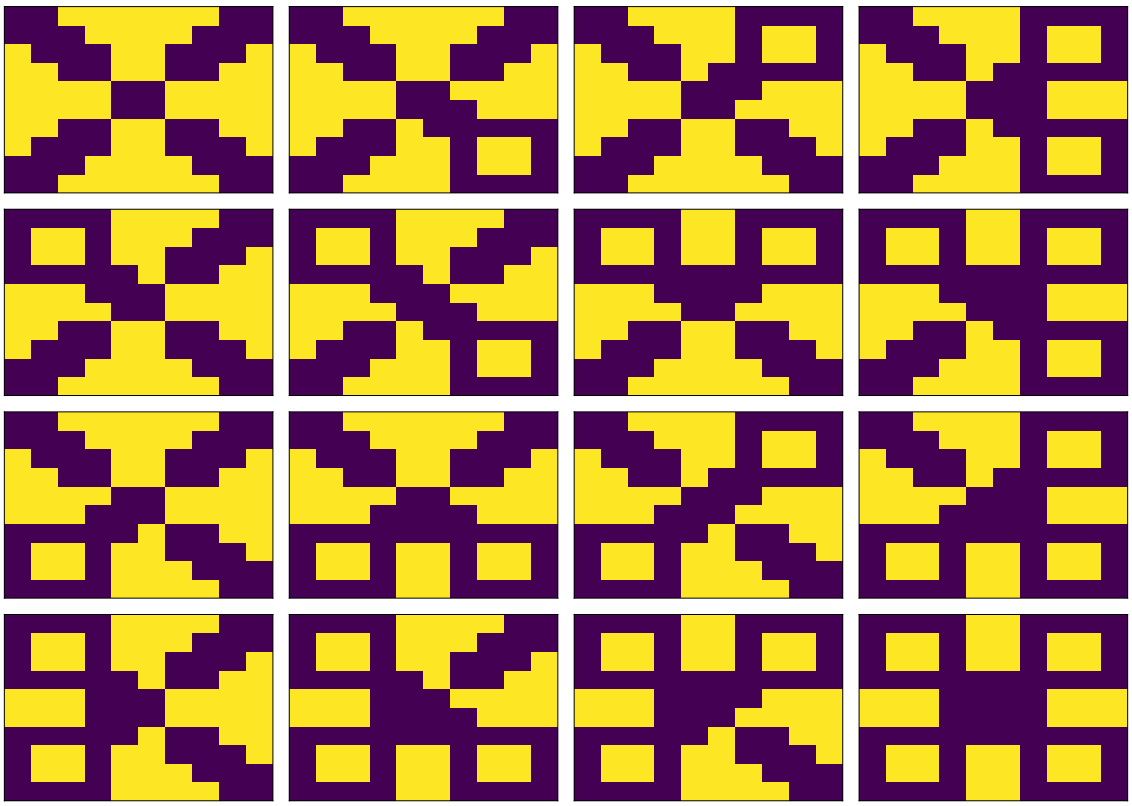

Figure 4.7: The sixteen **modularly varying** Boolean patterns used in the modularly varying pseudo-Boolean benchmarks. These are formed by selecting one of two possible motifs for each quadrant. Each 20×20 pattern is made up of 400 individual bits.

Table 4.4: Parameters used for the control and population-seeding algorithms on the max-ones MVG and leading-ones MVG problems.

Parameter	Max-Ones MVG	Leading-Ones MVG
Genome length	$L = 400$	
Parent selection	Binary tournament	
Mutation type	Flip each gene with probability p	
Mutation probability	$p = 1/L$	
Crossover probability	0 (no crossover)	
Population size	$\mu = 10$	
Generations for the single-task control	2,000	20,000
Generations for transfer during the training phase	2,000	20,000
Generations for transfer during the testing phase	2,000	20,000
Independent runs	100	

each of which computes the number of matching bits to one of the 16 MVG patterns using Equation 4.6. I construct the *leading-ones MVG* benchmark similarly by treating each pattern $\mathbf{z}$ as the target bitstring of a generalized leading-ones problem as defined by Equation 4.9.

The algorithm is similar to the one used in section 4.2.2, and the parameters involved in each experiment are detailed in Figure 4.4. Because the leading-ones problems are more challenging for an EA to solve, they are given a larger budget of evaluations (generations) to work with—but otherwise the algorithms are configured identically for both experiments.

In the experiments, I proceed to test a single-task control against population-seeding transfer on *every possible pair* of tasks. I run each algorithm 100 times on each task pair. In this way I gain a clear picture of exactly which pairs of tasks display a positive transfer relationship, and which do not. Each independent run includes training on the source task and testing on the target task (so each run potentially transfers a different best-found individual from the source task). Specifically, the population-seeding algorithm is run on each source task in the max-ones (leading-ones) MVG benchmark for 2,000 (20,000) generations,

saving the best solution found at any point during the run. Then it runs on the target task for 2,000 (20,000) generations, replacing one individual in the initial population with the prior best solution from the source task.

For these experiments, because the global optimum is easy to find given sufficient generations, I focus on the *area under the BSF curve* (AUC) as a measure of algorithm performance. I only report metrics on the **test phase** of the transfer experiments—my focus being on the cost involved in solving the target task (treating effort spent training on the source task as a sunk cost).

Results

Average best-so-far fitness curves for all source-target pairs of the max-ones MVG benchmark are shown in Figure 4.8. Each sub-plot corresponds to a single target task, and each curve within a subplot corresponds to a difference source task that information was transferred from, with the bold dashed line indicating the single-task control. From these plots it is evident that the transfer experiments always perform at least as well as the control, and often out-perform it. The performance gains on this task come primarily from "jump-start effects" [Taylor and Stone, 2009], in which transfer confers an initial fitness boost that helps accelerate convergence to better solutions.

Figure 4.9 aggregates the same data into a bar-plot view showing the mean area under the curve (AUC) for each source-target pair. Table 4.5 likewise shows the median AUC for each pair, along with indications of which experiments exhibited a statistically significant transfer effect (by a Wilcoxon rank-sum test with Bonferroni correction within each row). As one might expect given the high similarity of the problems in this domain, most task pairings results in positive transfer—with tasks that have greater overlap in target patterns leading to higher boosts in performance. Specifically, there is positive transfer in *every case* except for when the source and task problems are *maximally different*. This is the only case in which the source and target patterns share less than 50% similarity—**confirming Hypothesis 4.3**.

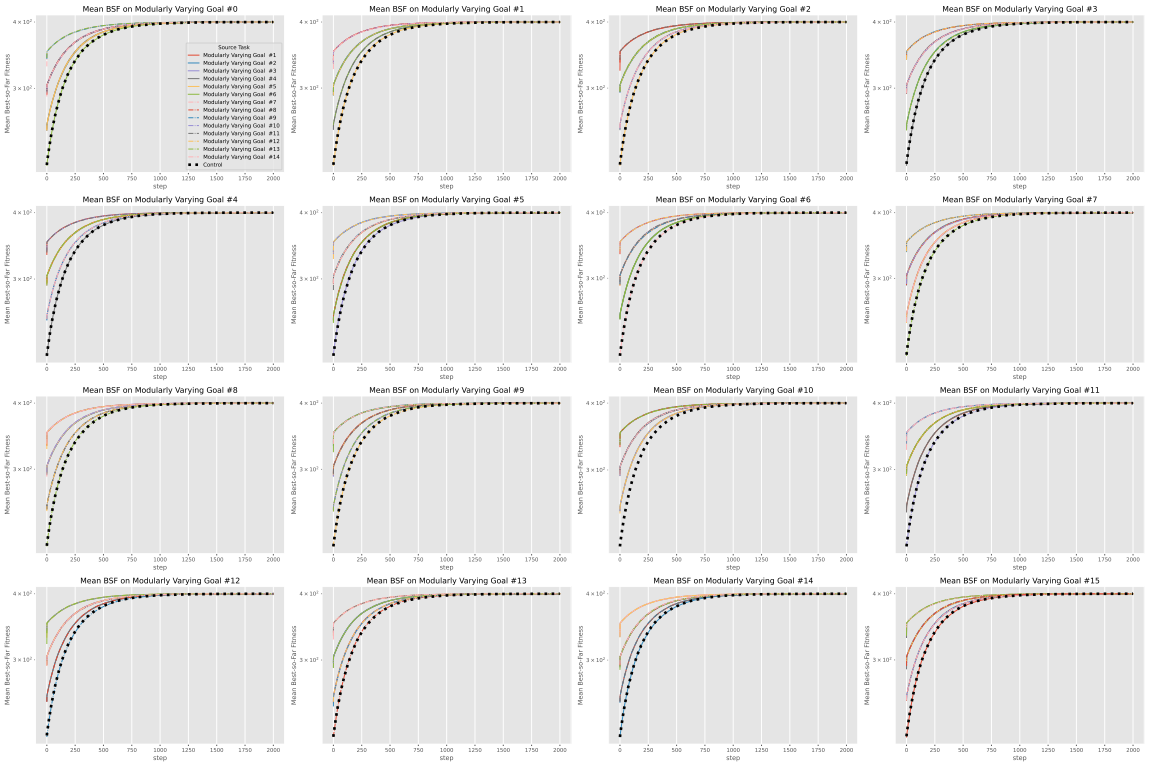

Figure 4.8: Mean best-so-far fitness curves for **population seeding** on the **one-max MVG tasks**. Each subplot represents a *target task* and each curve is the mean of 100 independent runs. The bold black line is the single-task control, and the remaining (color) lines indicate the mean BSF on the target function (during the "testing" phase), each using information from a different source task.

Table 4.5: *Median AUC* values for pairwise population seeding on the **one-max MVG tasks**. Each row presents experiments for one target task, the columns giving results for the single-task control and each source-task experiment, respectively. **Bold** numbers indicate values that show a statistically significant difference from the single-task control with a Wilcoxon rank-sum test and a Bonferroni correction (applied within each row). Asterisks indicate the smallest p-value that significance is achieved at: $* = 0.05$, $** = 0.005$, and $*** = 0.0005$.

Target	Single-Task Control	MVG #0	MVG #1	MVG #2	MVG #3	MVG #4	MVG #5	MVG #6	MVG #7	MVG #8	MVG #9	MVG #10	MVG #11	MVG #12	MVG #13	MVG #14	MVG #15
MVG #0	7.57e+05		**7.88e+05***	**7.88e+05***	**7.77e+05***	**7.88e+05***	**7.77e+05***	**7.77e+05***	**7.67e+05***	**7.89e+05***	**7.77e+05***	**7.77e+05***	**7.66e+05***	**7.77e+05***	**7.67e+05***	**7.66e+05***	7.57e+05
MVG #1	7.57e+05	**7.89e+05***		**7.77e+05***	**7.89e+05***	**7.77e+05***	**7.89e+05***	**7.67e+05***	**7.78e+05***	**7.77e+05***	**7.88e+05***	**7.67e+05***	**7.77e+05***	**7.66e+05***	**7.77e+05***	7.58e+05	**7.67e+05***
MVG #2	7.58e+05	**7.88e+05***	**7.78e+05***		**7.89e+05***	**7.78e+05***	**7.66e+05***	**7.88e+05***	**7.78e+05***	**7.78e+05***	**7.66e+05***	**7.88e+05***	**7.77e+05***	**7.67e+05***	7.58e+05	**7.77e+05***	**7.67e+05***
MVG #3	7.58e+05	**7.77e+05***	**7.89e+05***	**7.88e+05***		**7.67e+05***	**7.77e+05***	**7.78e+05***	**7.89e+05***	**7.66e+05***	**7.77e+05***	**7.77e+05***	**7.88e+05***	7.58e+05	**7.67e+05***	**7.67e+05***	**7.78e+05***
MVG #4	7.58e+05	**7.89e+05***	**7.77e+05***	**7.77e+05***	**7.67e+05***		**7.88e+05***	**7.88e+05***	**7.77e+05***	**7.77e+05***	**7.66e+05***	**7.67e+05***	7.58e+05	**7.89e+05***	**7.77e+05***	**7.77e+05***	**7.67e+05***
MVG #5	7.58e+05	**7.77e+05***	**7.89e+05***	**7.66e+05***	**7.77e+05***	**7.88e+05***		**7.77e+05***	**7.89e+05***	**7.66e+05***	**7.78e+05***	7.57e+05	**7.66e+05***	**7.77e+05***	**7.89e+05***	**7.67e+05***	**7.77e+05***
MVG #6	7.58e+05	**7.77e+05***	**7.67e+05***	**7.89e+05***	**7.77e+05***	**7.89e+05***	**7.78e+05***		**7.89e+05***	**7.66e+05***	7.58e+05	**7.78e+05***	**7.67e+05***	**7.77e+05***	**7.67e+05***	**7.88e+05***	**7.77e+05***
MVG #7	7.57e+05	**7.67e+05***	**7.77e+05***	**7.78e+05***	**7.88e+05***	**7.77e+05***	**7.88e+05***	**7.88e+05***		7.57e+05	**7.67e+05***	**7.67e+05***	**7.78e+05***	**7.66e+05***	**7.77e+05***	**7.77e+05***	**7.89e+05***
MVG #8	7.57e+05	**7.89e+05***	**7.77e+05***	**7.77e+05***	**7.67e+05***	**7.78e+05***	**7.67e+05***	**7.67e+05***	7.57e+05		**7.88e+05***	**7.89e+05***	**7.77e+05***	**7.88e+05***	**7.77e+05***	**7.77e+05***	**7.67e+05***
MVG #9	7.57e+05	**7.78e+05***	**7.89e+05***	**7.66e+05***	**7.77e+05***	**7.66e+05***	**7.77e+05***	7.57e+05	**7.67e+05***	**7.89e+05***		**7.77e+05***	**7.88e+05***	**7.78e+05***	**7.89e+05***	**7.66e+05***	**7.78e+05***
MVG #10	7.58e+05	**7.78e+05***	**7.66e+05***	**7.89e+05***	**7.77e+05***	**7.66e+05***	7.58e+05	**7.78e+05***	**7.66e+05***	**7.88e+05***	**7.77e+05***		**7.89e+05***	**7.77e+05***	**7.67e+05***	**7.88e+05***	**7.78e+05***
MVG #11	7.57e+05	**7.66e+05***	**7.77e+05***	**7.78e+05***	**7.88e+05***	7.57e+05	**7.67e+05***	**7.66e+05***	**7.78e+05***	**7.77e+05***	**7.88e+05***	**7.89e+05***		**7.67e+05***	**7.77e+05***	**7.78e+05***	**7.88e+05***
MVG #12	7.58e+05	**7.77e+05***	**7.67e+05***	**7.66e+05***	7.57e+05	**7.88e+05***	**7.77e+05***	**7.78e+05***	**7.66e+05***	**7.89e+05***	**7.78e+05***	**7.78e+05***	**7.66e+05***		**7.89e+05***	**7.88e+05***	**7.78e+05***
MVG #13	7.57e+05	**7.67e+05***	**7.77e+05***	7.58e+05	**7.66e+05***	**7.77e+05***	**7.88e+05***	**7.67e+05***	**7.77e+05***	**7.77e+05***	**7.88e+05***	**7.66e+05***	**7.77e+05***	**7.88e+05***		**7.77e+05***	**7.89e+05***
MVG #14	7.58e+05	**7.67e+05***	7.58e+05	**7.77e+05***	**7.67e+05***	**7.77e+05***	**7.67e+05***	**7.89e+05***	**7.77e+05***	**7.78e+05***	**7.67e+05***	**7.88e+05***	**7.77e+05***	**7.89e+05***	**7.77e+05***		**7.89e+05***
MVG #15	7.57e+05	7.57e+05	**7.67e+05***	**7.67e+05***	**7.78e+05***	**7.66e+05***	**7.77e+05***	**7.77e+05***	**7.89e+05***	**7.66e+05***	**7.78e+05***	**7.77e+05***	**7.89e+05***	**7.77e+05***	**7.89e+05***	**7.88e+05***	

Figure 4.9: Bar blots of the performance of **population seeding** on all source-task pairings of the **one-max MVG tasks**. Each individual bar shows the mean *area under the best-so-far curve* (AUC) performance of running on a target task given a particular source task (or a no-transfer control with fully random initialization of the population), along with 95% confidence intervals on the mean. Each subgroup of bars within the plots represents a different target task. The leftmost bar (indicated with black hatch marks) in each subgroup is the single-task control; the remaining bars in each subgroup show the performance on the target task after seeding with the best-found solution in a run on the source task (indicated by color). Each subgroup of bars has one bar that appears as if it were "missing," because I do not plot results of a problem transferring to itself (i.e., the same problem cannot be both the source and target).

Similar analysis is given in Figures 4.10 for the experiments on the leading-ones MVG tasks, with AUC bar plots in Figure 4.11. Here I observe a few large positive-transfer effects within each target-task group. Specifically, these large effects occur when the target patterns (Figure 4.7) of the source and target tasks exactly share the bottom (earliest) 50% of their bits—allowing an especially large jump-start effect on leading-ones.

Table 4.5 shows the quantitative results on the leading-ones MVG suite with Wilcoxon rank-sum tests of differences from the single-task control. I see highly significant ($p < 0.0005$) effects for the strongly matching patterns—and beyond that I observe a smattering of weakly significant effects ($p < 0.05$). The latter are the small positive effects that occur when just the first few bits overlap among target patterns: these are not enough to provide a large jump-start effect, but they do provide some benefit (because the initial fitness of random solutions on leading-ones is so small, even a small similarity among problems can provide

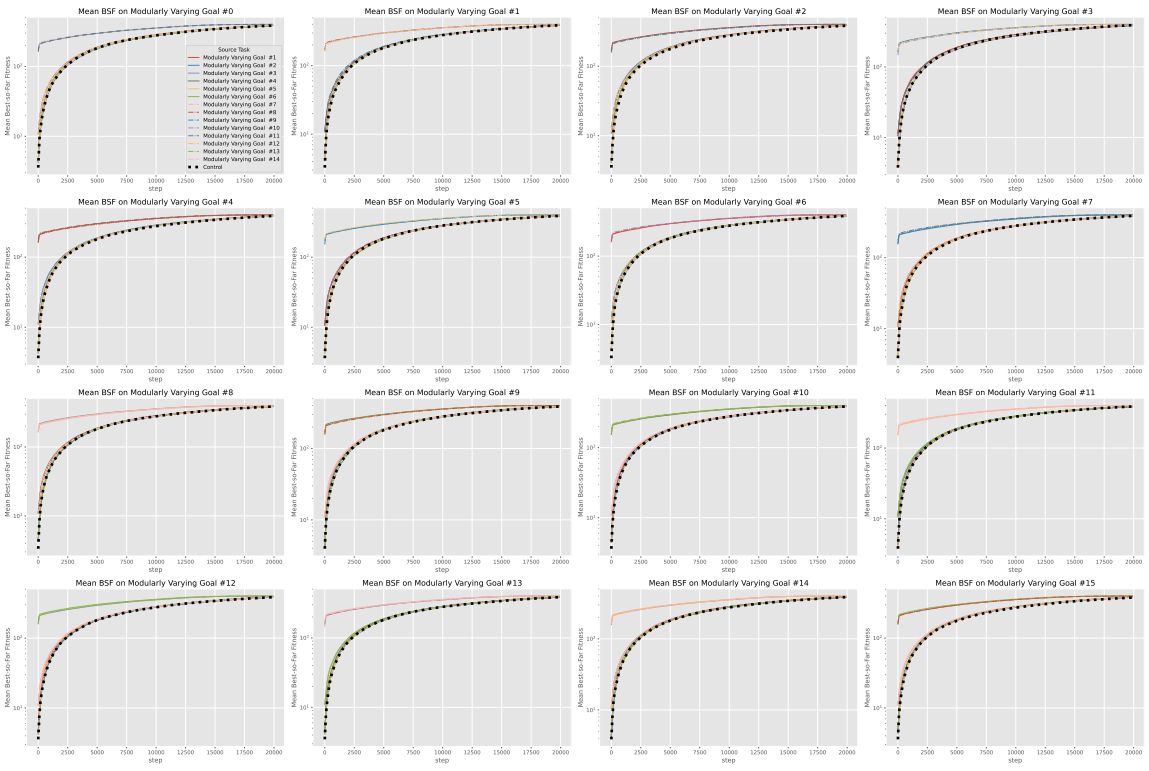

Figure 4.10: Mean best-so-far fitness curves for **population seeding** on the **leading-ones MVG tasks**. Each subplot represents a *target task* and each curve is the mean of 100 independent runs. The bold black line is the single-task control, and the remaining (color) lines indicate the mean BSF on the target function (during the "testing" phase), each using information from a different source task.

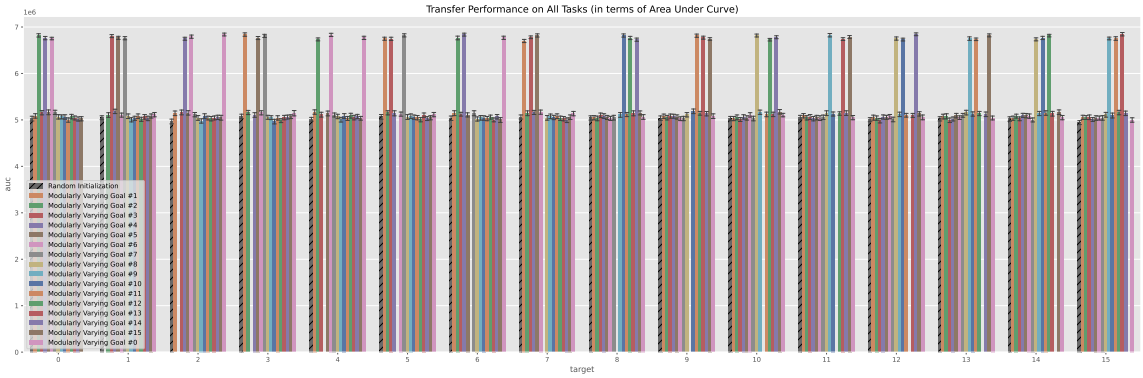

Figure 4.11: Bar blots of the performance of **population seeding** on all source-task pairings of the **leading-ones MVG tasks**. Each individual bar shows the mean *area under the best-so-far curve* (AUC) performance of running on a target task given a particular source task (or a no-transfer control with fully random initialization of the population), along with 95% confidence intervals on the mean. Each subgroup of bars within the plots represents a different target task. The leftmost bar (indicated with black hatch marks) in each subgroup is the single-task control; the remaining bars in each subgroup show the performance on the target task after seeding with the best-found solution in a run on the source task (indicated by color). Each subgroup of bars has one bar that appears as if it were "missing," because I do not plot results of a problem transferring to itself (i.e., the same problem cannot be both the source and target).

some benefit with population seeding). Because the effect size of transfer corresponds to the degree of initial overlap in the patterns, overall these results **confirm Hypothesis 4.4**.

4.3.3 Population Seeding on Real-Valued Functions

The results of section 4.3.2 are useful for understanding how population-seeding transfer operators on different types of simple functions, and how a very simple and easy-to-analyze form of problem similarity (i.e., overlapping target patterns) affects transfer effects. Max-ones and leading-ones problems (and thus the benchmarks I have build from them) have little correspondence to real-world optimization problems, however.

In this section I extend a similar empirical approach to analyze all pairs of a benchmark of simple real-valued optimization problems. The problems in this benchmark are

Table 4.6: *Median AUC* values for pairwise population seeding on the **leading-ones MVG tasks**. Each row presents experiments for one target task, the columns giving results for the single-task control and each source-task experiment, respectively. **Bold** numbers indicate values that show a statistically significant difference from the single-task control with a Wilcoxon rank-sum test and a Bonferroni correction (applied within each row). Asterisks indicate the smallest p-value that significance is achieved at: $* = 0.05$, $** = 0.005$, and $*** = 0.0005$.

Target	Single-Task Control	MVG #0	MVG #1	MVG #2	MVG #3	MVG #4	MVG #5	MVG #6	MVG #7	MVG #8	MVG #9	MVG #10	MVG #11	MVG #12	MVG #13	MVG #14	MVG #15
MVG #0	5.01e+06		5.08e+06	**6.82e+06***	**5.19e+06***	**6.77e+06***	**5.12e+06***	**6.77e+06***	**5.18e+06***	5.07e+06	5.06e+06	5.08e+06	4.99e+06	5.09e+06	5.04e+06	5.02e+06	5.02e+06
MVG #1	5.03e+06	5.12e+06		5.1e+06	**6.81e+06***	**5.19e+06***	**6.77e+06***	5.11e+06	**6.75e+06***	5.09e+06	4.98e+06	5.04e+06	5.06e+06	5e+06	5.06e+06	5.03e+06	5.1e+06
MVG #2	4.96e+06	**6.84e+06***	**5.13e+06***		**5.16e+06***	**6.75e+06***	**5.11e+06***	**6.81e+06***	**5.13e+06***	5.01e+06	4.97e+06	5.09e+06	5.06e+06	5.06e+06	5.06e+06	5.03e+06	5.05e+06
MVG #3	5.07e+06	5.13e+06	**6.85e+06***	5.14e+06		5.13e+06	**6.78e+06***	5.16e+06	**6.83e+06***	5.03e+06	5.01e+06	4.98e+06	5.05e+06	4.96e+06	5.04e+06	5.04e+06	5.07e+06
MVG #4	4.97e+06	**6.76e+06***	**5.16e+06***	**6.73e+06***	5.12e+06		**5.14e+06***	**6.83e+06***	**5.11e+06***	5.06e+06	5e+06	5.07e+06	5.02e+06	5.1e+06	5.03e+06	5.08e+06	5.01e+06
MVG #5	5.05e+06	5.12e+06	**6.74e+06***	5.15e+06	**6.75e+06***	5.13e+06		5.13e+06	**6.83e+06***	5.05e+06	5.07e+06	5.06e+06	5.06e+06	5.01e+06	5.08e+06	5.02e+06	5.02e+06
MVG #6	5.02e+06	**6.77e+06***	**5.17e+06***	**6.76e+06***	5.12e+06	**6.86e+06***	5.11e+06		5.1e+06	5.01e+06	5.04e+06	5.05e+06	5.05e+06	5.06e+06	5e+06	5.09e+06	4.98e+06
MVG #7	5.09e+06	5.13e+06	**6.71e+06***	5.15e+06	**6.77e+06***	5.16e+06	**6.81e+06***	**5.16e+06***		5.02e+06	5.07e+06	5.05e+06	5.07e+06	4.99e+06	5.03e+06	4.99e+06	5.07e+06
MVG #8	5.06e+06	5.04e+06	5.05e+06	5.02e+06	5.1e+06	5.09e+06	5.03e+06	5.03e+06	5.06e+06		5.1e+06	**6.82e+06***	5.12e+06	**6.77e+06***	5.14e+06	**6.75e+06***	**5.14e+06***
MVG #9	5.03e+06	5.11e+06	5.09e+06	5.03e+06	5.1e+06	5.06e+06	5.06e+06	4.99e+06	5.02e+06	5.14e+06		**5.15e+06***	**6.84e+06***	**5.14e+06***	**6.77e+06***	5.11e+06	**6.73e+06***
MVG #10	5.02e+06	5.09e+06	5.01e+06	5.05e+06	5e+06	5.09e+06	5.03e+06	5.09e+06	5.02e+06	**6.81e+06***	**5.11e+06***		5.11e+06	**6.73e+06***	5.11e+06	**6.8e+06***	**5.16e+06***
MVG #11	5.05e+06	5.05e+06	5.11e+06	5.06e+06	5.05e+06	5.06e+06	5.01e+06	5.04e+06	5.02e+06	5.13e+06	**6.83e+06***	5.12e+06		5.16e+06	**6.74e+06***	5.11e+06	**6.78e+06***
MVG #12	5.01e+06	5.04e+06	5.04e+06	5.08e+06	5e+06	5.02e+06	5.06e+06	5.09e+06	5e+06	**6.77e+06***	**5.15e+06***	**6.76e+06***	**5.1e+06***		5.14e+06	**6.86e+06***	**5.15e+06***
MVG #13	5.06e+06	5.04e+06	5.08e+06	5.07e+06	4.98e+06	5.02e+06	5.08e+06	5.04e+06	5.09e+06	**5.15e+06***	**6.74e+06***	5.13e+06	**6.71e+06***	**5.12e+06***		5.16e+06	**6.82e+06***
MVG #14	4.99e+06	5.04e+06	5.06e+06	5.08e+06	5.01e+06	5.15e+06	5.08e+06	5.08e+06	5e+06	**6.74e+06***	**5.13e+06***	**6.78e+06***	**5.13e+06***	**6.8e+06***	**5.11e+06***		**5.18e+06***
MVG #15	4.94e+06	5.01e+06	**5.07e+06***	**5.04e+06***	**5.07e+06***	5e+06	5.04e+06	**5.05e+06***	5.03e+06	**5.11e+06***	**6.75e+06***	**5.1e+06***	**6.78e+06***	**5.18e+06***	**6.84e+06***	**5.14e+06***	

taken from the multi-factorial optimization (MFO) problem suite. These functions were collected by Da et al. [2017b] as an extension of the benchmark that Gupta et al. [2016c] first used to demonstrate the feasibility of their multi-factorial evolutionary algorithm (MFEA). They were originally presented as 9 *pairs* of tasks that were selected for certain similarity properties—partially intersecting global optima, for example, or high fitness correlation—and these have often served in the literature as an initial empirical sanity check on the effectiveness of multi-task evolutionary algorithms.

To my knowledge, these functions have not previously been used in experiments with *sequential* transfer algorithms. So it is an open question whether sequential transfer approaches can be effective in transferring knowledge on the MFO tasks. In this section I perform experiments to address this question, but I also extend my analysis to consider *all pairs* of the MFO tasks—that is, rather than considering only the 9 pairs of tasks that Da et al. [2017b] assigned to source and target roles, I consider all 11 unique functions that appear in the MFO sweet, and all $11 * 10 = 110$ possible pairings of them.

Hypotheses

First, I expect that population-seeding knowledge transfer will perform effectively in this domain (much like multi-task algorithms have been shown to in the past).

Hypothesis 4.5. *Population seeding on the all-pairs MFO tasks will often lead to positive transfer.*

If true, this suggests that population-seeding—despite being a very simple mechanism for representing and transferring knowledge—has benefits and utility that are not entirely dissimilar from those that the MFEA and related multi-task algorithms have shown.

Recall furthermore that in section 4.2.2 I only observed very small negative transfer effects (if any at all) with population seeding on tasks that were not well-suited to transfer. I expect, then, that population seeding will rarely or never exhibit negative transfer on the all-pair MFO benchmark.

Hypothesis 4.6. *Population seeding on the all-pairs MFO tasks will rarely or never exhibit negative transfer.*

When this claim holds, it implies that population-seeding transfer does not harm performance. This is a desirable property to have when attempting to perform source selection with various tasks, some of which may not be effective sources for transfer.

Methods

To restate, while I am using the 11 unique functions that appear in the multi-factorial optimization (MFO) benchmark suite of Da et al. [2017b], I am *not* limiting my analysis to the original 9 *pairings* of these functions. Instead, I consider all possible pairings—forming an "all-pairs" MFO suite. Surface plots for each are given in Figure 4.12. All functions are 50-dimensional unless otherwise noted.

These 11 MFO functions are based on transformations of 7 classic real-valued functional forms, summarized in Table 4.7, such that they are defined over a unified 50-dimensional

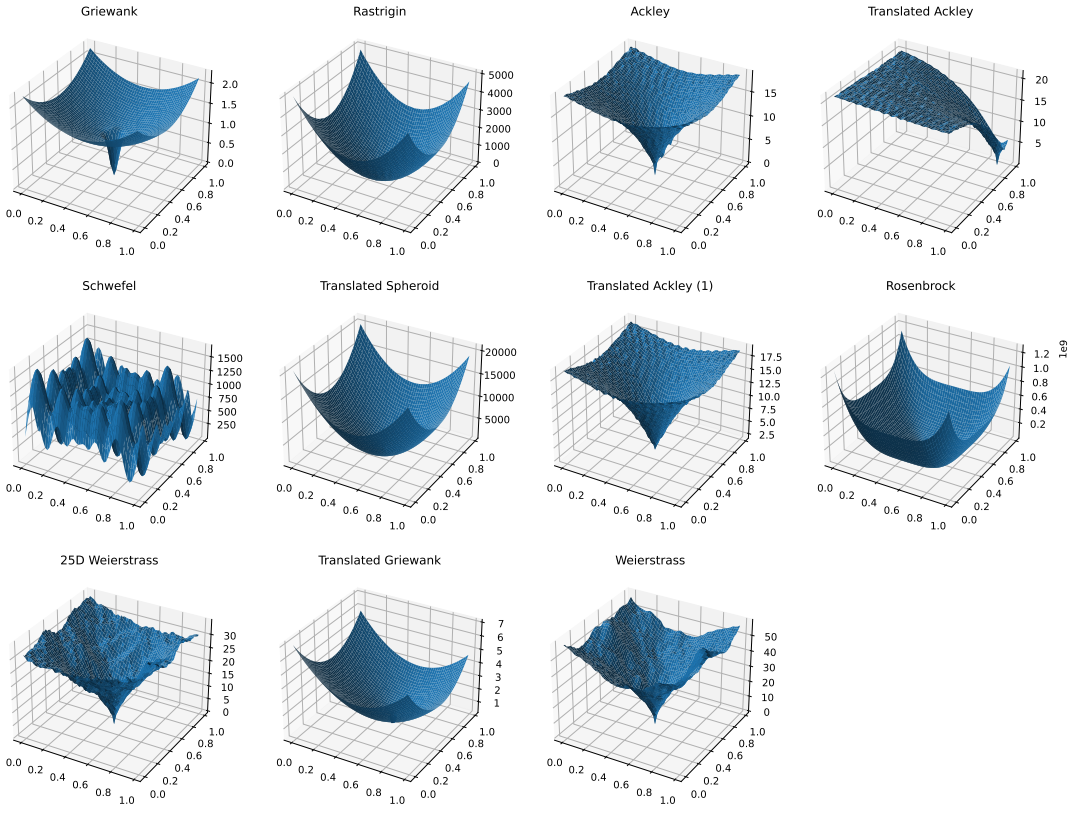

Figure 4.12: Two-dimensional projections of the 11 functions in the MFO benchmark. These surfaces are obtained by projecting the original 50-dimensional (or 25-dimensional, in the case of the 25D Weierstrass) functions into 2 dimensions by fixing all of the remaining dimensions of to the values of that function's global optimum.

Table 4.7: Definitions of the 7 real-valued functions that underlay the MFO benchmark. These are scaled, rotated, and/or translated to create a total of 11 unique benchmark functions. In the original MFO benchmark, these are paired to form 9 predefined source-target pairs—here, however, I consider all 11×11 pairs of tasks.

Name	Basic Equation	Original Range
Griewank	$f(\mathbf{x}) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	$(-50, 50)$
Rastrigin	$f(\mathbf{x}) = An + \sum_{i=1}^{n} x_i^2 - A\cos(2\pi x_i)$	$(-50, 50)$
Ackley	$f(\mathbf{x}) = -ae^{\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right)} - e^{\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right)} + a + \exp(1)$	$(-50, 50)$
Schwefel	$f(\mathbf{x}) = \sum_{i=1}^{d}\left(-x_i \cdot \sin\left(\sqrt{\lvert x_i\rvert}\right)\right) + \alpha \cdot d$	$(-500, 500)$
Spheroid	$f(\vec{x}) = \sum_{i}^{n} x_i^2$	$(-100, 100)$
Rosenbrock	$f(\mathbf{x}) = \sum_{i=1}^{d-1}\left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$	$(-50, 50)$
Weierstrass	$f(\mathbf{x}) = \sum_{i=1}^{d}\left[\sum_{k=0}^{kmax} a^k \cos\left(2\pi b^k(x_i + 0.5)\right) - n\sum_{k=0}^{kmax} a^k \cos(\pi b^k)\right]$	$(-0.5, 0.5)$

search space. Following Da et al. [2017b], I form each of the 11 functions applying scaling, projection, rotation, and/or translation to one of the 7 classic real-valued functions. Specifically, the ranges of all dimensions of each function are scaled such that that their input values fall within (0, 1). In the case of the 25D Weierstrass function (whose dimensionality differs from the other problems), a projection is additionally used to convert unified phenotypes in $(0, 1)^{50}$ to truncated phenotypes in $(0, 1)^{25}$ (i.e., discarding the unused dimensions). Together, scaling and projection provide a unified representation which ensures that the phenotype space is the same (and of similar scale) for each problem. Some of the functions are further rotated and translated (as indicated by their labels in Figure 4.12). The rotation performed is an arbitrary orthonormal transformation (i.e., rotation and reflection)—the exact parameters of each transformation were defined in the original Matlab implementation of the MFO benchmark, distributed by Da et al. [2017b] at `http://www.bdsc.site/websites/MTO/index.html`. I have reused their rotation matrix values in my Python implementation of the benchmark.

The evolutionary algorithm I use as the single-task control and the base of the population-seeding transfer strategy in this experiment uses a real-vector representation with additive

Table 4.8: Parameters used for the control and population-seeding algorithms on the all-pairs MFO task suite.

Parameter	Value
Genome length	$L = 50$
Parent selection	Binary tournament
Mutation type	Additive Gaussian on each gene with probability p
Mutation probability	$p = 1/L$
Mutation width	$\sigma = 0.05$
Crossover probability	0 (no crossover)
Population size	$\mu = 50$
Generations for the single-task control	2,000
Generations for transfer during the training phase	2,000
Generations for transfer during the testing phase	2,000
Independent runs	30

Gaussian mutation. Genomes are initialized uniformly within the range of the MFO benchmark's "unified representation" $(0, 1)^{50}$, except that in the population-seeding case a single individual is replaced with the best-found individual on the source task. The single-task control runs for 2,000 generations on the target task, whereas the transfer algorithm runs for 2,000 generations on the source task, and then an additional 2,000 generations on the target task. The evolutionary components and parameters I use are all summarized in Table 4.8.

The metrics I use to quantify transfer effects in this experiment are the (mean) fitness of the *best-solution found* and, secondarily, the mean *area under the BSF curve* (AUC). Both are calculated from 30 independent runs of the algorithms on each source-target pair in the all-pairs MFO suite. In practice best-solution-found metrics are often what is most important (since the quality of the final solution tends to be what matters most) [Luke and Panait, 2002], but the AUC metrics give us additional insight into algorithm efficiency. As in section 4.3.2, I only report metrics on the test phase of the transfer experiments.

Results

The mean best-so-far curves for the control and population-seeding EKT method are shown in Figure 4.13. Much like the similar plots from section 4.3.2, each subplot in this figure corresponds to a single target task, and each curve represents transfer from a different source task (or the control, in the case of the dashed black curve). In this case the functions are being minimized, and it is evident that on most tasks transfer tends to be favorable. Only on the Schwefel function do I observe cases of negative transfer.

These results are summarized in the bar-plots of Figure 4.14, which show mean results in terms of best-solution-found (top) and AUC (bottom). Here it is evident that some functions (like the 25D Weierstrass) benefit a great deal from many of the transfer conditions, but other functions do not benefit at all (namely the Schwefel function, and the first translated Ackley function).

Table 4.9 shows the quantitative results with Wilcoxon rank-sum tests of differences from the single-task control. Most of the visually discernible improvements are statistically significant (though not all, since I take a Bonferroni correction, making the tests conservative). Two of the 11 tasks show no benefit from transfer from any source task: namely the Translated Ackley (0) and Schwefel functions. There is one statistically significant case of negative transfer: namely from the Translated Griewank to the Schwefel function.

Overall, from these results I conclude that positive transfer does often occur with population seeding on the all-pairs MFO tasks. There are exceptions—some tasks do not benefit at all—but since the purpose of this experiment was to confirm that population seeding is an effective EKT approach on this benchmark (rather than a *perfect* one), I consider **Hypothesis 4.5 to be confirmed**. Likewise, since there is only one instance of statistically significant negative transfer, **Hypothesis 4.6 is confirmed** (namely that negative transfer rarely occurs).

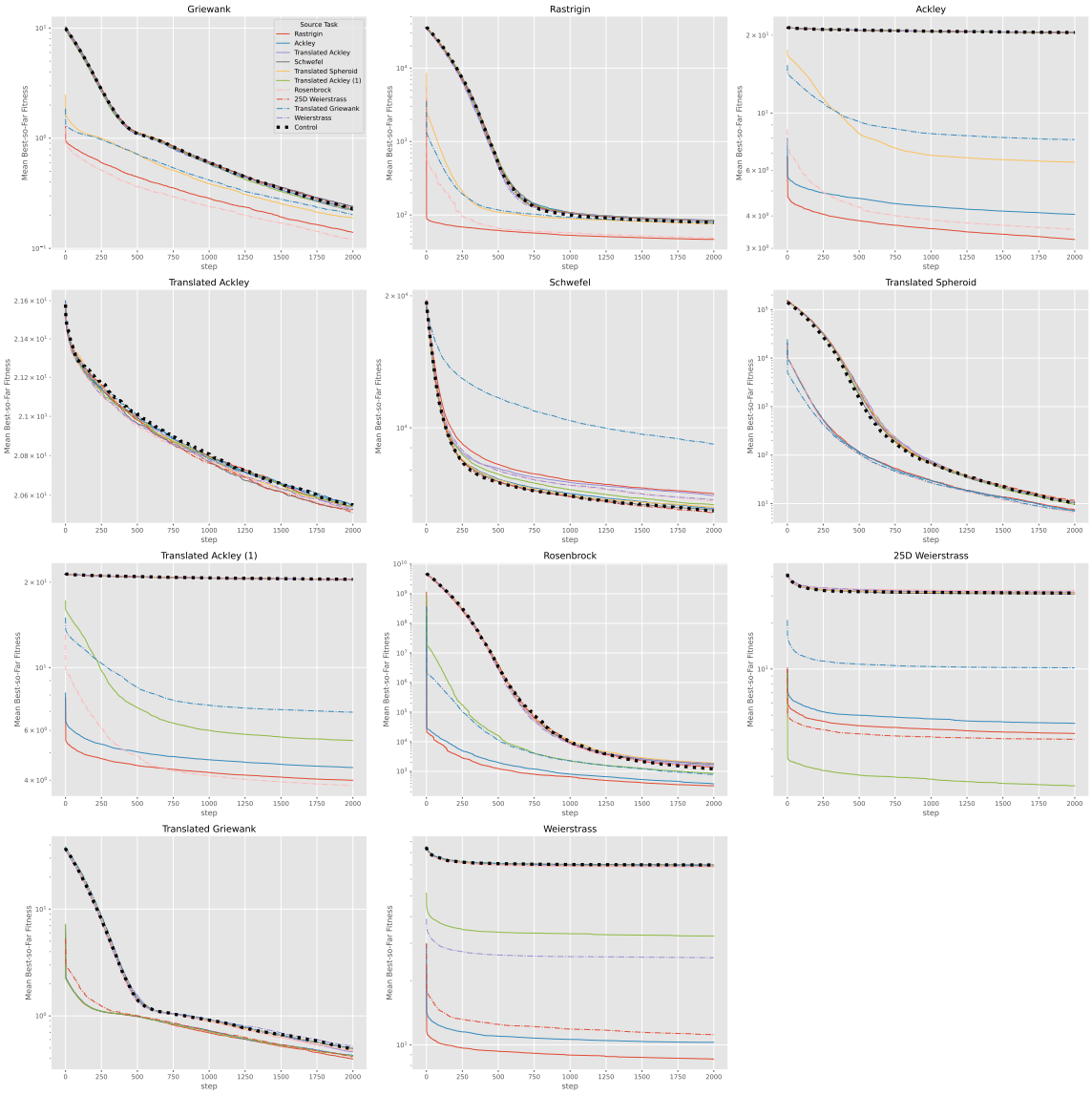

Figure 4.13: Mean best-so-far curves for *population-seeding transfer* on each of the target tasks in the all-pairs MFO benchmark. Within each subplot, each curve represents the performance on the target task when transferring knowledge from a given source task. The dotted black line in each subplot is the control: a single-task algorithm running on the target task with no transfer.

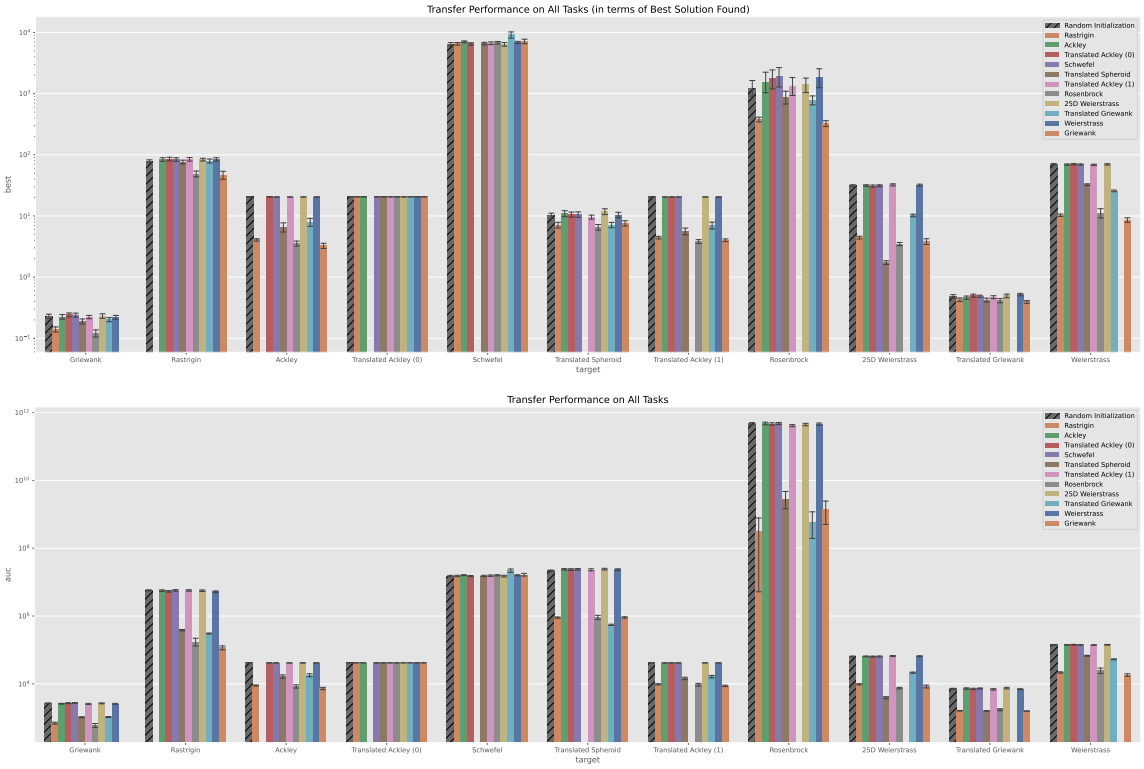

Figure 4.14: Bar plots of the performance of population-seeding transfer on all source-task pairings of the MFO benchmark functions. Each individual bar shows the mean performance of running on a target task given a particular source task (or a no-transfer control with fully random initialization of the population), along with 95% confidence intervals on the mean. The top figure shows mean performance in terms of the fitness of the *best solution found*, while the bottom figure shows mean performance in terms of the *area under the best-so-far curve* (AUC). Each subgroup of bars within the plots represents a different target task. The leftmost bar in each subgroup is the single-task control; the remaining bars in each subgroup show the performance on the target task after seeding with the best-found solution in a run on the source task (indicated by color). Each subgroup of bars has one bar that appears as if it were "missing," because I do not plot results of a problem transferring to itself (i.e., the same problem cannot be both the source and target).

Table 4.9: *Median best-found fitness* values for pairwise population seeding on the **MFO benchmark**. Each row presents experiments for one target task, the columns giving results for the single-task control and each source-task experiment, respectively. **Bold** numbers indicate values that show a statistically significant difference from the single-task control with a Wilcoxon rank-sum test and a Bonferroni correction (applied within each row). Asterisks indicate the smallest p-value that significance is achieved at: $* = 0.05$, $** = 0.005$, and $*** = 0.0005$. The **bold red** value highlights a significant instance of negative transfer.

Target	Single-Task Control	Griewank	Rastrigin	Ackley	Trans. Ackley (0)	Schwefel	Trans. Spheroid	Trans. Ackley (1)	Rosenbrock	25D Weierstrass	Trans. Griewank	Weierstrass
Griewank	0.22		**0.138*** **	0.218	0.233	0.236	0.181	0.223	**0.106*** **	0.234	0.195	0.219
Rastrigin	78.391	**43.334*** **		81.023	86.145	82.84	73.42	81.567	**46.652*** **	84.428	76.582	82.865
Ackley	20.479	**2.881*** **	**4.078*** **		20.471	20.468	**5.762*** **	20.402	**3.363*** **	20.47	**6.963*** **	20.492
Trans. Ackley (0)	20.561	20.56	20.542	20.509		20.536	20.555	20.527	20.546	20.521	20.553	20.527
Schwefel	6293.814	6598.347	6602.557	6979.089	6480.715		6824.837	6677.748	6896.336	6435.193	**11178.665***	6879.305
Trans. Spheroid	10.417	**7.119***	**6.433*** **	9.757	10.48	10.506		9.269	**6.033*** **	11.262	**6.495*** **	10.064
Trans. Ackley (1)	20.476	**3.955*** **	**4.519*** **	20.482	20.457	20.458	**4.995*** **		**3.856*** **	20.461	**5.88*** **	20.438
Rosenbrock	925.065	**313.874*** **	**360.384*** **	1051.125	971.292	993.457	**702.882***	802.377		993.822	**712.475***	1005.737
25D Weierstrass	31.169	**3.556*** **	**4.339*** **	32.266	30.954	31.697	**1.722*** **	32.573	**3.58*** **		**10.046*** **	31.45
Trans. Griewank	0.487	**0.395*** **	0.409	0.461	0.503	0.479	0.43	0.484	**0.401***	0.485		0.52
Weierstrass	70.282	**8.147*** **	**10.255*** **	70.143	69.561	69.564	**32.701*** **	69.117	**8.775*** **	69.293	**25.623*** **	

4.3.4 Discussion of Population Seeding

Population seeding is a very simple strategy for evolutionary knowledge transfer. When successful, it tends to lead to *jump-start* effects early in the run, which are just one possible way that knowledge transfer can have an impact on target-task performance [Taylor and Stone, 2009]. This provides a promising advantage, in that it is possible to determine very quickly whether seeding-based transfer has had a positive effect or not. In particular, when transfer is not useful, selection is able to very quickly remove the useless individual and free up resources for other lineages. This confers a certain *resilience to negative transfer* upon population-seeding approaches.

The trade-off, however, is that population seeding is clearly a very limited method of knowledge representation—and jump-start effects, more generally, are a limited form of benefit. In their seminal discussion of exaptation, Gould and Vrba [1982] note that successful reuse of structures in evolution may not always have an immediate positive impact: transferred structures may need to undergo considerable refinement before they show an advantage at performing a new function. The most powerful kinds of success one would like to see with population seeding, then, may be cases where transferred individuals provide not a

jump-start effect, but instead a long-term exploratory benefit that leads the algorithm away from local optima and into higher-quality basins of attraction. Building benchmarks that elicit this kind of behavior—and studying the conditions under which population seeding (or other methods) is well or poorly suited to exhibit such benefits—would be an interesting angle for future research.

In the population-seeding experiments I have presented in sections 4.3.2 and 4.3.3, it is interesting to now that I did not use crossover of any kind in the algorithm implementations. When transfer leads to better solutions, then, the high-performing lineage must be *entirely descended* from the transferred solution—in which case the randomly sampled portion of the initial population makes no genetic contribution to the solution. Having a diverse initial population is typically important for EA performance. This result may suggest, then, that transfer is beneficial here only when the similarity among problems is *so strong* and easy to exploit that it *outweighs the benefit of diversity.*

This result is surprising, particularly when combined with the fact (which I have not published here) that I have not found crossover to provide any significant added benefit to population-based seeding in these domains. Intuitively, it seems likely that recombining partial solutions to form solutions that have origins partly in transfer, partly in random exploration ought to improve the performance of instance-based knowledge transfer. I have yet to succeed in demonstrating this myself, however, and I think this forms an interesting avenue for future research.

4.4 Source Selection for Sequential Transfer

Having presented preliminary examples of how evolutionary knowledge transfer—and population-seeding in particular—can show benefits on simple classes of problems, I now turn to some preliminary investigation of the source-selection problem.

In this section I consider source selection from two perspectives: first, I present a very preliminary experiment that attempts to apply similarity metrics to problems *a priori* to

predict the potential for positive transfer between them. In the second approach, I consider a many-source population-seeding approach, which exploits the tendency of population-seeding to be resilient to negative transfer and uses this property as a means of performing source selection across a moderately large set of source tasks. These two experiments shed light on Research Questions 5 and 6, respectively.

My experiments here are preliminary investigations that only suggest directions for future research. But if either of these approaches can be successfully developed into effective source-selection strategies for sequential transfer, they could open up new possibilities for the kind of human-machine teaming that I described in section 2.2.5. This in turn could considerably broaden the applicability of knowledge transfer to new domains —particularly to domains where the similarities among problems are not immediately obvious to practitioners.

4.4.1 Transferability Prediction with Correlation and Distance

Similarity estimation is a natural fit for knowledge transfer applications. As I discussed in section 2.2.5, several approaches to online similarity estimation have been developed in the context of multi-task optimization. But these are not directly applicable to sequential transfer, and they are limited in their focus on using distributional similarity among populations (typically via information theoretic measures based on Kullback-Leibler divergence) as a proxy for understanding problem similarity.

Here I experiment with two very simple offline measures of problem similarity: Spearman's rank-correlation coefficient, and the distance between two tasks' global optimum. These correspond to two basic kinds of problem similarity identified by Gupta et al. [2016c] (namely correlational similarity and intersecting optima). The former can be measured in practical applications, though the latter is only available to us for retrospectively analyzing problems whose optimum is already known (a methodology that has often proved useful in the study of optimization problem structure [Jones and Forrest, 1995]). I also give results for a hybrid similarity measure that combines both of these measures into a single score. This preliminary approach to synthesizing information from multiple measures of problem

similarity suggests that this may be an effective approach.

Methods

I apply similarity metrics to the all-pairs MFO benchmark functions that I used in section 4.3.3, and compare the values of these metrics to population-seeding data from that section to see how predictive the former are of the latter.

To measure the *correlational similarity* between a pair of functions (f, g) with the domain $(0, 1)^{50}$, I first sample 10,000 genomes $x \in (0, 1)^{50}$. For each genome, I then obtain fitness evaluations $F = f(x), G = g(x)$ for the genome on each fitness functions. In this way I obtain 10,000 *paired fitness samples* of the landscapes. Then I apply Spearman's rank-correlation coefficient, given by

$$\rho_s(f, g) = \frac{\text{cov}(R_F, R_G)}{\sigma_{R(F)}\sigma_{R(G)}}, \tag{4.17}$$

where $R(\cdot)$ indicates the *rank* of the values within each set.

Second, I also measure the *solution distance* as the Euclidean distance between the two task's global optimum:

$$D(x_f^*, x_g^*) = \|x_f^* - x_g^*\|_2. \tag{4.18}$$

This assumes that I know the global optima of our problems *a priori*. This assumption doesn't hold in practice, of course, but the resulting measure still carries information about how global optimum position relates to transferability.

Third, I also test the predictive power of a *combined metric*. I construct this as a linear function of the above two metrics according to the following equation, which converts ρ_s (which is a similarity measure) to a distance measure $(1-\rho_s)$ and adding it to the Euclidean distance:

$$D_{\text{combined}}(f, g) = \frac{D(x_f^*, x_g^*) + (1 - \rho_s(f, g))}{2}. \tag{4.19}$$

The independent variable in this experiment is **transferability**—i.e., the degree to which

using a task f as a source task improves our performance on target task g. Here I use the ratio of the *area under the fitness curve* (AUC) for the transfer experiment to the AUC for the single-task control as the measure of transferability:

$$T_{f\to g} = \frac{\text{AUC}_{f\to g}}{\text{AUC}_g}, \tag{4.20}$$

where $\text{AUC}_{f\to g}$ indicates the area-under-the-curve while solving g using information from f, and AUC_g indicates the AUC while solving g directly in a single-task control. Because the all-pairs MFO tasks are minimization problems, lower values of $T_{f\to g}$ indicate better transfer in this case.

Results

A visualization of the pairwise Spearman and Euclidean similarities and are shown in Figure 4.15 (two small matrices on top), along with the values of the transferability ratio from population seeding (large matrix on the bottom). This gives us a view of how tasks do (or do not) cluster into sub-groups of mutually similar tasks. In all three matrices, the tasks have been ordered by applying a hierarchical agglomerative clustering (HAC) algorithm using the negative of Spearman's ρ as a distance function—thus grouping clustered tasks together in the Spearman-ρ plot and keeping the same ordering for reference in the others.

Visually, it is evident that neither of these two similarity measures is perfectly reliable at predicting transferability. While the Spearman-ρ metric successfully predicts many of the successful transfers in the bottom right of the matrix, it fails to predict the successful transfers to the variations on the Weierstrass and Ackley functions (in the bottom-center of the matrix). The Euclidean-distance metric, on the other hand, does successfully predict these transfer instances (with the exception of the translated spheroid, which it registers as dissimilar from all other tasks), but this metric (being symmetric) erroneously concludes that transfer will also be useful in the opposite direction, which is not the case.

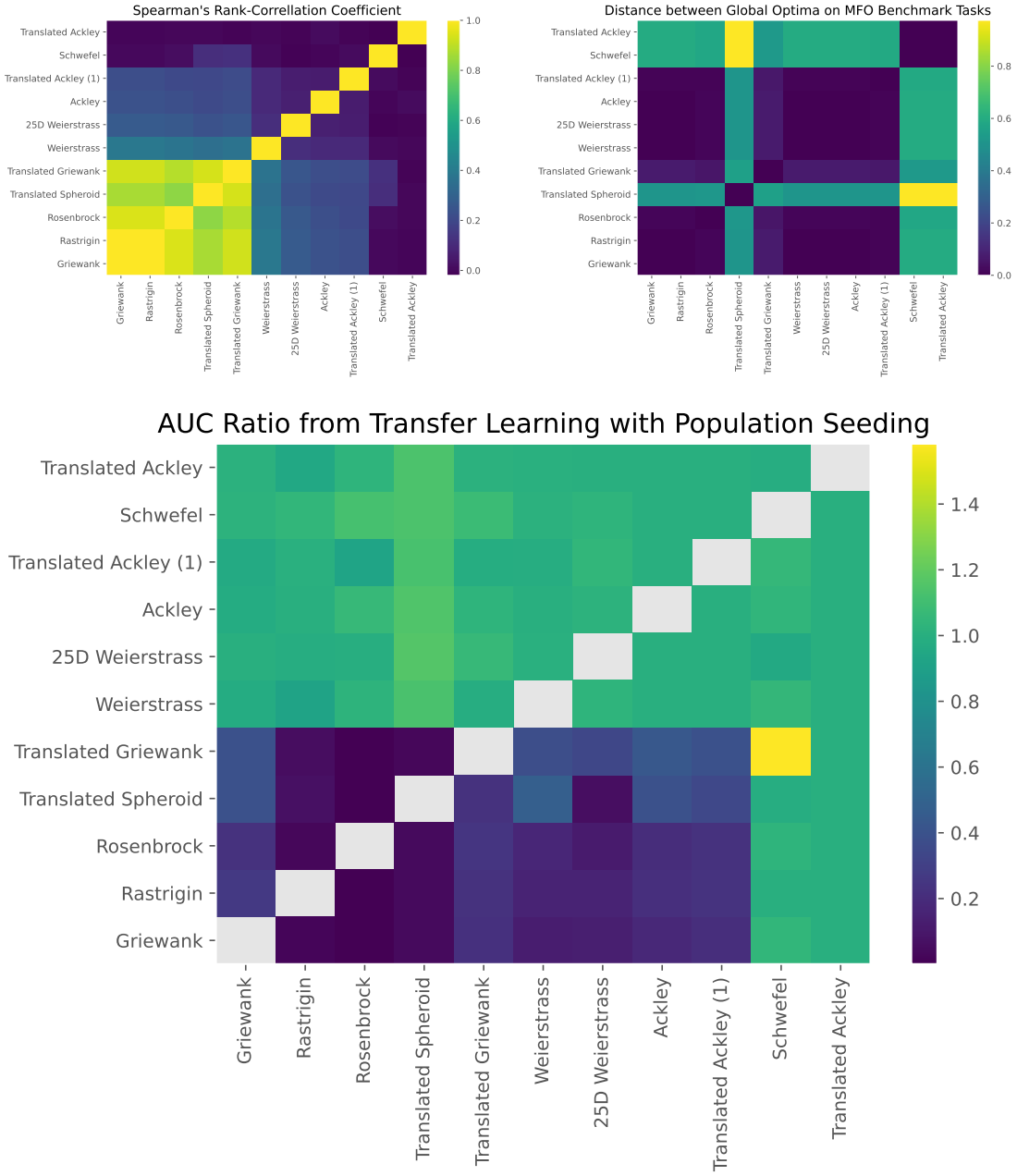

Figure 4.15: Similarity matrices (**top**) and a matrix of transfer performance with population seeding (**bottom**) for the all possible source-task pairings of the 11 tasks in the MFO benchmark. The row indicates the source task, and each column is a different target task. Top-left measures the Spearman's rank-correlation coefficient for each pair, while top-right indicates the Euclidean distance between the functions' local optima. The transfer performance in the bottom matrix is measured by the ratio of the AUC of the BSF curve on the target task to the BSF curve of a single-task control on the same task (so an AUC ratio < 1 indicates positive transfer, > 1 indicates negative transfer).

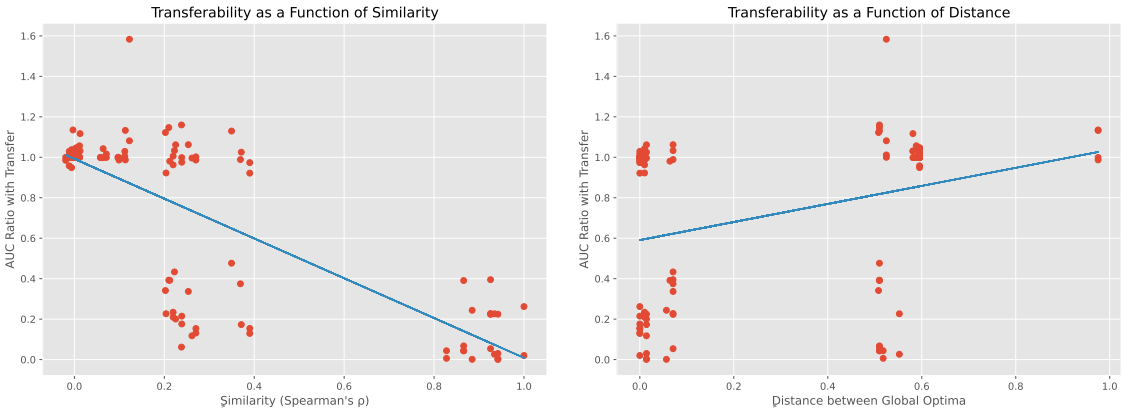

Figure 4.16: Associations between transferability (on the y axis) and the ρ_s (left) and Euclidean distance (right) metrics on the **MFO benchmark**. The data shown here is the same as in the matrix in Figure 4.15.

In general, however, high correlation scores do correlate with better (lower) transferability ratios, and so does lower Euclidean distance. This can be seen in Figure 4.16, which shows least-square regression models for each metric independently.

The complementary nature of the two metrics suggests that combining the two would improve their predictive power. This is indeed the case, as can be seen from a similar analysis in Figure 4.17, where the "combined distance metric" refers to the distance measure given in Equation 4.19.

4.4.2 Many-Source Sequential Transfer

As suggested by Research Question 6 in section 2.2.7 and by the analysis of transfer in population seeding above in section 4.3, many-source sequential transfer may be a powerful approach to performing source selection for knowledge transfer.

In this section I provide a preliminary test of this hypothesis by applying a many-source population-seeding strategy to the all-pairs MFO benchmark from section 4.3.3.

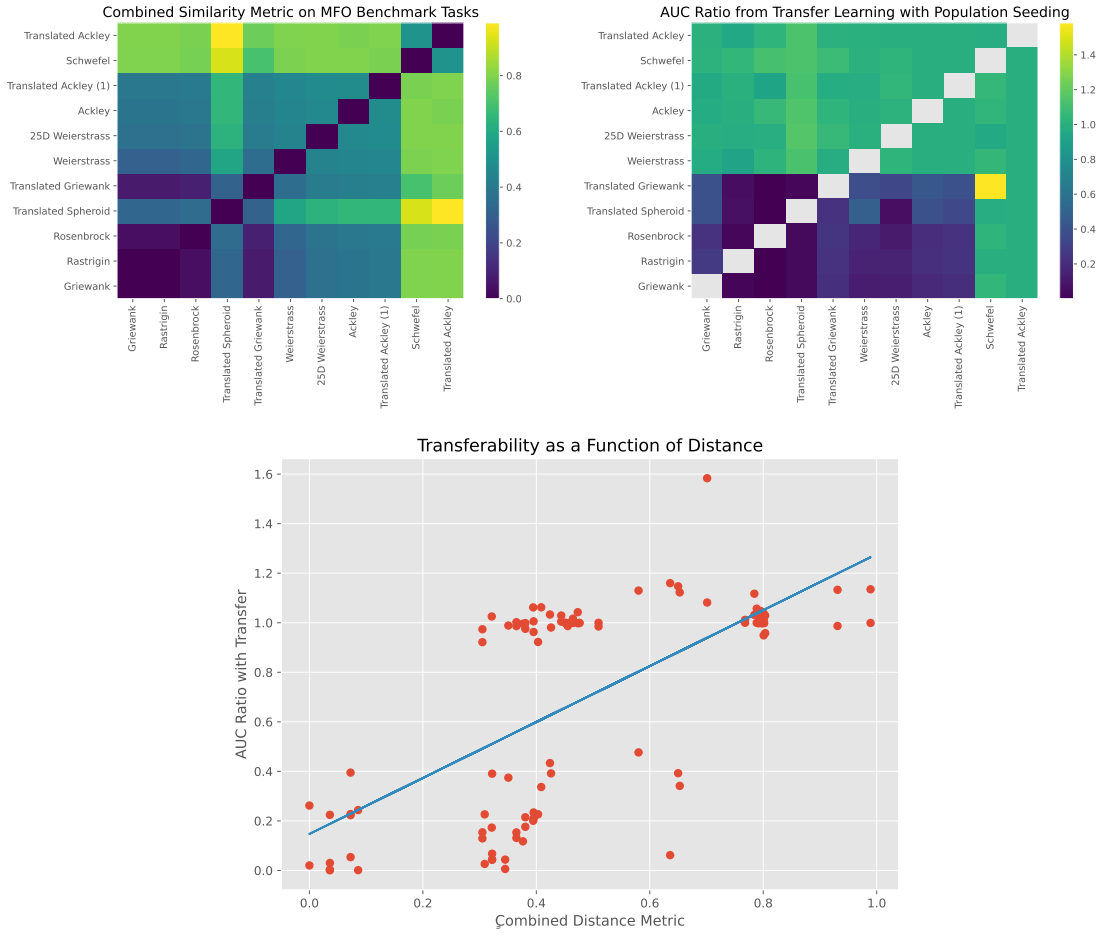

Figure 4.17: Association between transferability (on the y axis) and the combined measure of task distance on the **MFO benchmark**. The distance matrix and AUC matrix for the task pairs are shown at the top, and the linear correlation between the two is evident in the scatter plot below.

Methods

I performed many-to-one sequential knowledge transfer with a many-source population-seeding algorithm as follows. I took turns treating each of the 11 unique functions in the MFO suite as the target task. In each experimental batch, the remaining 10 tasks all serve as source tasks. I refer to a particular arrangement of 10 source tasks and 1 held-out target task as a "source-target split" For each choice of target task, I first performed 30 independent runs of "training": this involves running the EA on each source task for 2,000 generations (for a total of 20,000 generations per independent trial—although the runs can be executed in parallel). I then saved the best-found solution on each of the 10 source tasks into a *repertoire*—a file that stores the solution instances from all of the source tasks.

With 30 repertoires thus collected for each source-target split, I then proceed to the "testing" phase. For each repertoire, I run 30 additional independent runs of the EA on the target task, using population seeding. In this population-seeding configuration, all 10 individuals from the repertoire are inserted into the initial population. The remaining $\mu - 10 = 40$ spaces in the population are filled with random individuals.

In total, this experimental procedure led me to collect 330 repertoires on the source tasks (30 runs for each source-target split), and to perform 9,900 independent runs of the EA on the target functions (900 for each target task: 30 runs each for each of the 30 repertoires trained with that target held-out). The underlying EA used in both population seeding and a single-task control is the same as in section 4.3.3. The parameters are listed again in Table 4.10 for unambiguity.

Results

The mean area-under-curve (AUC) results for each target task using many-source transfer are shown in Figure 4.18. The confidence intervals around the mean estimates are very tight, indicating highly significant differences between the control and many-source algorithm in most cases.

Table 4.10: Parameters used for the control and many-source population-seeding algorithms on all-pairs MFO task suite.

Parameter	Value
Genome length	$L = 50$
Parent selection	Binary tournament
Mutation type	Additive Gaussian on each gene with probability p
Mutation probability	$p = 1/L$
Mutation width	$\sigma = 0.05$
Crossover probability	0 (no crossover)
Population size	$\mu = 50$
Generations for the single-task control	2,000
Generations for transfer during the training phase	2,000
Generations for transfer during the testing phase	2,000
Independent runs	30

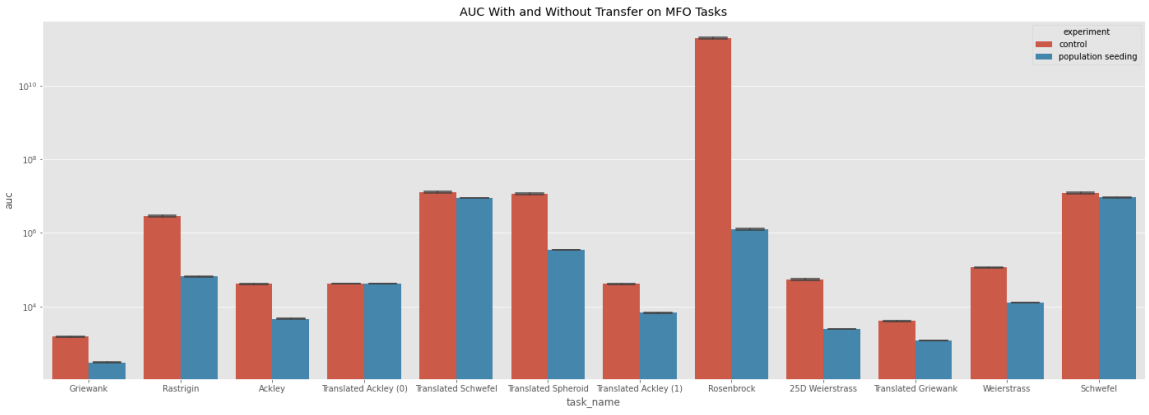

Figure 4.18: Mean AUC results on the held-out target task when performing many-source population seeding from the remaining 10 tasks. Error bars indicate 95% confidence intervals on the mean.

Overall, the many-source transfer approach does not show any evidence of negative transfer. Not only do I observe significant improvement on most target tasks, but the degree of improvement closely tracks the best improvement that I observed previously in the pairwise experiments of section 4.3.3. This suggests that the many-source transfer algorithm is effectively identifying the source task in the repertoire that has the highest relevance to the target task in each instance.

4.5 Conclusions & Discussion

In this chapter I have present results of my preliminary efforts to understand where sequential evolutionary knowledge transfer is likely to be most useful, and how source-selection strategies might expand the set of applications it can be applied to. Throughout this chapter I have kept the notion of a problem *class* central to the discussion, rather than narrowing in on a few carefully selected problem instances where I know that transfer is useful. This is because I believe that the practical usefulness of transfer is determined largely by the similarity structures ("taskonomy," if you like [Zamir et al., 2018]) that occur within problem domains.

In addition to the no-free-lunch theorems that I have proved and the proof that leading-ones is $O(n \log n)$ with suitable knowledge transfer, the empirical hypotheses I have tested are summarized in Table 4.11.

4.5.1 Research Question 4: Problem Classes

In Research Question 4, I asked whether no-free-lunch theorems hold for EKT and, relatedly, what kinds of problem classes positive transfer is likely to occur in with some frequency. My proofs of NFLTs have answered the first part of the question in the affirmative. What the implications of these theorems are is more difficult to say (see section 4.2.3), but of the five problem classes I have introduced in this chapter for the study of EKT, I have shown through Hypotheses 4.1 and 4.2 that the two least restricted classes exhibit positive transfer

Table 4.11: Summary of the hypotheses tested in Chapter 4.

Hypothesis	Description	Result
	Experiments on Permuted Max-Ones and Multi-Peaks Problem Classes	
Hyp. 4.1	Positive transfer is rare on permuted max-ones.	Supported
Hyp. 4.2	Positive transfer is rare on multi-peaks.	Supported
	Transferability in Instance-Based EKT	
Hyp. 4.3	Positive transfer on most max-ones MVG tasks.	Supported
Hyp. 4.4	Positive transfer with overlap on leading-ones MVG.	Supported
Hyp. 4.5	Positive transfer common on all-pairs MFO tasks.	Supported
Hyp. 4.6	Negative transfer rare on all-pairs MFO tasks.	Supported

effects very rarely. This seems to indicate that the no-free-lunch principle for transfer still holds to some degree as its formal requirements are relaxed; or at least, it suggests that transfer sources are hard to find in worlds where tasks are generated according to simple random principles. In real-world problem classes, problems are likely to have richer similarity structures—perhaps because real-world structures share common causal principles, or tend to have lower complexity in the sense of Kolmogorov [Cover and Thomas, 2006, ch. 14].

I have presented three problem classes where transfer performs well—the MVG max-ones, MVG leading-ones, and all-pairs MFO benchmarks. In all of these, population-seeding transfer proves surprisingly effective at exploiting similar tasks—and, especially, of avoiding negative transfer in the process. From a practical perspective, this bolsters my belief that population-seeding is worth using as a simple and easily applicable EKT strategy for many applications.

4.5.2 Research Question 5: Transfer Prediction

Toward Research Question 5, I have presented a preliminary view into how empirical measures of problem similarity might be used in the future to predict in advance which source tasks in a problem class may be useful for solving a particular target task. My analysis here is based on very simple principles: Spearman's rank-correlation of fitness landscapes, global

optima distance, and a combination of the two. These methods have clear limitations in the kind and amount of information they can gather about inter-task synergies. In the future, I would like to pursue a study of how the tools of exploratory landscape analysis (ELA) [Kerschke, 2017, Scott and De Jong, 2016a]. In particular, the two seminal papers in the field of ELA by Mersmann et al. [2011] and Kerschke et al. [2014] have established a standard collection of several dozen easy-to-compute landscape features. These can be used as features to quantify a variety of different properties of fitness landscapes. A promising area for future research would be to investigate the predictive power of these ELA features—perhaps in conjunction with supervised machine learning—for the task of predicting transferability and performing source-selection.

4.5.3 Research Question 6: Many-Source Transfer

My main contribution in the context of Research Question 6 has been the introduction of and validation of many-source population seeding as a viable approach to EKT. This strategy performs remarkably well on the all-pairs MFO suite that I present here, avoiding negative transfer while being very quick to isolate solutions in its repertoire that are beneficial for transfer. On a practical level, algorithms of this kind could dramatically reduce the cost and difficulty of applying EKT to knew problems: an effective many-source algorithm may save practitioners the trouble of manually testing a large number of candidate source tasks by trial-and-error. The reality, however, may not be so simple: the good performance I have shown of many-source transfer on the all-pairs MFO benchmark is directly attributable to the fact that transfer provides a *jump-start effect* on these problems. It is the jump-start effect that allows many-source EKT to rapidly isolate useful material and discard non-useful material. In more general cases, many-source transfer is likely to be difficult, because the problem of source selection within a large repertoire of material raises challenges—including catastrophic forgetting [Rusu et al., 2016] or the memory swamping problem, depending on the knowledge representation used [Markovitch and Scott, 1988]. My contributions here are just a preliminary step, then, toward the larger program of understanding and refining the

potential of many-source algorithms.

The goal of understanding transferability and source selection in evolutionary computation is currently held back by a lack of adequate benchmarks. In the EKT literature as it stands today, there are few benchmark environments that are available to test targeted questions about how knowledge transfer works. The multi-factorial optimization (MFO) suite of Gupta et al. [2016c] is perhaps the most widespread, and I have used it as one of my primary benchmarks here, but it consists of a small set of relatively simple real-valued optimization problems that are limited in the insights they can offer into algorithm performance and generalization. The MFO suite currently occupies a position much like the De Jong test suite used to occupy for single-task optimization [De Jong, 1975] (indeed, several of the MFO functions descend from the De Jong suite): a small set of functions that is widely used as a first proving grounds to test new algorithms upon. The EC community has long moved past the De Jong suite, however, in favor of more sophisticated benchmarks that do a better job of testing specific properties that are useful for understanding applications [Hansen et al., 2010, McDermott et al., 2012]. Over time, sub-communities within AI often come to "blacklist" certain problems or benchmarks that have become overused, as it becomes evident that these problems are not sufficiently informative to form scientifically useful test subjects [White et al., 2013]. I believe the evolutionary knowledge transfer community can and should follow a similar trajectory of progressive methodologies.

One solution to the limitations of simple canonical benchmarks is to focus on more complex, application-centered benchmarks. Zhang et al. [2021b] for example focus on a set of automatically generated job-shop scheduling problems, and Bali et al. [2019] apply their A-MFEA-RL algorithm for multi-task reinforcement learning to the Meta-world benchmark of robotic arm-manipulation tasks [Yu et al., 2020]. This is a useful route to pursue here, and one that I intend to pursue in future work. But application-oriented benchmarks of this kind—while often persuasive and realistic—lack the simplicity that is necessary to understand how the properties of individual problems interact with an algorithm to offer causal explanations of their behavior and performance. In the future I would like to see

more problem classes of the kind I have introduced here—the permuted max-ones, multi-peak, MVG max-ones, and MVG leading-ones classes—studied and expanded upon to allow the community to build a deeper understanding of the conditions under which transfer and source selection can be successful, and the conditions under which it is likely to fail.

Chapter 5: Representation-Based Evolutionary Knowledge Transfer

> Instead of devoting the necessary time and critical thinking required to frame a problem, to adjust our representation of the pieces of the puzzle, we have become complacent and simply reach for the most convenient subroutine, a magic pill to cure our ills. The trouble with magic is that, empirically, it has a very low success rate, and often relies on external devices such as mirrors and smoke.
>
> —Michalewicz and Fogel [2013]

In this chapter I address my final remaining research questions: whether genetic representations can be automatically learned for evolutionary algorithms to facilitate knowledge transfer (Research Questions 8 and 7; see section 2.2.7). If successful, representation-based knowledge transfer may permit information to be learned and transferred a higher level of abstraction than is possible with the instance-based methods that currently dominate the EKT literature.

5.1 Multi-Task Evolution via Shared Layers

In this section, I experiment with an implicit kind of representation learning that occurs naturally in Cartesian genetic programming (CGP).[1] The essential feature of this application is that the solutions being evolved are themselves *executable objects*—and in particular Boolean functions. It is straightforward to represent executable objects with graph structures, in which some sub-components feed into other sub-components that contribute to the function's output.

[1] I have published all the results from this section in Scott and De Jong [2017].

In particular, a key feature of CGP is the natural way in which it represents the *sharing* of sub-graphs within a function. I exploit this property to achieve multi-task knowledge transfer by evolving a single graph structure that uses multiple output to encode solutions to different tasks.

Among the transfer methodologies surveyed and categorized in Chapter 2, the multi-task CGP algorithm can be seen as either an instance-based knowledge representation or a representation-based one. This is because where executable objects are concerned, the distinction between a concrete function instance and the *representation* of computational primitives that those functions are built from is blurred. The driving motivation here is the recognition that, even if complete solutions to a problem are not reusable for other tasks, some partial solutions or computational sub-components may perform operations that have some generalized usefulness across tasks. The challenge lies in identifying those reusable sub-components and effectively exploiting them.

In this section I present results of multi-task CGP applied to the task of synthesizing a suite of 9 two-valued Boolean functions. I narrow my focus to Cartesian genetic programming for these experiments because CGP lends itself especially naturally to what I call a *shared sub-graphs* model of information reuse. In many application domains, candidate solutions to a task can be represented as a graph structure that computes some function over a set of input nodes. The internal nodes of this kind of structure compute intermediate results, which are converted by further processing into output values which (in a successful solution) solve the application's computational objective. Early work on multi-task learning for pattern recognition tasks quickly recognized that the intermediate results of such computations have the potential to be reused across related tasks, even in cases where the complete graph is too specialized to be reused [Caruana, 1998]. In contemporary neural network vocabulary, this approach is known as the *hard parameter sharing* strategy for multi-tasking (as opposed to *soft parameter sharing*, which maintains entirely separate graphs for different solutions, with a soft constraint that encourages them to share information) [Crawshaw, 2020]. Specifically,

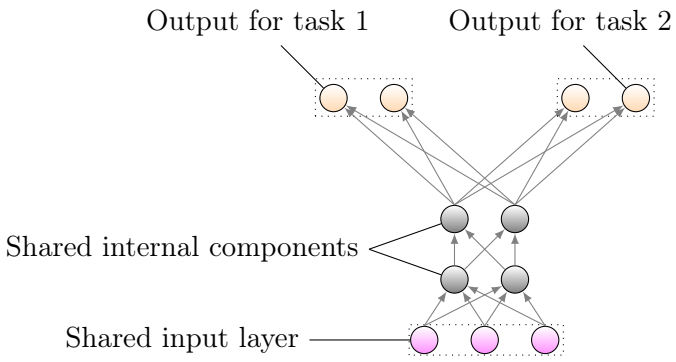

Figure 5.1: In the shared sub-graphs model of multi-task evolution, a single composite individual aims to solve multiple tasks simultaneously, potentially reusing partial solutions across two or more tasks.

an otherwise single-task algorithm that operates with graphs can be converted into a multi-task approach by simply augmenting the graph with a separate set of output nodes for each task being solved, as shown in Figure 5.1. This basic idea can be generalized to optimization algorithm design. It gives us a means of incorporating knowledge transfer into existing single-task algorithms, instead of turning to a more complex, special-purpose algorithm.

The objective in this case is to evolve the edges that connect the nodes so as to synthesize a circuit that computes several distinct Boolean functions. As mutation alters which nodes are connected, it has the possibility of causing the sub-graph that computes the output for one task to reuse a portion of the sub-graph that computes the output for another task. As a result, mutation itself effectively works to solve a restricted version of the source selection problem. While in one sense, source selection has already been performed when a human decides on the set of objectives that a multi-task algorithm will be applied to, the algorithm must still solve the online problem of determining which tasks information should be transferred among within this set, which specific information from those knowledge sources should be transferred, and in what order. By using the shared sub-graphs model to convert a standard, single-task version of CGP into a multi-task CGP algorithm, I investigate the ability of this evolutionary approach to automatically find opportunities to reuse information.

Hypotheses

At a high level, if the multi-task approach leads to a performance boost, it indicates that some form of beneficial knowledge reuse is occurring. A natural question, then, is to look for such an improvement:

Hypothesis 5.1. *A multi-task approach to Cartesian genetic programming will be able to reduce the amount of computational resources that are required to solve moderately sized sets of tasks (on the order of 10).*

Now, since multi-task algorithms aim to meet several criteria for success, the demands posed by the various tasks may sometimes interact in complex ways over the course of an evolutionary run. The multi-task approach will be successful when these interactions are generally helpful, as when progress on one task directly facilitates concomitant progress on another task. But even when information sharing is helpful overall, it may be that the opposite occurs in some cases or at some points during a run: progress on one task may sometimes delay progress on another. Such negative effects are often referred to with the term *task interference*, which is borrowed from psychology [Pashler, 1994]. Such interference can affect the efficiency of a multi-task algorithm even if it doesn't lead to fully fledged negative transfer in the overall approach.

Hypothesis 5.2. *Multi-task Cartesian genetic programming will exhibit task interference.*

In particular, one of the ways that task interference may appear is through mutations that improve the solution to one task by *damaging* the solution to another. Destructive mutations of this kind may lead to an unnecessarily inefficient multi-task optimization procedure. Following this intuition, one might mitigate the effects of this kind of task interference by explicitly protecting components of successful solutions from mutation:

Hypothesis 5.3. *If the point-mutation operator is biased to prefer leaving solutions to successfully solved tasks intact, it will improve the performance of multi-task CGP on moderately sized sets of tasks.*

The hypothesis that mutation weighting of this kind will be beneficial is not a forgone conclusion. It's possible, for example, that by seeking to protect established components of the graph from the destructive effects of mutation (thus mitigating one form of task interference), the algorithm's long-term ability to evolve shared components that are useful across multiple tasks may be impeded (exacerbating another form of task interference).

Background on Cartesian Genetic Programming

Cartesian genetic programming (CGP) is one of several approaches to the evolution of programs and/or graph structures that are currently popular in the evolutionary computation community [Miller and Thomson, 2000]. For our purposes in this study, the main advantages of the Cartesian approach over other common representations (such as tree-based GP) are twofold. First, CGP explicitly evolves directed acyclic graph (DAG) structures. This makes it straightforward to represent structures that reuse one or more sub-components across several regions of a larger solution. "Graphs, by definition," notes Miller, "allow the implicit reuse of nodes, as nodes can be connected to the output of any previous node in the graph" [2011, p. 18–19], whereas more traditional genetic programming approaches lend themselves less naturally to such reuse. Second, CGP often performs remarkably well with small population sizes and with a point-mutation operator as its only variation strategy. This makes for a very simple algorithm. CGP algorithms have also been shown to be resistant to developing bloat—i.e., the widespread problem of rapidly growing program length that plagues most genetic programming systems [Miller and Smith, 2006]. These features are attractive to us here because they make it possible to analyze the behavior of the algorithm in isolation from the more complex dynamics that would likely be introduced by crossover operators, program bloat, and larger population sizes.

Since its introduction in the early 2000s, several alternative variations of CGP have been introduced to offer support for crossover operators, cyclic graphs, encapsulated components, "snapping" connections, and other advanced features [Clegg et al., 2007, Kaufmann and Platzner, 2008, Khan et al., 2010, Wilson et al., 2018]. Except where otherwise specified,

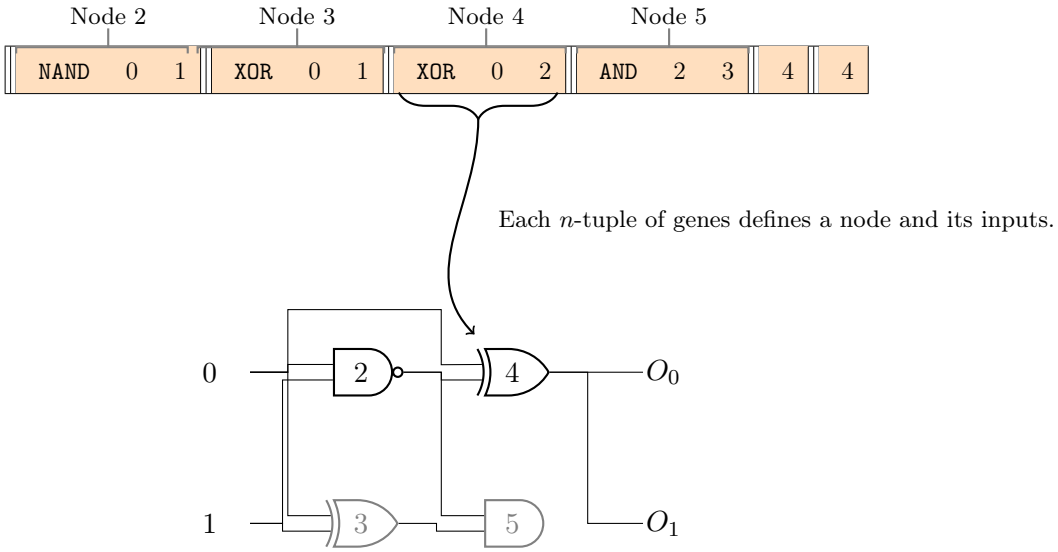

Figure 5.2: An example of how a CGP genome is decoded into a graph structure, envisioned here as a logic circuit.

my focus here is on the original, "standard" CGP algorithm.

The distinguishing feature of CGP is that it uses a linear, fixed-length genome to represent the nodes and edges of a graph-based solution to some problem. The graph itself can be thought of as the solution's "phenotype." The nodes in this graph are laid out in two dimensions along CGP's eponymous Cartesian grid, as illustrated in Figure 5.2. Each of these grid nodes computes some function over their inputs. In addition to the nodes lying along the grid, the graph as a whole is fed a fixed number of external inputs and outputs, which act as non-computational nodes. Edges between all the nodes collectively form a directed acyclic graph that extends from the external inputs to the external outputs.

The graph topology of CGP candidate solutions is constrained in a few different ways. Notably, the acyclic property of the graph is enforced by requiring that a node in a given column can only take its input from nodes in previous columns—thus imposing a topological sort over the graph's nodes. Each computational node has a fixed number of inputs, and each of these inputs must be connected to exactly one other node. As a result, the total number of edges in each graph is constant throughout the execution of the algorithm. Each node's

output, meanwhile, is free to be used as an input by zero or more other nodes that occur in later columns. The standard CGP algorithm additionally takes three parameters that constrain the DAG topology, *viz.* the number of rows and columns n_r and n_c in the Cartesian grid, and a "levels back" parameter l that specifies the maximum number of columns that a connection is allowed to "skip over" to connect two nodes. In practice, however, there is often no observed performance advantage to restricting the pattern of connections in this way. When this is the case, and when no restrictions on the graph topology are required by the problem domain itself, the best performance is achieved by fixing $n_r = 1$ and $l = n_c$. It would be misleading, then, to place undue emphasis on the two-dimensional "Cartesian" aspect of Cartesian genetic programming: CGP is simply a strategy for evolving directed acyclic graphs, which may or may not be constrained to follow a particular layered grid pattern.

While a candidate solution produced by CGP always contains exactly $n_r \cdot n_c$ computational nodes, usually only a subset of these nodes lie on a path to an external output. A substantial fraction of the nodes typically go unused in a given solution, and can be discarded without affecting the behavior of the function that the graph computes. This "intron" material has a significant effect on the algorithm's evolutionary search strategy, as mutations can easily activate and deactivate portions of the graph by altering the edges that connect various components.

As for the genome, CGP represents each computational node as a tuple of genes $(f_i(\cdot), a_1, \ldots, a_n)$. These values encode the function $f_i(\cdot)$ that the node computes, along with a sequence of integers $a_1, \ldots, a_n$ which identify the nodes that each of its n inputs are connected to. For example, a node that computes the `XOR` of the outputs of nodes 4 and 5 would have the corresponding tuple $(\texttt{XOR}, 4, 5)$. By convention, the graph's external inputs are given the integer identifiers $0, 1, \ldots, N_{\texttt{in}}$ where $N_{\texttt{in}}$ is the number of external inputs. The integer identifiers for the computational nodes then begin with $N_{\texttt{in}} + 1, N_{\texttt{in}} + 2$, and so on. In the graph of logic circuit elements shown in Figure 5.2, for example, the circuit's two inputs are labelled 0 and 1, respectively, while the four computational nodes are numbered

2, 3, 4, and 5. The full genome in CGP is formed by concatenating the tuples for all of the nodes, along with an additional $N_{\texttt{out}}$ integral values, where $N_{\texttt{out}}$ is the fixed number of external outputs that the graph computes. These values indicate which node's output values should be used as the graph's external outputs.

For simplicity, every node is assumed to have the same arity (number of inputs) n. If the function set contains functions of differing arity, n is set equal to the maximum arity. Then, if a node is assigned a function that takes less than n inputs, the extra inputs are simply ignored during the evaluation of the graph. This allows CGP to work with a fixed-length genome even as the arity of individual nodes changes over the course of evolution.

Standard CGP uses a simple point-mutation operator as its variation mechanism. At each generation, each element (gene) in an individual's genome independently undergoes mutation with probability m, where m is a fixed global mutation rate. Recall that a CGP genome contains two kinds of genes: function genes, which are categorical and assume values from some pre-specified function set, and connection genes, which assume integer values. If a function gene is mutated, its value is simply replaced with a randomly chosen function from the function set. Likewise, when a connection gene is mutated, its value is replaced with a randomly selected integer. The new integer is chosen so as to respect the graph constraints described above.

The effect is that mutation can alter connections among nodes, and it can alter the function that a node computes. Nodes are never added or removed, however, and the total number of nodes remains fixed.

CGP traditionally operates with a $(1 + \lambda)$-style, overlapping-generations population model. In this model, truncation selection is used to select the highest-fitness individual as a parent, after which λ offspring are generated by applying the mutation operator. The value of λ is typically a small integer (often 4). The fittest individual from the combined parent-offspring population is then selected as the parent for the next generation. Importantly, if no offspring individual has fitness superior to the parent, but some offspring has equal fitness to the parent, then the algorithm selects the offspring. This ensures that genetic

drift can occur in the absence of immediate fitness improvements.

In evolutionary algorithms in general, a $(1+\lambda)$ approach normally induces a notoriously greedy search strategy. Its use here stands in sharp contrast with the very large population sizes and diversity-preserving selection operators that are typically used with other genetic programming algorithms. Several studies of CGP's behavior have indicated that genetic drift plays a substantial role in allowing it to achieve high-quality search results with such a simple population model [Vassilev and Miller, 2000, Yu and Miller, 2001, 2002].

Methods

The initial set of tasks that I will use to investigate the hypotheses presented above is the `9-LOGIC` suite. This is a set $\mathcal{T}$ of nine basic logic function synthesis tasks that were originally used by Lenski et al. [2003] to demonstrate that complex tasks are sometimes easier to solve in conjunction with other tasks than they are to solve directly:

$$\mathcal{T} = \{\texttt{AND}, \texttt{AND\_N}, \texttt{EQU}, \texttt{NAND}, \texttt{NOR}, \texttt{NOT}, \texttt{OR}, \texttt{OR\_N}, \texttt{XOR}\}.$$

The details of these functions are given in Table 5.1. With the exception of `NOT`, all of them have an arity of 2, and they output exactly one Binary value.

Because these objective functions are relatively simple logic tasks, our primitive set (i.e., the functions that can be used on the nodes of the graphs evolved by CGP) consists only of $\{\texttt{NAND}\}$. Real-world applications almost always use a much less rudimentary primitive set, but I postpone the investigation of more complex objectives and realistic primitive sets to future studies.

Using the shared sub-graphs approach to multi-task optimization with CGP requires only very minimal modifications to the original algorithm. CGP has often been applied to synthesize graph structure solutions to single tasks with more than one output. The approach I take here is simply to partition the outputs, such that different output variables are assigned to different tasks. Specifically, I represent solutions to all nine tasks at once by

Table 5.1: The Boolean functions that make up the `9-LOGIC` suite.

Objective	Description
`AND`	x and y
`AND_N`	x and $\neg y$
`EQU`	(x and y) or ($\neg x$ and $\neg y$)
`NAND`	$\neg(x$ and $y)$
`NOR`	$\neg x$ and $\neg y$
`NOT`	$\neg x$
`OR`	x or y
`OR_N`	x or $\neg y$
`XOR`	(x and $\neg y$) or ($\neg x$ and y)

constructing a function over two Boolean inputs with nine Boolean outputs—one for each task.

To evaluate the fitness of a solution, I enumerate all of the entries of the truth table for each objective function. In general, truth tables grow exponentially in the number of inputs, and this is not a tractable way to evaluate the fitness of Boolean functions—but in this case each function has only $2^2 = 4$ truth table entries, so an exhaustive approach to fitness evaluation is feasible. I combine the `9-LOGIC` objectives into a single, composite objective based on the truth table shown in Table 5.2. During fitness evaluation, each individual produces $9 \cdot 4 = 36$ Boolean output values (one for each function and truth table row). The individual's fitness is equal to the fraction of these Boolean values that correctly match the target behavior.

Because each task has the same number of outputs, this approach to fitness calculation is equivalent to *averaging* the separate task-specific fitnesses to arrive at a single scalar fitness value. More complex methods for utilizing fitness information from multiple tasks are of course conceivable—perhaps based on lexicase selection, for instance [Helmuth et al., 2015]. In keeping with the theme of this chapter, however, I aim to start with the simplest

Table 5.2: Truth table representation of the composite objective function I use for multi-task Cartesian genetic programming.

x	y	AND	AND_N	EQU	NAND	NOR	NOT	OR	OR_N	XOR
F	F			T	T	T	T		T	
F	T				T		T	T		T
T	F		T		T			T	T	T
T	T	T		T				T	T	

approach to multi-task problem-solving, with the aim of creating a baseline that can be improved upon with future work.

In the standard (multi-task) CGP configurations discussed so far, every gene has an equal probability of being mutated. Call this the *constant mutation weighting* case. To approach Hypothesis 5.3, I introduce two additional weighted mutation schemes which avoid mutating genes that belong to a successful solution to a task.

In particular, consider an individual with a genome $\vec{x}$. I say that gene i *contributes* to a task $t \in \mathcal{T}$ if the gene encodes either an edge or a node that lies on a path from a graph input to one of the outputs for task t. Let $\mathcal{C}_i(\vec{x})$ denote the set of such tasks that the ith gene's circuit element contributes information to, and let $f_j(\vec{x})$ be the fitness of $\vec{x}$ on the jth task. Then I define *linear mutation weighting* as follows:

$$m_i^{\texttt{linear}}(\vec{x}) = a + \frac{b-a}{\|\mathcal{C}_i(\vec{x})\|} \sum_{j \in \mathcal{C}_i(\vec{x})} (1 - f_j(\vec{x})),$$

where $m_i^{\texttt{linear}}(\vec{x})$ gives the mutation rate for the ith gene of $\vec{x}$. Intuitively, this offers us a mutation rate that scales linearly on $[a, b]$ with the average value of $\vec{x}$'s fitness on the tasks $\mathcal{C}_i(\vec{x})$ that gene i contributes to.

I also test an *exponential mutation weighting* scheme:

$$m_i^{\texttt{exp}}(\vec{x}) = a + (b-a)\exp\left(\frac{c}{\|\mathcal{C}_i(\vec{x})\|}\sum_{j\in\mathcal{C}_i(\vec{x})} f_j(\vec{x})\right).$$

Here I likewise have a mutation rate that scales over $[a, b]$, except now scaling occurs exponentially with the average fitness of the tasks i contributes to. The scaling constant c is assumed to be negative.

Multi-task optimization approaches can be applied in a number of different ways. It thus especially important to be clear about the kind of performance goals I am pursuing in a particular experiment. One may wish to solve all the tasks in the task set, for instance, or one may be interested only in solving one particular target task (using information gleaned from other tasks strictly as a means toward that end). The objective I have set for this experiment takes the former approach: the goal is to find a solution for each of the nine tasks, assuming that none of them have been solved before the algorithm begins to run.

The usual means of measuring computational cost in an evolutionary computation context is to examine the number of fitness evaluations required to meet some stopping criterion. In CGP, however, the cost of evaluating an individual depends on the number of nodes that must be executed in their associated graphical phenotype. Because the size of the graphs is itself a tunable parameter, a fair comparison of CGP algorithms must take the relative graph sizes of competing methods into account. With this in mind, I take the product of the total number of nodes in an individual and the number of fitness evaluations the algorithm performs as my measure of cost:

$$\texttt{cost}(x) = n_c(x)n_r(x)\texttt{evals}(x)$$

I refer to the units of this quantity as *node-evaluations*.[2]

[2]I have used the total number of nodes $n_c(x)n_r(x)$ as my estimate of the cost of evaluating a single individual in CGP. In reality, however, efficient CGP implementations are able to avoid spending unnecessary

For the multi-task CGP algorithms, then, my measure of performance is the number of node-evaluations that are necessary on average to find a solution to each of the nine tasks. I compare these results against a similarly-configured single-task CGP approach that also aims to find solutions to each of the nine tasks, but without the use of any cross-task information sharing.

Parameter Tuning

Putting all these pieces together, I have one single-task CGP algorithm and three multi-behavior algorithms to test the performance of (a constant-mutation case, the linear case, and the exponential case). In order to achieve a fair comparison, I perform a parameter sweep over CGP's free parameters to select configurations of each algorithm that solve the suite of tasks in the least computational effort on average. Preliminary experiments indicated that there was no benefit to constraining the CGP graphs through the use of row constraints or the "levels back" parameter, so I fix the number of rows $n_r = 1$ and the levels back $l = n_c$. Following convention for these simple experiments, moreover, I fix the number of offspring λ to 4, yielding a $(1+4)$-style EA. For the weighted mutation operators, the minimum mutate rate a is fixed at 0.001, and the exponential scaling factor c is fixed to -3.

These decisions leave just two parameters unspecified, which are easy to set via a grid search: the constant mutation rate m (or, in the case of the weighted operators, the maximum mutation rate b), and the number of columns n_c. Because I set $n_r = 1$, the total number of computational nodes in the CGP graph is equal to n_c. All the parameters involved in the four algorithms are summarized in Table 5.3.

For each of the three multi-task CGP approaches, I conducted a two-dimensional parameter sweep, taking a sample of $n = 25$ runs for each parameter configuration, with each run carried out to a budget of 500,000 node-evaluations. From these results, which are visualized in Figure 5.3, I selected parameter configuration that was able to find a solution to each of

computation effort to calculate the state of unused "intron" nodes within the graph. Since these introns often make up a significant fraction of a CGP phenotype, this can yield non-negligible cost savings. The node-evaluations metric that I use here, then, should technically be viewed as an *upper bound* on performance.

Table 5.3: Parameters used by all four CGP algorithms under study.

Parameter	Value	Description
m	(Tuned)	Mutation rate (for constant weighting)
b	(Tuned)	Maximum mutation rate (for linear/exponential weighting)
n_c	(Tuned)	Number of columns in the CGP grid
n_r	1	Number of rows in the CGP grid
l	$= n_c$	Levels back
λ	4	Number of offspring per generation
a	0.001	Minimum mutation rate (for linear/exponential weighting)
c	-3	Exponential mutation weighting constant

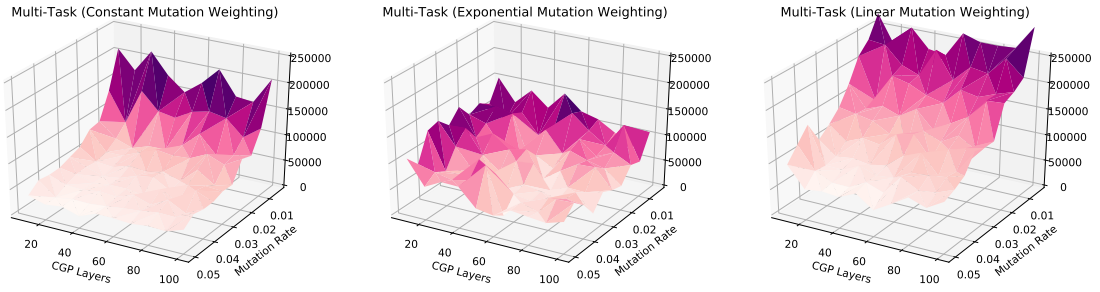

Figure 5.3: Parameter sweep results for the three **multi-task** CGP algorithms. Altitude indicates the number of node-evaluations that were necessary to find a solution to each of the nine tasks, averaged over 25 independent runs.

the nine tasks with the smallest number of node-evaluations on average.

For the single-task control, I tuned the parameters for each task independently. That is, rather than selecting one configuration that performs best over all nine tasks on average, I chose the nine configurations that performed best on each respective task. Because it allows for specialization on each task, this parameter tuning strategy is generous to the single-task approach. The idea is to ensure that our control sets a strong baseline to compare against. The results of the parameter sweep, again based on $n = 25$ runs per grid point and a budget of 500,000 steps, are illustrated in Figure 5.4.

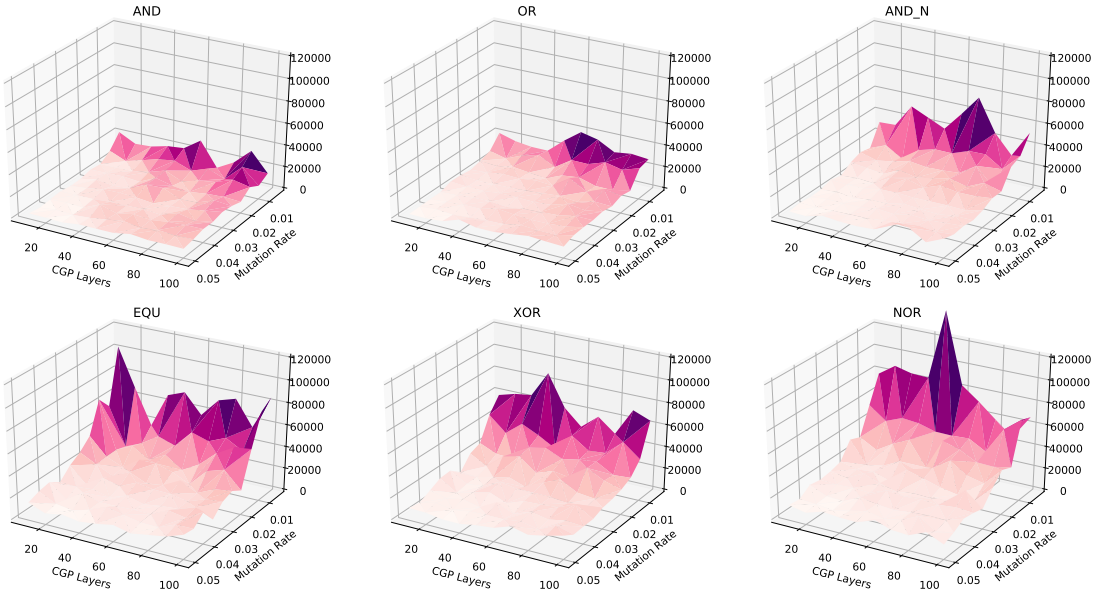

Figure 5.4: Parameter sweep results for the **single-task** CGP control, shown here for six of the nine objectives. Altitude indicates the number of node-evaluations that were necessary to solve the task, averaged over 25 independent runs.

As a result of this parameter-tuning process, some algorithms are configured to use different circuit sizes than others. This is why it is important to measure performance in terms of node-evaluations, rather than raw fitness evaluations.

Results

The algorithm configurations that were selected after parameter tuning are shown in Table 5.4. When I ran the tuned algorithms on the `9-LOGIC` suite, the individual tasks displayed a clear progression of difficulty. As can be seen in the distribution of convergence times for the single-task control, shown in Figure 5.5, by far the easiest task to solve is `NOT`, whereas `EQU`, `XOR`, and `NOR` are dramatically more difficult.

On average, I find that the multi-task approaches with constant and exponential weighting are able to find a solution to each of the 9 tasks after considerably fewer node-evaluations

Table 5.4: Parameter configurations that were selected after tuning. The total cost of solving all the tasks with the single-task approach are tabulated by summing the cost of solving each task independently.

Algorithm	Mutation Weighting	Task	Mutation Rate	CGP Layers	Avg. Steps	Avg. Node-Evals	Evals
Single-Task	Constant	NOT	0.015	10	1	40	4
Single-Task	Constant	NAND	0.045	10	7	280	28
Single-Task	Constant	OR_N	0.050	10	16	640	64
Single-Task	Constant	AND	0.045	10	31	1240	124
Single-Task	Constant	OR	0.050	20	25	2000	100
Single-Task	Constant	AND_N	0.050	20	50	4000	200
Single-Task	Constant	EQU	0.040	20	89	7120	356
Single-Task	Constant	XOR	0.050	40	38	6080	152
Single-Task	Constant	NOR	0.045	30	65	7800	260
Single-Task	Constant	Total			322	29200	1288
Multi-Task	Constant	All	0.050	40	116	18560	464
Multi-Task	Exponential	All	0.045	60	91	21840	364
Multi-Task	Linear	All	0.050	60	195	46800	780

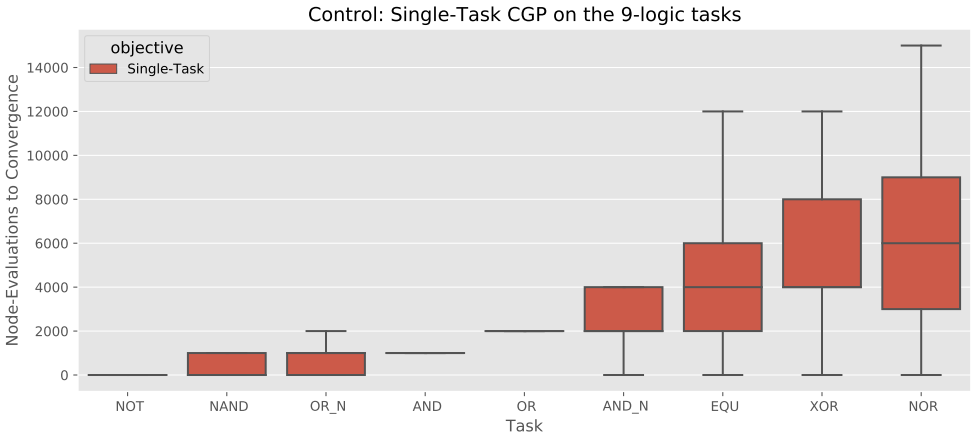

Figure 5.5: Distribution of the computational effort required by the **single-task control** across each of the nine tasks. A handful of outliers are omitted from this plot.

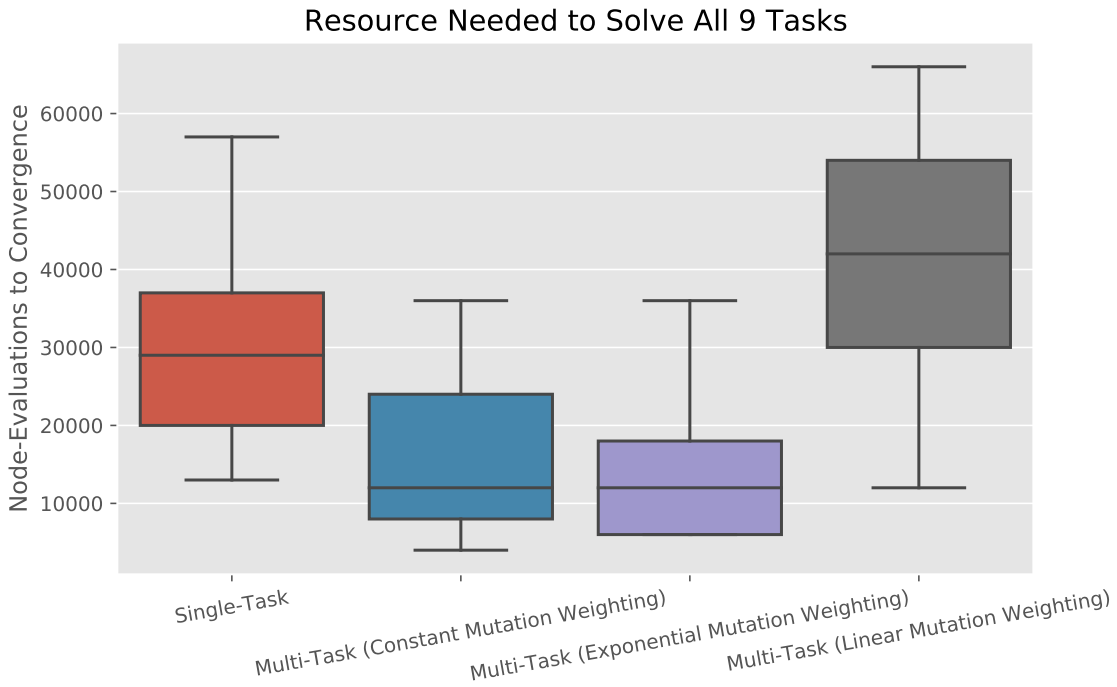

Figure 5.6: Distribution of the computational effort required by each algorithm to solve all nine tasks. A handful of outliers are omitted from this plot.

time than the single-task method (see Figure 5.6). The performance improvement these algorithms show over the single-task control is statistically significant at $p < 0.005$ (using a Wilcoxon rank-sum test to compare median performance). This result offers evidence that is consistent with a **positive reply to Hypothesis 5.1**: in the illustrative context of the `9-LOGIC` tasks, a multi-task approach to CGP does show improved efficiency over a single-task strategy.

This evidence implies that a simple point-mutation mechanism operating on a graph structure is by itself able to search for and find effective opportunities to reuse information across a moderately sized set of tasks.

The second hypothesis (Hypothesis 5.2) regards task interference. Two pieces of evidence emerge from these experiments that suggest that, despite the positive overall performance results, significant task interference does still occur in multi-task CGP.

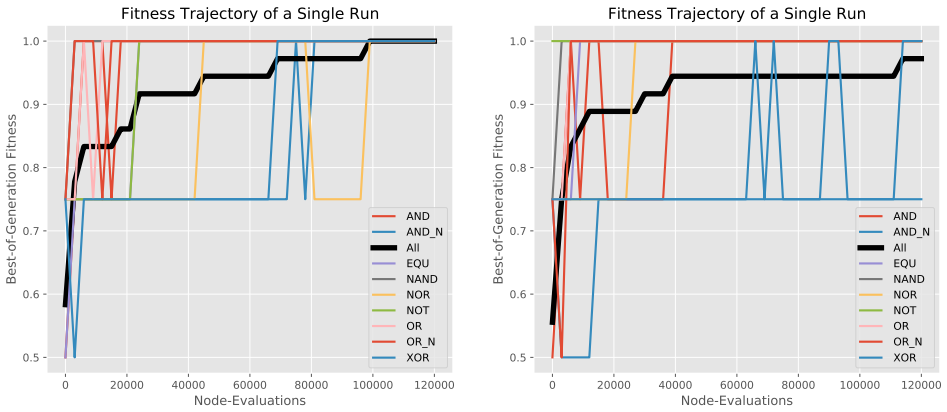

Figure 5.7: Best-of-generation fitness trajectories for two independent runs of the **multi-task CGP** method with **constant mutation**. The algorithm selects for improvements to composite fitness (solid black line), leading to improvements in the solutions to individual tasks (light lines).

First, when I plot the best-of-generation fitness for individual runs of multi-task CGP on the nine tasks, it becomes clear that the evolutionary process often achieves improvement on one task at the cost of a decrease in performance on some other task. Figure 5.7 shows two different runs of multi-task CGP with constant mutation weighting, with individual best-of-generation curves plotted for each of the nine tasks, and the composite objective function made up of all of the tasks (thick black line, denoted "All"). Because the algorithm uses truncation selection, the composite fitness is monotonically increasing. But many mutations still occur that lead to a decrease in the fitness of a specific subtask, balanced out by an increase in another. These movements show that a trade-off is operating, caused by the decision to require sub-graph material to be shared across multiple tasks.

The broader effect of the interference engendered by multi-tasking is visible in Figure 5.8. Here I break down the performance of the multi-task CGP methods by task. Each box indicates the number of node-evaluations that were required to achieve a solution to a particular task. The single-task results on each task are also shown for comparison. In general, the multi-task algorithms make for relatively poor single-task problem solvers. While they are

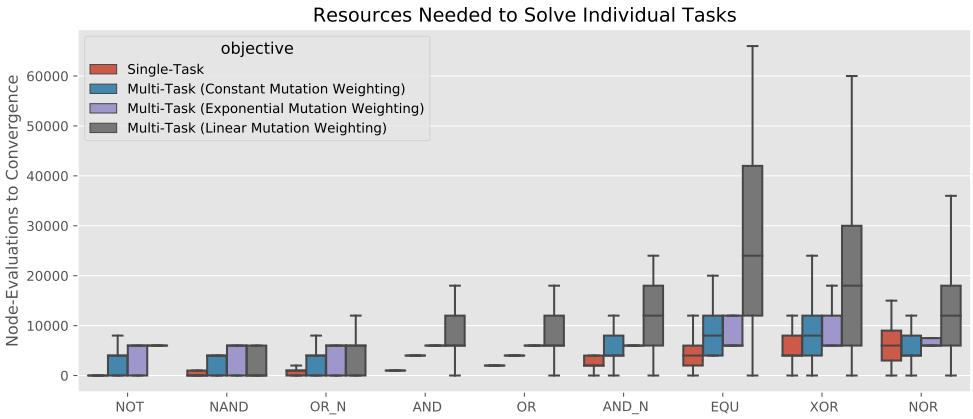

Figure 5.8: Distribution of the computational effort required by the **all four algorithms** across each of the nine tasks. Overhead from multi-tasking ensures that the single-task approach is the most effective way to approach individual problems. A handful of outliers are omitted from this plot.

usually able to solve *all nine tasks* with much less effort than it would take to solve each task independently, the overhead of multi-tasking does slow them down from the perspective of an individual task.

Considerably more study will be necessary to understand how task interference works in multi-task genetic programming systems of this kind. But at a high level, I find the data consistent with an **affirmative reply to Hypothesis 5.2**: I do observe task interference in multi-task CGP.

The median performance of the exponential mutation-weighting scheme is statistically indistinguishable from the constant mutation rate approach (Wilcoxon rank-sum, $p > 0.005$). As is evident from Figure 5.6, moreover, I observe a severe performance degradation over all other alternatives when applying the linear weighting approach ($p < 0.005$). The evidence remains **inconclusive, then, when it comes to Hypothesis 5.3**.

5.2 Representational Transfer for Real-Valued Optimization

In this section, I consider a different (more explicit) approach to learning representational information for the purposes of knowledge transfer. Whereas in section 5.1 I relied on the graph structure of an instance of an executable object to produce a reusable representation, in this section I consider real-valued optimization problems that do not involve executable objects. Instead, I explicitly learn a genotype-to-phenotype map for a class of problems, and show that the resulting representation transfers beneficially to new problem classes.

The research community often thinks of evolutionary algorithms as stochastic processes that search a genotype space $\mathbb{G}$ for solutions that optimize some fitness function f. Genotypes, however, are often abstract representations that can be conveniently manipulated by evolutionary operators, but that have no obvious meaning by themselves as solutions to a problem. Importantly, then—as I discussed in section 2.2.2—in many applications, fitness functions aren't defined over genotype space, but instead over a *phenotype space*, where candidate solutions are described in a fashion that is natural to the problem domain. A fitness function for traveling salesman problems takes graph tours as input, for instance, while a fitness function for real-valued optimization accepts vectors in $\mathbb{R}^n$. Here, it is necessary to specify *genotype-to-phenotype mapping* $\mathcal{R}$ (a.k.a. *representation*, or *encoding*) as part of the EA, which transforms individuals from their genetic representation (such as a bitstring) into a corresponding phenotype (the input to the fitness function).

At a high level of abstraction, the design decisions that go into constructing a simple (μ, λ)- or $(\mu + \lambda)$-style EA include choosing the reproduction and selection operators and their parameters, the values of μ and λ, and a stopping condition. In particular, however, the search behavior of the algorithm depends heavily on the choice of representation. This is because the representation determines the *pattern of phenotypic change* that occurs when reproductive operators are applied to individuals' genotypes. The selection of a good representation for the problem domain is commonly seen as one of the most important components of EA design—a view that is consistent with biologists' understanding of the

crucial role that genotype-phenotype relationships play in natural evolution [Gerhart and Kirschner, 2007].

In this work I present what is to my knowledge one of the first demonstrations of representation learning that has been created for evolutionary algorithms. I propose a relatively simple *meta-representation*—a representation for representations—that allows a restricted class of genotype-to-phenotype maps to be specified that convert fixed-length bitstrings into real-valued vectors. The familiar binary encoding is a special case of this general class of bitstring encodings. I find that I am able to use meta-evolution to learn a representation that makes it easy to solve a synthetic set of training, validation, and test problems. I also show that the resulting representation *generalizes* in the sense that it is useful on other problem instances from the same problem class and dimensionality it was trained on, and even on completely new problems from outside the class.

Methods

A genotype-to-phenotype mapping defines the effect that each gene has on the phenotypic representation of a solution. In traditional binary representations for real-valued optimization, each bit in the genome has an independent additive effect on exactly one phenotypic trait. For instance, the bitstring $\vec{g} = 0101$ can be converted into the real-valued phenotype $p = 5.0$ by interpreting the two non-zero bits as $2^0 = 1$ and $2^2 = 4$ and collecting the sum. In binary code, A) since the effect of each bit is purely additive, there are no non-linear interactions among genes with respect to the phenotype, and B) the magnitude of each bit's impact varies exponentially, allowing both small and large jumps to occur in phenotype space when bit-flip mutation is applied.

I construct a general class of bitstring representations that preserve these two properties. I also allow our mappings to have the property of *pleiotropy*—genes may influence multiple traits. Let a phenotype consist of a vector in $\mathbb{R}^n$. Each element of the phenotype vector is a *phenotypic trait*. I define a mapping $\mathcal{R} : \mathbb{B}^m \mapsto \mathbb{R}^n$ by assigning a weight w_{ij} to every possible gene-trait interaction. Each gene is further assigned a factor $s_i \in \mathbb{R}$, which serves

as an exponent that scales the magnitude of all the gene's phenotypic effects. Specifically, let $\vec{g} \in \mathbb{B}^m$ be a bitstring genome corresponding to an individual in the sub-EA. Then each trait in the corresponding phenotype vector $\vec{p} \in \mathbb{R}^n$ is determined by the equation

$$p_j = \sum_{i=1}^{m} 2^{s_i} w_{ij} g_i. \tag{5.1}$$

So when g_i is 1, a value proportional to w_{ij} is contributed to each trait p_j for all $j \in \{1..n\}$. When g_i is 0, the w_{ij}'s contribute nothing to $\vec{p}$. If the s_i's are uniformly distributed, then the magnitude of each gene's effect will be exponentially distributed. It is sometimes helpful to think of the parameters in Equation 5.1 as a set of m vectors of length n. The bits in the sub-EA bitstring $\vec{g}$ *select* which of the vectors $2^{s_i}\vec{w}_i$ are added together in the phenotype space $\mathbb{R}^n$ to collectively form a candidate solution:

$$\vec{p} = \mathcal{R}(\vec{g}) = \sum_{i=1}^{m} 2^{s_i} \vec{w}_i g_i. \tag{5.2}$$

Each mapping of this type can be seen as a linear transformation from the space of bitstrings to $\mathbb{R}^n$, and any linear transformation between these spaces can be encoded by an appropriate choice of weights. Accordingly, I refer to mappings of this type as *linear pleiotropic encodings*.

Since I have allowed gene effects to be pleiotropic, these mappings bear some similarity to a representation once explored by Altenberg [Altenberg, 1994]. Equation 5.1 can also be interpreted as a feed-forward neural network with linear activation functions (Figure 5.9). Since I am limiting our choice of mappings to linear transformations that take binary inputs, however, the expressive power of this class of representations is more limited than the complex neural network mappings studied by Simões et al. [Simões et al., 2014].

Fully $m \cdot (n+1)$ real-valued constants are required to specify the parameters that make up a single linear pleiotropic mapping $\mathcal{R}$. The question I pose here is whether it is feasible to

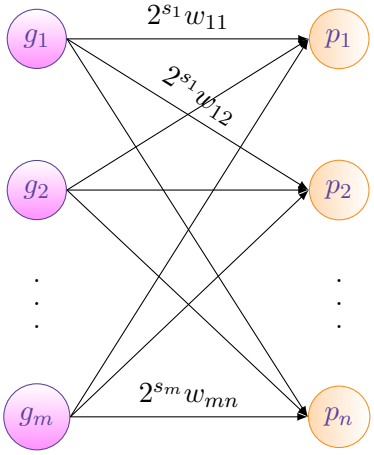

Figure 5.9: A linear pleiotropic encoding can be interpreted as a feed-forward neural network, in which genes have an additive effect on phenotypes.

automatically search this high-dimensional parameter space for a mapping that is effective on a class a problems.

For the purpose of meta-evolution, I group the parameters of a linear pleiotropic encoding into a sequence of m tuples of the form $(s_i, \vec{w}_i) \in \mathbb{R} \times \mathbb{R}^n$. As is made clear from Equation 5.2, the values in the ith tuple of a meta-individual completely define the effect that the ith bit of a sub-EA individual has on the phenotype. This sequence of tuples forms the meta-level genome, and I treat each tuple as a meta-level gene.

I use a generational evolutionary algorithm with two-point crossover to tune the values of the tuples $(s, \vec{w})$. Each tuple in a child has a chance of being selected for mutation. When the ith tuple is selected for mutation, I add values drawn i.i.d. from a normal distribution to both s_i and to each element of $\vec{w}_i$, unless otherwise specified. The number of genes m that the sub-EA uses to represent solutions in $\mathbb{R}^n$ is a free parameter that must be chosen *a priori*. I opt to use 20 genes per dimension, simply because traditional bitstring encodings typically need about that many in order to have a sufficiently fine-grained ability to cover the phenotype space. The remaining design decisions I use to implement the meta-EA are detailed in Table 5.5.

The exponential scaling factors s_i do not add to the expressive power of the meta-representation scheme. Any linear pleiotropic mapping defined by a sequence of tuples $(s_i, \vec{w}_i)$ can be represented by an equivalent mapping $(1, \vec{v}_i)$, where $v_{ij} = 2^{s_i} w_{ij}$. The merit of explicitly including the s_i's is that it changes how the meta-EA's mutation operator affects vectors that differ exponentially in length. When s_i is large, a small Gaussian perturbation of the values in $\vec{w}_i$ has a large effect on the vector $2^{s_i}\vec{w}_i$. When s_i is small, perturbing the values of $\vec{w}_i$ has a small effect, allowing the shorter vectors to be fine-tuned.

The fitness I assign to a candidate mapping $\mathcal{R}$ should reflect $\mathcal{R}$'s ability to serve as a useful component of a sub-EA's search heuristic. One way to measure this is to plug $\mathcal{R}$ into a sub-EA, and then run the sub-EA several times on an objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$ and see how it does. This raises the concern of over-fitting, however: if the algorithm succeeds in adapting an encoding $\mathcal{R}$ that can easily solve f every time, has it found a good search heuristic, or has it simply memorized the location of f's global optimum by encoding it into $\mathcal{R}$? If my goal is only to optimize a single difficult function f, then I don't care about over-fitting. If one wishes to reuse $\mathcal{R}$ to solve new instances from a class of problems, however, one needs to encourage the meta-EA to find a more general solution.

Here I assume that I have at our disposal a *training set* composed of several examples from a class of problems, all of the same dimensionality. Such classes arise frequently in algorithmic applications—traveling salesmen problems with 20 cities, for instance, or room-scheduling problems with 20 rooms. To assign fitness to a mapping $\mathcal{R}$, I plug $\mathcal{R}$ into a sub-EA, and run the sub-EA once on each problem in the training set. The average best fitness the sub-EA achieves on the training problems becomes $\mathcal{R}$'s fitness.

For this study, I synthesize a problem domain that is diverse enough to demonstrate the ability for the meta-EA to learn, but that is still relatively simple to analyze. I define the *translation class of* f, $\mathcal{T}(f)$, as the set of all functions created by applying an arbitrary offset to f in its input space within some bounds. Every element of $\mathcal{T}(f)$ has the same shape, but the location of the global optimum varies. As such, it is not sufficient to construct a representation that memorizes the location of the optimum.

Table 5.5: Default configuration used in the meta-EA experiments, except where otherwise specified.

Component	Meta-EA	Sub-EA
Type	$(\mu + \mu)$	$(\mu + \mu)$
Pop. Size (μ)	50	50
Gene Type	$(s, \vec{w}) \in \mathbb{R} \times \mathbb{R}^n$	$b \in \mathbb{B}$
Genes/dimension	20	20
Initialization Bounds	$s \in [-4, 4]$, $w_i \in [-1, 1]$	n/a
Parent Selection	Binary tournament	Binary tournament
Reproduction	2-point crossover	2-point crossover
Mutation	Gaussian perturbation of all values ($\sigma = 1$)	Bit-flip
Mutation Rate	$1/L$ chance per gene	$1/L$
Mutation Bounds	Soft	n/a
Objective	Mean best fitness of 10 sub-EA runs	$\mathcal{T}$(`Valley`) with no rotation
Stopping Condition	500 generations	40 generations without improvement

For the objective f I choose the "valley objective" defined in Bassett [2012]:

$$f(\vec{x}) = 10\delta(\vec{x}, L) + \|\vec{x} - \vec{o}\|, \tag{5.3}$$

where L is some line that passes through the optimum $\vec{o}$, and $\delta(\vec{x}, L)$ denotes the distance between $\vec{x}$ and the nearest point on the line. In two dimensions, this function defines a valley with linearly sloping sides (via the first term) and a slight conical gradient that prevents its floor from being flat (via the second term) – see Figure 5.10. I define the translation class $\mathcal{T}$(`Valley`) such that random translations are applied within the bounds $[-15, 15]$ along each dimension.

Hypotheses

I have defined a meta-evolutionary scheme for representation learning and a synthetic problem class to exercise it on—namely the translation class $\mathcal{T}$(`Valley`). I now investigate the meta-EA's ability to learn a genotype-phenotype mapping on a set of training instances that

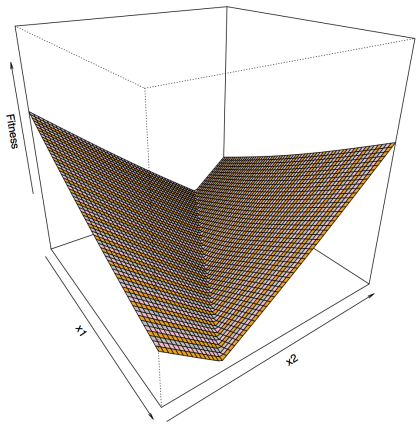

Figure 5.10: The valley objective.

is useful for solving new problem instances.

First, I have made a number of design decisions in the implementation of the meta-EA itself, not all of which may be optimal. In particular, I included an extra scale parameter s_i in each gene of the meta-representation. Is this useful? Or is it superfluous? This is the first hypothesis in this section:

Hypothesis 5.4. *It will be easier to improve the training fitness of a mapping $\mathcal{R}$ on the translation class $\mathcal{T}(\texttt{Valley})$ if one mutates both the magnitude and the elements of each vector in the mapping than if one only mutates one or the other.*

Next, I predict that one ought to be able to learn a genotype-to-phenotype map that allows us to effectively solve arbitrary instances of $\mathcal{T}(\texttt{Valley})$. This is serves as a proof of concept for this approach to learning representations:

Hypothesis 5.5. *Let $\mathcal{R}$ be a linear pleiotropic encoding evolved to solve instances of the translation class $\mathcal{T}(\texttt{Valley})$. Then $\mathcal{R}$ will perform competitively against traditional bitstring encodings when applied to new instances of $\mathcal{T}(\texttt{Valley})$.*

Provided that I succeed in learning a good mapping for $\mathcal{T}(\texttt{Valley})$, it would be useful if

I could say something about *why* the learned linear pleiotropic representation works, rather than just *whether* it works. I may be able to determine a pattern in the resulting map that corresponds to features of the landscapes it was trained on:

Hypothesis 5.6. *The learned linear pleiotropic encoding for $\mathcal{T}(\texttt{Valley})$ will contain a concentration of vectors that are aligned with the bottom of the valley.*

Now, one might expect that since I am tailoring the map to a specific problem class, that it will only be useful for solving instances of that class. In some cases, however, the implicit search heuristic encoded by $\mathcal{R}$ may be of more *general* use, much like Gray code is useful on a wide diversity of problems. Stated differently, the information the algorithm learns about how to solve instances of $\mathcal{T}(f)$ may *transfer* to other problem classes:

Hypothesis 5.7. *Let $\mathcal{R}$ be a linear pleiotropic encoding evolved to solve instances of the translation class $\mathcal{T}(\texttt{Valley})$. Then $\mathcal{R}$ will perform comparably to traditional bitstring encodings on new problems that do not belong to $\mathcal{T}(\texttt{Valley})$.*

Results

As specified in Table 5.5, I ran 50 independent runs of a meta-EA with a population size of 50 for 500 generations. Each run is initialized with an independent sample of 10 training problems from $\mathcal{T}(\texttt{Valley})$. The fitness of a mapping $\mathcal{R}$ is defined as the average best fitness that the sub-EA achieves on the 10 training problems. The sub-EA stops when it has gone 40 generations with no improvement in its best fitness. I found that setting the meta-level population much lower than 50 caused it to converge prematurely, while increasing it beyond 50 did not measurably improve the final best training fitness (not shown).

It is not clear *a priori* whether it ought to be beneficial to mutate the s_i's and weights w_{ij}, or whether only one or the other need to vary for learning to occur. I ran experiments for four different adaptation schemes (Figure 5.11): A) one in which the scale factors (which control the magnitudes of the vectors) were all fixed at a value of 1, while the weights w_{ij}

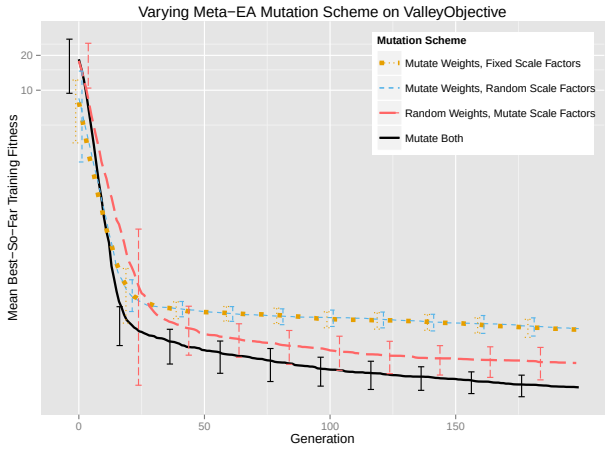

Figure 5.11: Meta-level fitness trajectories for different mutation methods. Error bars denote standard deviation.

were mutated, B) the s_i's were randomly initialized and then held constant while the weights were mutated, C) the s_i's were mutated while the weights were held constant at their initial random values, and finally D) both the s_i's and the w_{ij}'s were allowed to mutate. The results **confirm Hypothesis 5.4**: mutating both is more effective than holding one fixed. For the remainder of the paper all experiments were conducted with both types of mutation enabled (D).

To determine whether the algorithm begins to over-fit as evolution proceeds, an additional 20 instances of $\mathcal{T}(\texttt{Valley})$ are reserved as a validation set. Each run of the meta-EA took several hours to execute on a sequential processor, even though each sub-EA run took less than one second—so the meta-EA represents a substantial investment of resources. Because measuring the validation fitness of the population is expensive and does not affect evolution, I only take validation measurements every 20 generations.

Figure 5.12 shows the improvement in the mean training fitness across the 50 runs for the meta-EA, compared an experiment where the meta-EA was replaced by a random search algorithm. The validation curve shows the mean fitness of the *best-of-generation* individual with on the validation set. In general, this is a different individual than the individual with

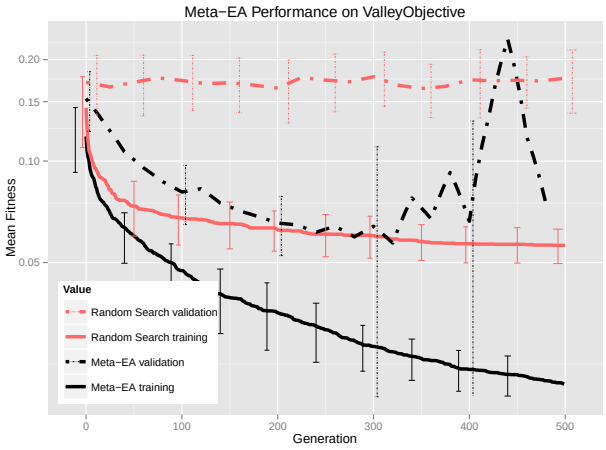

Figure 5.12: Training and validation fitness for 50 independent runs of the meta-EA. Error bars denote standard deviation on the mean. Reported is the mean best-so-far fitness on the training set and the mean best-of-generation fitness on the validation set.

the best-so-far training fitness (sometimes the individual with the best training fitness in a generation is not the individual that performs best in terms of validation fitness). The best-of-generation validation fitness, however, consistently improves for about the first two hundred generations before becoming unpredictable late in the run (with both mean fitness and its variance increasing).

My goal in the experiment in Figure 5.12 is to select a mapping that will perform well on instances it hasn't been trained on. I obtained a high-quality mapping by selecting the individual that had the best *validation* fitness of the run among the times that validation fitness was measured. So, I do not use information from the validation set during evolution, but at the end of the run I use the validation set to choose which individual to select as our trained genotype-to-phenotype mapping $\mathcal{R}^*$. This procedure helps avoid selecting mappings that are over-fit (in the sense of being unable to generalize to new instances that were not part of training). To evaluate the performance of the mapping chosen from the validation results, I construct a tertiary *test set* by taking 100 more random instances of the translation class $\mathcal{T}(\texttt{Valley})$. I chose a learned encoding $\mathcal{R}^*$ from a typical run of the meta-EA, plugged

it into a sub-EA, and ran the sub-EA once on each function in the test set.

The left-hand side of Figure 5.13 shows the results, compared against genetic algorithms that use Gray code and a standard binary encoding. The only difference between the three algorithms is the encoding method used. I find that the learned encoding's average performance is statistically indistinguishable from Gray code's performance at solving new instances of $\mathcal{T}(\texttt{Valley})$, and that it performs better than the standard encoding.

Now, it is well known that most common forms of recombination struggle with rotated problems if the rotation introduces interactions between variables that aren't present when the problem is aligned with the axes [Salomon, 1996]. This is known as epistasis, and is a well studied problem in EAs. To see how my learned encoding $\mathcal{R}^*$ handles epistasis, I applied a $\pi/6$ radian rotation to every problem in the tertiary test set. The result is shown in right-hand side of Figure 5.13. The performance of the traditional encodings are very poor on the rotated version of the valley landscape. Because the mappings the meta-EA learns are pleiotropic, however, the sub-EA often alters more than one trait at a time when a single bit is flipped. As a result, it can deal with epistasis much better than the traditional encodings are able to. So I have **confirmed Hypothesis 5.5** (the learned encoding performs competitively), and I have demonstrated a simple form of transfer: the learned encoding is able to generalize to solve problems with a rotation that it was not trained on.

My third prediction was that the mappings I trained on the instances of $\mathcal{T}(\texttt{Valley})$ would hold a particular signature: I anticipate that the weight vectors $2^{s_i}\vec{w}_i$ will be pointed in a direction that aligns with the floor of the valley (the line L in Equation 5.3).

I applied a $\pi/6$ radian rotation to the valley objective, and created a translation class of rotated valleys $\mathcal{T}_{\pi/6}(\texttt{Valley})$. I ran 50 independent runs of the meta-EA with training and validation sets drawn from $\mathcal{T}_{\pi/6}(\texttt{Valley})$. Then I took the mappings $\mathcal{R}^*$ with the best validation fitness from each of the 50 runs, and analyzed the 40 vectors that made up each mapping, for a total of 2,000 vectors.

The results (Figure 5.14) indicate that a large proportion of the vectors are indeed aligned

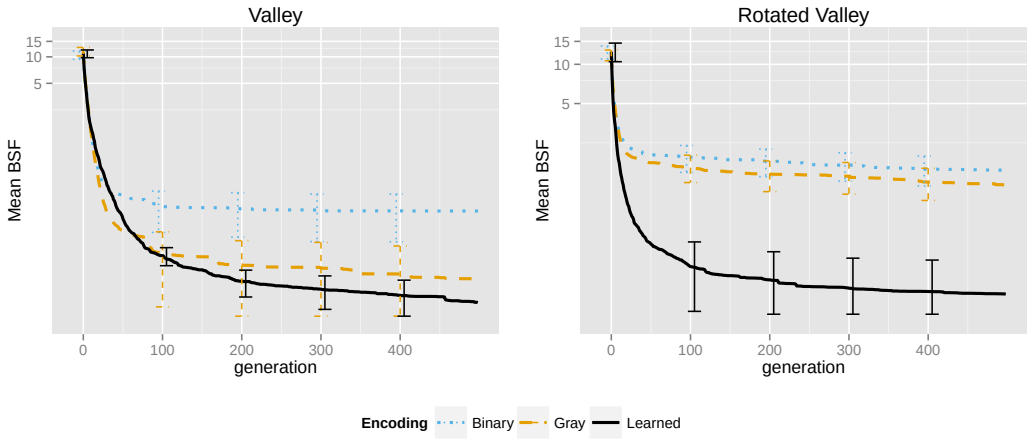

Figure 5.13: Comparing the performance of a learned representation to Gray code and standard binary encoding on instances of the non-rotated (**Left**) and rotated (**Right**) valley function. Error bars indicate the 95% confidence interval on the mean over the 100 test problems.

with the valley floor (indicated by the dashed red line). This **confirms Hypothesis 5.6**: a good pleiotropic mapping is one that permits mutations that take it along the valley floor. No such rule seems to apply to the larger vectors, however. This could indicate that the learned bias is more important during the exploitation phase of the search process than exploration.

I have shown that the mappings that the algorithm was able to learn on the translation class of the valley objective perform as well as Gray code on other instances of the same problem class. But how does a learned encoding perform on problems unlike anything it has seen before, such as multi-modal landscapes?

I took the same mapping $\mathcal{R}^*$ that I trained on $\mathcal{T}(\texttt{Valley})$ above, and applied it to new test sets of 100 instances each from the sphere, Rastrigin, Rosenbrock, and Ackley functions. I then applied a $\pi/6$ radian rotation to those instances, and ran the learned encoding on those too, for a total of 8 new translation classes. The results show that the learned encoding performed as well as or better than Gray code on all 8 classes (Figure 5.15), even though $\mathcal{R}^*$ was not trained on functions of this sort. This preliminary result **supports Hypothesis 5.7**.

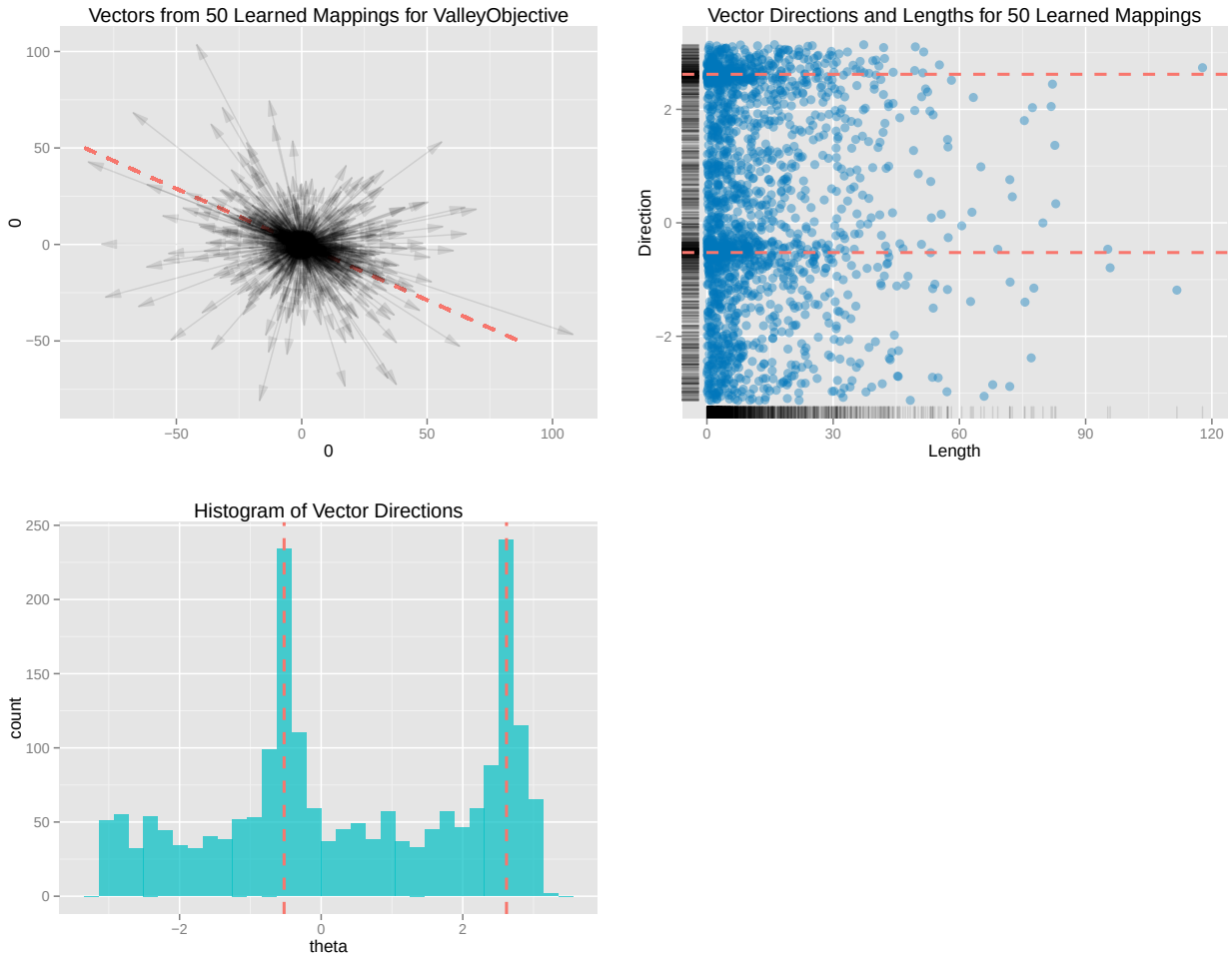

Figure 5.14: Three ways of summarizing the 2,000 vectors that make up 50 learned mappings that were trained on $\mathcal{T}_{\pi/6}(\texttt{Valley})$. The dashed line indicates the angle of the valley floor, which lies at a $-\pi/6$ rotation from the axis in these figures.

Table 5.6: Summary of the hypotheses tested in Chapter 5.

Hypothesis	Description	Result
	Multi-Task Evolution via Shared Layers	
Hyp. 5.1	Multi-task CGP reduces computational cost.	Supported
Hyp. 5.2	Task interference occurs in multi-task CGP.	Supported
Hyp. 5.3	Biasing mutation in multi-task CGP improves performance.	Unsupported
	Representational Transfer for Real-Valued Optimization	
Hyp. 5.4	Better pleiotropic rep. learning when mutating scale parameters.	Supported
Hyp. 5.5	Learned pleiotropic encodings rival traditional encodings.	Supported
Hyp. 5.6	Learned pleiotropic vectors align with the valley floor on the valley problem.	Supported
Hyp. 5.7	Learned pleiotropic encodings transfer to new classes.	Supported

5.3 Conclusions & Discussion

The hypothesis that I tested in this chapter are summarized in Table 5.6.

Multi-Task Cartesian Genetic Programming

I've shown how a shared sub-graphs approach to multi-task optimization can allow us to efficiently find solutions to each of a set of target tasks. A very simple genetic programming algorithm like CGP, when adapted to serve as a multi-task optimization system, is sometimes capable of improving our ability to solve at least nine tasks at a time. A considerable amount of overhead and/or task interference is involved, however, so it remains to be seen how well this sort of approach might be able to scale to larger numbers of tasks, for instance. And while these results show the promise of a multi-task approach for applications where our goal is to quickly solve all of a number of tasks simultaneously, it's not clear how useful this strategy might be in cases where one wishes to make use of information that is available from (a large set of) multiple auxiliary tasks as a sort of catalyst or means to the end of enhancing our ability to solve one particularly tricky target task.

Linear Pleiotropic Encodings

I have introduced a new 'representation for representations'—the linear pleiotropic encodings—to facilitate learning genetic representations for classes of real-valued optimization problems. Although this brief study was confined to simple, synthetic problem classes, I have shown a proof of concept that learned pleiotropic representations can preform competitively with traditional bitstring encodings, that they are robust to rotations of the landscape, and that in some cases they may even be useful on problem classes that they were not trained for.

The results of my final experiments in particular indicate that not only does this representation learning scheme avoid over-fitting to the specific problems I train it on, but it also displays a remarkably general-purpose problem-solving ability, akin to Gray code. I conjecture that this robustness may be a result of the limited expressive power of linear pleiotropic encodings, which may prevent them from being over-fit to the environment they evolved for.

Overall, my results suggest that, while meta-evolution remains a costly way to design algorithms, representation learning may not be as intractable as is commonly believed. The general approach I have presented here is not necessarily specific to real-valued problems, either – pleiotropy is a versatile concept, and a similar scheme could be adapted to, for instance, pseudo-Boolean functions.

A limitation of the present work is that the meta-EA requires a fixed number of genes m and number of phenotypic dimensions n. In contrast to binary or Gray codes, which scale easily, a linear pleiotropic mapping learned for one dimensionality n cannot be directly used on a problem with a different number of dimensions, nor can the number of bits be adjusted as needed without learning a new encoding from scratch. Additionally, a possible threat to the validity of the conclusions here is that all of my experiments on tertiary test sets of the various problem classes were conducted with one learned mapping that was the result of a single run of the meta-EA. While I believe these results are representative of the meta-EA's typical learning behavior, future work will need to confirm these conclusions with statistical

rigor.

On a more general level, part of the reason that meta-evolution is so expensive is that there is no obvious way to do credit assignment. If one mapping is better than another, which tuples in the meta-representation are responsible for the difference in performance? Soria Alcaraz et al. have demonstrated the usefulness of using measures of 'effective fitness' to predict which components of hyper-heuristics are the most promising as evolution progresses [Soria Alcaraz et al., 2014]. One could conceive of incorporating a similar credit-assignment mechanism into the evolution of EA representations.

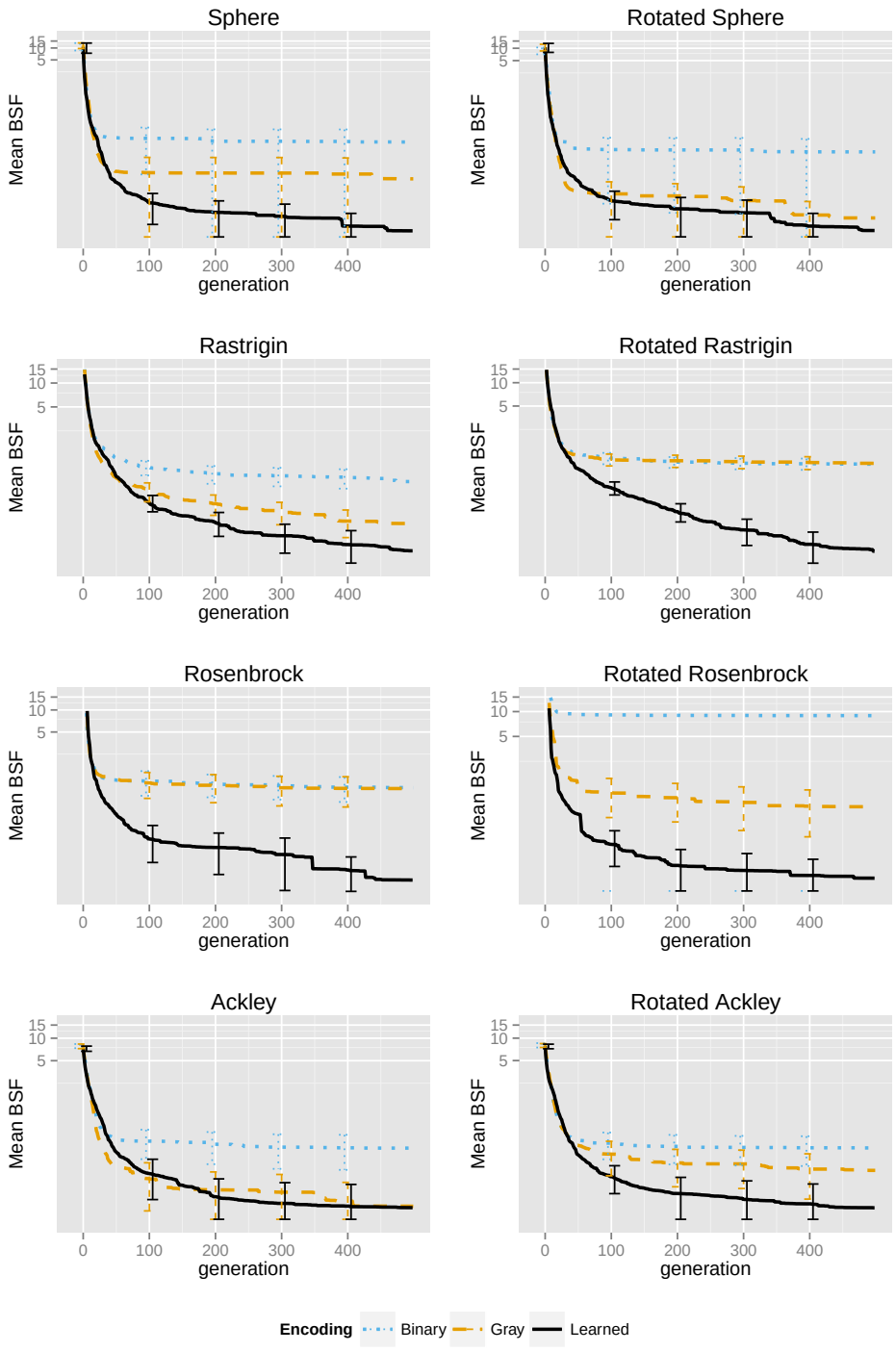

Figure 5.15: When I train a pleiotropic encoding on the class of valley objectives and then use it on instances of other functions, it performs as well as or better than a traditional binary encoding. Shown here is the sub-EA BSF averaged over 50 instances of each problem class. Error bars denote 95% confident intervals on the mean.

Chapter 6: Discussion

At a high level, in this dissertation I have presented research into how evolutionary algorithms can be used in more efficient ways to solve problems. In Chapter 1 I distinguished between two sides of the EA efficiency coin: on one side are methods that increase the number of queries per unit time that can be made to an objective function, while on the other are methods that reduce the number of queries that are needed to solve a problem.

6.1 Asynchronous Parallelism

6.1.1 Contributions

In my research, the first side of the coin has led me to asynchronous parallelism and the asynchronous steady-state model of evolution. When I began using and studying ASEAs, the community was uncertain about a number of important aspects of their performance: how much speedup do they produce in practice? Do they exhibit a strong evaluation-time bias? Do they create the appearance of efficiency only to trade it off with excess computation? I have contributed answers to these questions in the form of two theorems and some twenty-two tests of empirical hypotheses. These results are summarized in detail in the Conclusions & Discussion section of Chapter 3.

The main contribution of my work on speedup in ASEAs (Research Question 1) has been **an analytical model for predicting the throughput improvement** that they exhibit over a generational algorithm. This impact of this result is that it provides engineers and project managers with better tools for planning evolutionary algorithm projects. Knowing in advance, for example, that an ASEA will perform much more efficiently on a given application may allow engineers to plan to use these algorithms from the beginning

of a project—rather than switching to them after wasting months of effort on less efficient algorithms. On large super-computing clusters, moreover, users must often apply in advance for a budget of compute time to use for a project. Being able to estimate how much of that budget an ASEA will consume in a given experimental iteration can inform project planning and budget requests. More broadly, my study of speedup in ASEAs has shown that ASEAs are **generally good problem solvers overall**: these optimization strategies are practical tools that promise to be useful on many kinds of applications with different fitness landscapes and evaluation-time properties—without exhibiting any severe pathological "gotchas" that make them inappropriate for many applications. The most sensitive aspect of ASEA behavior is initialization, so I have contributed a study of **asynchronous initialization strategies**, demonstrating how three commonly used strategies impact the behavior and performance of EAs under different assumptions.

My other major contribution in this area has been to **alleviate concerns over evaluation-time bias in ASEAs** (Research Question 2). Through a suite of different experiments, I have characterized the impact that evaluation-time properties have on ASEAs' dynamics and ability to solve problems, and have found that the effect is weak and largely limited to initialization effects. The immediate impact of this work is that because "evaluation-time bias is no big deal," the primary reason practitioners have had to be concerned about their performance is mitigated. I have found at the same time, however, that the **quasi-generational EA (QGEA) is a poor replacement** for ASEAs—it does not avoid evaluation-time bias, and it essentially has a broken evolutionary feedback loop that dramatically slows down optimization. This result suggests that while hybrid asynchronous-generational models of evolution may be a promising way to design next-generation parallel EAs, future researchers must pursue these chimeras carefully.

My final contribution to asynchronous parallelism has been to show that, in settings where long-evaluating solutions tend to be of higher quality (Research Question 3), the **Selection WhilE EvaluaTing (SWEET) operator offers a promising approach to tighten the evolutionary feedback loop** and preventing the generation of useless "busy

work" which wastes resources.

6.1.2 Discussion

Overall, my research on asynchronous EAs—and the ASEA in particular—has convinced me that these algorithms are the future of the field where applications involving computationally intensive, distributed optimization are concerned. When I speak with my colleagues at major tech companies throughout the U.S., they immediately understand the importance of asynchrony and of minimizing idle resources on large compute clusters. Often, however, they raise concerns about certain other inefficiencies that centralized, asynchronous communication might lead to—such as excess computation that serves no useful purpose, or evaluation-time bias. I believe that my results in this chapter largely put these concerns to rest, and suggest practical strategies for managing the concerns that remain.

Ultimately, however, efficient parallelization by itself can only go so far toward scaling evolutionary algorithm to tackle increasingly challenging search and optimization problems. The various simulation and parameter-tuning problems that motivated me to begin this study remain computationally intensive regardless of how many instances of a simulation are kept running in parallel by an optimization algorithm. And if an algorithm requires orders of magnitude more fitness queries to solve than are feasible to compute in a given amount of time, then parallelization may hardly make a dent in certain especially difficult problems.

6.1.3 Future Work

My discussion of asynchronous EAs in section 3.6 raised several avenues for future work in this area.

One is to investigate hybrid approaches that use synchronous evaluation for initialization of the population, and then asynchronous steady-state dynamics for the remainder of evolution. This may offer a means of significantly reducing evaluation-time bias, while also avoiding the pathological behavior that I have shown (in section 3.2.2) some asynchronous

initialization methods can exhibit.

I have also introduced two paths to advancing the theory of ASEAs. One possibility for future work would pick up where my analytical analysis of throughput in section 3.3.1 left off—extending the bounds on asynchronous speedup to other distributions of evaluation times. If future work can prove similar bounds for distributions that are likely to appear in practice, the result would be more useful tools for planning projects that involve distributed optimization. Second, since the community still lacks a solid analytical understanding of evaluation-time bias, I would like to pursue a deeper mathematical analysis of this aspect of ASEA dynamics in future work.

And finally, since most of the ASEA results in this dissertation involved synthetic problems that targetted particular empirical questions—I should like to continue applying ASEAs to a variety of applications, collecting data on where they perform best and where specialized configurations (such as SWEET) are beneficial.

I gave more discussion of these points in section 3.6.

6.2 Evolutionary Knowledge Transfer

6.2.1 Contributions

Evolutionary knowledge transfer (EKT) is a relatively new and speculative approach to using optimization to solve complex problems. One contribution I have made in this dissertation is **a three-part analysis of theoretical motivations for knowledge transfer**—articulating how prior work falls into three motivational schools: the identical-elements principle (IEP), shared causal principles, and innovation engines. I believe these three lenses will help the research community to clarify the goals, opportunities, and challenges that this new field presents.

My primary contribution to EKT has been to highlight the importance that a theory of transferability within problem classes has for the effectiveness of transfer optimization (Research Question 4). I have done this by **proving no-free-lunch theorems for transfer**

optimization, by **proving the first asymptotic run-time result for transfer optimization**, and by engaging in related experiments that demonstrate the degree to which transfer is possible in different **benchmark problem classes**. As a side effect of this work, I have shown that **sequential population seeding offers an effective approach to EKT**. Population seeding is a very simple way of representing and transferring knowledge—and so has rarely been used as a knowledge transfer strategy—but in some problem classes, this simple mechanism is sufficient for carrying useful heuristic knowledge from one problem to another.

I have also contributed an initial demonstration of using landscape features of optimization problems to estimate their similarity (Research Question 5), and in particular I have shown that **combining multiple measures of problem similarity can improve transfer prediction**. I hope that this work will continue to inspire novel solutions to the source-selection problem.

A significant contribution of this dissertation has been to introduce the concept of **many-source sequential transfer** (Research Question 6) and to demonstrate **many-source population seeding as an effective knowledge-transfer strategy**. Algorithms of this kind can serve to widen the applicability of EKT by enabling a greater degree of human-machine teaming in the search for useful heuristic knowledge.

And finally, I have introduced **one of the first representation-learning methods for EAs and EKT**, demonstrating the feasibility of a meta-evolutionary approach to evolving and reusing solution representations (Research Question 8). This contribution is important because representation learning has been very rarely studied for EAs, despite the powerful opportunity it offers to encode heuristic knowledge. I have also contributed a study of **how executable structures can implicitly reuse subcomponents** while evolving monolithic solutions that solve multiple tasks simultaneously. Approaches of this kind have a lot in common with parameter-sharing methods of multi-task learning with neural networks, but we are only beginning to understand the potential they have to assist in other kinds of optimization tasks (Research Question 7).

I have discussed the details and practical importance of all of these contributions in sections 4.5 and 5.3.

6.2.2 Discussion

Most areas of artificial intelligence (AI)[1] continue to struggle today with the widespread problem of generalization and robustness [Schölkopf et al., 2021]. If algorithms were available that could generalize better across problems and domains—that could learn from experience, and that could transfer and adapt that experience to other, distantly related problems—then the AI-engineering workflow would look quite different. The practice community could move away from building or training expensive, bespoke algorithms for each application that often only the most well-funded institutions have the resources to develop, and instead draw on a rich community of experience to rapidly generalize on past successes to meet new needs. AI would be easier to scale to large, diverse organizations that face a multiplicity of domain-specific and computationally expensive challenges (a great many of which live in a "long tail" of applications that are not adequately met by high-profile developments in machine learning). And problems that are very difficult to solve at all—whether because of their inherent complexity, lack of human understanding, or lack of suitable data—could become easier to tackle.

Thanks in no small part to the proliferation of neural networks as the dominant model in machine learning, knowledge transfer is a concept that has gotten a great deal of attention in the AI community in recent years as a solution to generalization challenges. "Deep learning algorithms are based on learning intermediate representations which can be shared across tasks," Bengio [2009] explained at the dawn of this movement. "Hence they can leverage unsupervised data and data from similar tasks to boost performance on large and challenging problems that routinely suffer from a poverty of labelled data." This perspective has accrued more and more momentum ever since, as model zoos, breakthroughs in self-supervised

[1] If you'll permit me just for a moment to indulge in the contemporary sin—currently in vogue—of using "AI" as a vague catch-all. Here I roughly mean "machine learning and heuristic search," since the two fields have a lot in common.

representation learning [Chen et al., 2020a, Devlin et al., 2018], and most recently massive *foundation models* that use huge amounts of unsupervised data in a massively multi-task way [Radford et al., 2019a] have successively become central to the highest-profile uses of AI. This development was not automatic, however: "transfer learning is what makes foundation models possible," write Bommasani et al. [2021] in their recent review of the area, "but scale is what makes them powerful." It has required a concerted effort over many years for the machine learning community to build up the array of hardware, modeling techniques, experience, and datasets that are necessary to take full advantage of knowledge transfer.

As a rule, this is not how evolutionary algorithms operate. Evolutionary computing also struggles to effectively generalize and to perform robustly on new and diverse classes of problems. Knowledge transfer for search and optimization is only just beginning to be developed, however, as a possible response to this problem. In this dissertation, I have been inspired by the speculative view that natural evolution has not generally taken a linear, single-task path toward solving complex problems in biological systems—operating instead as a massively multi-task ecosystem that facilitates transfer, innovation, and "tinkering" on a wide scale [Arthur, 2009, Gerhart and Kirschner, 2007, Jacob, 1977, Lenski et al., 2003, Nguyen et al., 2016, Stanley and Lehman, 2015].

Innovation engines, however, have a lot of moving parts—assuming they are possible to build at all. So for this work I have adopted the more modest goal of aiming to understand a few of the "ingredients" that are necessary for transfer—with an eye toward how insights in this area can inform applications in the near-term. Overall, the contributions I have made here toward EKT offer a solid set of steps toward what I hope will become a larger research program in evolutionary knowledge transfer. Going forward, I am particularly interested in helping the community to establish better benchmarks for evaluating and understanding different EKT approaches.

6.2.3 Future Work

Evolutionary knowledge transfer is a young field with many lessons still being learned. In the near-term, I am pursuing a broader array of applications of these algorithms to domains where it seems transfer may prove especially useful: such as robot control for maze navigation and the synthesis of images to fool neural networks (a kind of adversarial AI application). Exploratory work of this kind may serve to help identify useful benchmark domains for testing the efficiency of different EKT strategies.

On a more fundamental level, I would like to see future work develop a more robust theoretical characterization of problem similarity—and of what kind of similarity distribution is needed in a set of problems in order for different transfer paradigms to be useful. I am especially interested in applying the tools of exploratory landscape analysis to see if problem similarity can be characterized empirically in such a way that predicts the prospects for transfer.

Lastly, the representation learning models that I presented in Chapter 5 likely have the potential to be much improved upon via further research and tuning. In the future I hope to more carefully investigate the conditions under which representation learning is possible, what kind of genotype-phenotype maps that result from such methods, and what kinds of problem classes make representation learning a good strategy for knowledge transfer.

I gave more discussion of these points for future work in sections 4.5 and 5.3.

6.3 Software Contributions

To support the experimental work in the forgoing chapters, I have also developed a new, general-purpose evolutionary algorithm framework, the Library for Evolutionary Algorithms in Python (LEAP), building on code by Jeff Bassett and extending it in collaboration with Coletti et al. [2020]. LEAP offers an improved framework for impelmenting evolutionary algorithms for research, taking advantage of Python's elegant syntax for higher-order programming to make it easy to compose new algorithms from combinations of arbitrary

selection and reproduction opertors, data-collection probes, and visualization operators. In addition to the many asynchronous steady-state and evolutionary knowledge-transfer algorithms in this dissertation (most of which were built atop LEAP), we have validated LEAP's design by implementing island models, cooperative co-evolution, adaptive evolution strategies, Pittsburgh-style evolutionary rule systems, Cartesian genetic programming, and distributed optimization on upwards of 2,000 processors, with applications to autonomous driving, computational neuroscience, image synthesis for adversarial deep learning, robot control via evolving neural networks, and more. I look forward to continuing to develop LEAP and support its growing user base in useful applications of distributed optimization in coming years.

6.4 Conclusion

In this dissertation, I have sought to address two fundamental problems that are facing evolutionary computation as a field: the problem of managing idle time when scaling to parallel and distributed applications, and the problem of pre-configuring algorithms with useful heuristic knowledge—in particulary via reusing knowledge from past tasks. I have shown that asynchronous EAs are an effective and promising family of algorithms for solving diverse optimization problems. And while evolutionary knowledge is transfer is a younger and more speculative field, I have helped to expand our fundamental understanding of this area with new theoretical and empirical results.

Bibliography

Stavros P Adam, Stamatios-Aggelos N Alexandropoulos, Panos M Pardalos, and Michael N Vrahatis. No free lunch theorem: A review. *Approximation and optimization*, pages 57–82, 2019.

Jeffrey O Agushaka and Absalom E Ezugwu. Initialisation approaches for population-based metaheuristic algorithms: A comprehensive review. *Applied Sciences*, 12(2):896, 2022.

Enrique Alba, editor. *Parallel Metaheuristics: A new Class of Algorithms*. John Wiley & Sons, Hoboken, New Jersey, 2005.

Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002.

Enrique Alba and José M. Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Gener. Comput. Syst.*, 17(4):451–465, January 2001. ISSN 0167-739X. doi: 10.1016/S0167-739X(99)00129-6. URL `http://dx.doi.org/10.1016/S0167-739X(99)00129-6`.

Reza Alizadeh, Janet K Allen, and Farrokh Mistree. Managing computational complexity using surrogate models: a critical review. *Research in Engineering Design*, 31(3):275–298, 2020.

Lee Altenberg. Evolving better representations through selective genome growth. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 182–187. IEEE, 1994.

Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, volume 3, pages 131–171. Elsevier/North-Holland, Amsterdam, London, New York, 1968.

Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, 1967.

W. Brian Arthur. *The Nature of Technology: What It Is and How It Evolves*. Free Press, 2009.

W. Brian Arthur and Wolfgang Polak. The evolution of technology within a simple computer model. *Complexity*, 11(5):23–32, May/June 2006.

Thomas Back. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Proceedings of the first IEEE conference on evolutionary computation. IEEE World Congress on Computational Intelligence*, pages 57–62. IEEE, 1994.

James E Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms (ICGA'87)*, pages 14–21, 1987.

Kavitesh Kumar Bali, Abhishek Gupta, Liang Feng, Yew Soon Ong, and Tan Puay Siew. Linearized domain adaptation in evolutionary multitasking. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1295–1302. IEEE, 2017.

Kavitesh Kumar Bali, Yew-Soon Ong, Abhishek Gupta, and Puay Siew Tan. Multifactorial evolutionary algorithm with online transfer parameter estimation: Mfea-ii. *IEEE Transactions on Evolutionary Computation*, 24(1):69–83, 2019.

Shumeet Baluja and Scott Davies. Combining multiple optimization runs with optimal dependency trees. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1997.

Thomas Bartz-Beielstein. *Experimental research in evolutionary computation: the new experimentalism*. Springer, Berlin, 2006.

Jeffrey K. Bassett. *Methods for improving the Design and Performance of Evolutionary Algorithms*. PhD thesis, George Mason University, Fairfax, VA, 2012.

DAVID A Baum and MICHAEL J Donoghue. A likelihood framework for the phylogenetic analysis of adaptation. *Adaptationism and optimality. Cambridge Univ. Press, Cambridge, UK*, pages 24–44, 2001.

Caleidgh Bayer, Ryan Amaral, Robert J Smith, Alexandru Ianta, and Malcolm I Heywood. Finding simple solutions to multi-task visual reinforcement learning problems with tangled program graphs. In *Genetic Programming Theory and Practice XVIII*, pages 1–19. Springer, 2022.

Shawn L. E. Beaulieu, Sam Kriegman, and Josh C. Bongard. Combating catastrophic forgetting with developmental compression. *arXiv:1804.04286 [cs]*, April 2018. doi: 10.1145/3205455.3205615. URL `http://arxiv.org/abs/1804.04286`. arXiv: 1804.04286.

Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Alex R Bertels and Daniel R Tauritz. Why asynchronous parallel evolution is the future of hyper-heuristics: A cdcl sat solver case study. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1359–1365, 2016.

Iwo Bladek and Krzystof Krawiec. Simultaneous synthesis of multiple functions using genetic programming with scaffolding. In *GECCO '16: Companion Proceedings of the 2016 Conference on Genetic and Evolutionary Computation*, pages 97–98, Madrid, Spain, 2016. ACM.

Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003a.

Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM computing surveys (CSUR)*, 35(3):268–308, 2003b.

Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied Soft Computing*, 11(6):4135–4151, 2011.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Lawrence D Brown, T Tony Cai, and Anirban DasGupta. Interval estimation for a binomial proportion. *Statistical science*, 16(2):101–133, 2001.

Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and John R Woodward. A classification of hyper-heuristic approaches. In *Handbook of meta-heuristics*, pages 449–468. Springer, 2010.

Erick Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

Erick Cantu-Paz. *Efficient and accurate parallel genetic algorithms*. Springer, 2000.

Jaime G Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In *Machine Learning: An Artificial Intelligence Approach*, volume 2, pages 371–392. Morgan Kaufmann, 1986.

KD Carlson, JM Nageswaran, N Dutt, and JL Krichmar. An efficient automated parameter tuning framework for spiking neural networks. *Frontiers in Neuroscience*, 8(10), 2014.

Rich Caruana. Multitask learning. In *Learning to learn*, pages 95–133. Springer, 1998.

Zheng Chai, Yujing Chen, Liang Zhao, Yue Cheng, and Huzefa Rangwala. Fedat: A communication-efficient federated learning method with asynchronous tiers under non-iid data. *arXiv preprint arXiv:2010.05958*, 2020.

Rohitash Chandra and Abhishek Gupta. Evolutionary multi-tasking for training feedforward neural networks. In *Proceedings of the International Conference on Neural Information Pr ocessing*, pages 37–46. Springer, 2016.

Rohitash Chandra, Yew-Soon Ong, and Chi-Keong Goh. Co-evolutionary multi-task learning for dynamic time series prediction, 2017a.

Rohitash Chandra, Yew-Soon Ong, and Chi-Keong Goh. Co-evolutionary multi-task learning with predictive recurrence for multi-step chaotic time series prediction. *Neurocomputing*, 2017b.

Hongyan Chen, Hai-Lin Liu, Fangqing Gu, and Kay Chen Tan. A multi-objective multitask optimization algorithm using transfer rank. *IEEE Transactions on Evolutionary Computation*, 2022a.

Kexin Chen, Alexander Johnson, Eric O Scott, Xinyun Zou, Kenneth A De Jong, Douglas A Nitz, and Jeffrey L Krichmar. Differential spatial representations in hippocampal ca1 and subiculum emerge in evolved spiking neural networks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

Qunjian Chen, Xiaoliang Ma, Yanan Yu, Yiwen Sun, and Zexuan Zhu. Multi-objective evolutionary multi-tasking algorithm using cross-dimensional and prediction-based knowledge transfer. *Information Sciences*, 586:540–562, 2022b.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PMLR, 2020a.

Yujing Chen, Yue Ning, Martin Slawski, and Huzefa Rangwala. Asynchronous online federated learning for edge devices with non-iid data. In *2020 IEEE International Conference on Big Data (Big Data)*, pages 15–24. IEEE, 2020b.

Mei-Ying Cheng, Abhishek Gupta, Yew-Soon Ong, and Zhi-Wei Ni. Coevolutionary multitasking for concurrent global optimization: With case studies in complex engineering design. *Engineering Applications of Artificial Intelligence*, 64:13–24, 2017.

Darren Chitty. A partially asynchronous global parallel genetic algorithm. In *Companion Proceedings of the 2021 Annual Conference on Genetic and Evolutionary Computation*, 2021.

Brian Christian and Tom Griffiths. *Algorithms to live by: The computer science of human decisions*. Macmillan, 2016.

Alexander W Churchill, Phil Husbands, and Andrew Philippides. Tool sequence optimization using synchronous and asynchronous parallel multi-objective evolutionary algorithms with heterogeneous evaluations. In *IEEE Congress on Evolutionary Computation (CEC) 2013*, pages 2924–2931. IEEE, 2013.

Janet Clegg, James Alfred Walker, and Julian Frances Miller. A new crossover technique for Cartesian genetic programming. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*, page 1580, London, England, 2007. ACM Press. ISBN 978-1-59593-697-4. doi: 10.1145/1276958.1277276. URL `http://portal.acm.org/citation.cfm?doid=1276958.1277276`.

Mark Coletti. *An analysis of a model-based evolutionary algorithm: Learnable Evolution Model*. PhD thesis, George Mason University, 2014.

Mark Coletti, Alex Fafard, and David Page. Troubleshooting deep-learner training data problems using an evolutionary algorithm on summit. *IBM Journal of Research and Development*, 64(3/4):1–12, 2019.

Mark A Coletti, Eric O Scott, and Jeffrey K Bassett. Library for evolutionary algorithms in python (leap). In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 1571–1579, 2020.

Mark A Coletti, Shang Gao, Spencer Paulissen, Nicholas Quentin Haas, and Robert Patton. Diagnosing autonomous vehicle driving criteria with an adversarial evolutionary algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 301–302, 2021.

Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2006.

Michael Crawshaw. Multi-task learning with deep neural networks: A survey. *arXiv preprint arXiv:2009.09796*, 2020.

Pádraig Cunningham and Barry Smyth. Case-based reasoning in scheduling: reusing solution components. *International Journal of Production Research*, 35(11):2947–2962, 1997.

Bingshui Da, Abhishek Gupta, Yew Soon Ong, and Liang Feng. The boon of gene-culture interaction for effective evolutionary multitasking. In *Australasian Conference on Artificial Life and Computational Intelligence*, pages 54–65. Springer, 2016a.

Bingshui Da, Abhishek Gupta, Yew-Soon Ong, and Liang Feng. Evolutionary multitasking across single and multi-objective formulations for improved problem solving. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1695–1701. IEEE, 2016b.

Bingshui Da, Yew-Soon Ong, Liang Feng, A. K. Qin, Abhishek Gupta, Zexuan Zhu, Chuan-Kang Ting, Ke Tang, and Xin Yao. Evolutionary multitasking for single-objective continuous optimization: Benchmark problems, performance metric, and baseline results, 2017a.

Bingshui Da, Yew-Soon Ong, Liang Feng, A Kai Qin, Abhishek Gupta, Zexuan Zhu, Chuan-Kang Ting, Ke Tang, and Xin Yao. Evolutionary multitasking for single-objective continuous optimization: Benchmark problems, performance metric, and baseline results. *arXiv preprint arXiv:1706.03470*, 2017b.

Christopher B Daniels, Sandra Orgeig, Lucy C Sullivan, Nicholas Ling, Michael B Bennett, Samuel Schürch, Adalberto Luis Val, and Colin J Brauner. The origin and evolution of the surfactant system in fish: insights into the evolution of lungs and swim bladders. *Physiological and Biochemical Zoology*, 77(5):732–749, 2004.

Charles Darwin. *On the origin of species*. J. Murray, London, 1859.

Adam Davies. *On the interaction of function, constraint and complexity in evolutionary systems*. PhD thesis, University of Southampton, April 2014.

K. A. De Jong. *Evolutionary computation: A unified approach*. MIT Press, Cambridge, MA, 2006.

Kenneth De Jong. Parameter setting in EAs: a 30 year perspective. In *Parameter Setting in Evolutionary Algorithms*, pages 1–18. Springer, 2007.

Kenneth A De Jong and Jayshree Sarma. Generation gaps revisited. In *Foundations of genetic algorithms*, volume 2, pages 19–28. Elsevier, 1993.

Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* University of Michigan, 1975.

Wouter Alexander de Landgraaf, AE Eiben, and Volker Nannen. Parameter calibration using meta-algorithms. In *2007 IEEE Congress on Evolutionary Computation*, pages 71–78. IEEE, 2007.

Pedro Paulo Balbi de Oliveira. Simulation of exaptive behaviour. In *International Conference on Parallel Problem Solving from Nature*, pages 354–364. Springer, 1994.

Francisco Fernández de Vega, Gustavo Olague, Daniel Lanza, Wolfgang Banzhaf, Erik Goodman, Jose Menendez-Clavijo, Axel Martinez, et al. Time and individual duration in genetic programming. *IEEE Access*, 8:38692–38713, 2020.

Matjaž Depolli, Roman Trobec, and Bogdan Filipič. Asynchronous master-slave parallelization of differential evolution for multi-objective optimization. *Evolutionary Computation*, 21(2):261–291, 2013.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Benjamin Doerr and Frank Neumann. *Theory of evolutionary computation: Recent developments in discrete optimization*. Springer Nature, 2020.

Carola Doerr. Complexity theory for discrete black-box optimization heuristics. In *Theory of evolutionary computation*, pages 133–212. Springer, 2020.

Stefan Droste, Thomas Jansen, and Ingo Wegener. On the analysis of the (1+ 1) evolutionary algorithm. *Theoretical Computer Science*, 276(1-2):51–81, 2002.

Amanda S. Dufek, Douglas A. Augusto, Helio J. C. Barbosa, Pedro L. S. Dias, , and Jack R. Deslippe. An efficient fault-tolerant communication algorithm for population-based metaheuristics. In *Companion Proceedings of the 2021 Annual Conference on Genetic and Evolutionary Computation*, 2021.

Juan José Durillo, Antonio J Nebro, Francisco Luna, and Enrique Alba. A study of master-slave approaches to parallelize NSGA-II. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS) 2008*, pages 1–8. IEEE, 2008.

Matteo D'Auria, Eric O Scott, Rajdeep Singh Lather, Javier Hilty, and Sean Luke. Assisted parameter and behavior calibration in agent-based models with distributed optimization.

In *International Conference on Practical Applications of Agents and Multi-Agent Systems*, pages 93–105. Springer, 2020.

David Eby, RC Averill, William F Punch, and Erik D Goodman. Evaluation of injection island ga performance on flywheel design optimisation. In *Adaptive Computing in Design and Manufacture*, pages 121–136. Springer, 1998.

Stefan Edelkamp. Planning with pattern databases. In *Sixth European Conference on Planning*, 2014.

AbdElRahman ElSaid, Joshua Karnas, Zimeng Lyu, Daniel Krutz, Alexander G Ororbia, and Travis Desell. Neuro-evolutionary transfer learning through structural adaptation. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 610–625. Springer, 2020.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.

Liang Feng, Yew-Soon Ong, Ivor Wai-Hung Tsang, and Ah-Hwee Tan. An evolutionary search paradigm that learns with past experiences. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

Liang Feng, Yew-Soon Ong, Ah-Hwee Tan, and Ivor W Tsang. Memes as building blocks: a case study on evolutionary optimization+ transfer learning for routing problems. *Memetic Computing*, 7(3):159–180, 2015a.

Liqiang Feng, Yew Soon Ong, Michele Lim, and Ivor Tsang. Memetic search with inter-domain learning: A realization between CVRP and CARP. *IEEE Transactions on Evolutionary Computation*, 19(5):644–658, 2015b.

S. W. Fentress. Exaptation as a means of evolving complex solutions. Master's thesis, University of Edinburgh, 2005.

Chrisantha Fernando, Vera Vasas, Eörs Szathmáry, and Phil Husbands. Evolvable neuronal paths: a novel basis for information and search in the brain. *PloS one*, 6(8):e23534, 2011.

Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

Sevan G Ficici, Ofer Melnik, and Jordan B Pollack. A game-theoretic investigation of selection methods used in evolutionary algorithms. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, volume 2, pages 880–887. IEEE, 2000.

Richard E Fikes, Peter E Hart, and Nils J Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

Carlos M Fonseca and Peter J Fleming. Parallel implementation of multiobjective genetic algorithms. In *Symposium on Intelligent Systems in Control and Measurement*, Miskolc, Hungary, 1998.

Tobias Friedrich and Markus Wagner. Seeding the initial population of multi-objective evolutionary algorithms: A computational study. *Applied Soft Computing*, 33:223–230, 2015.

Raluca D Gaina, Simon M Lucas, and Diego Pérez-Liébana. Population seeding techniques for rolling horizon evolution in general video game playing. In *2017 IEEE Congress on Evolutionary Computation (CEC)*, pages 1956–1963. IEEE, 2017.

Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive science*, 7(2):155–170, 1983.

J. Gerhart and M. Kirschner. The theory of facilitated variation. *Proceedings of the National Academy of Sciences*, 104, May 2007.

Mario Giacobini, Enrique Alba, and Marco Tomassini. Selection intensity in asynchronous cellular evolutionary algorithms. In Erick Cantú-Paz, JamesA. Foster, Kalyanmoy Deb, LawrenceDavid Davis, Rajkumar Roy, Una-May O'Reilly, Hans-Georg Beyer, Russell Standish, Graham Kendall, Stewart Wilson, Mark Harman, Joachim Wegener, Dipankar Dasgupta, MitchA. Potter, AlanC. Schultz, KathrynA. Dowsland, Natasha Jonoska, and Julian Miller, editors, *Genetic and Evolutionary Computation — GECCO 2003*, volume 2723 of *Lecture Notes in Computer Science*, pages 955–966. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40602-0. doi: 10.1007/3-540-45105-6_107. URL `http://dx.doi.org/10.1007/3-540-45105-6_107`.

Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

Ira Goldstein and Seymour Papert. Artificial intelligence, language, and the study of knowledge. *Cognitive science*, 1(1):84–123, 1977.

Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, and Qingfu Zhang. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 2015.

Olivier Goudet, Diviyan Kalainathan, Philippe Caillou, Isabelle Guyon, David Lopez-Paz, and Michele Sebag. Learning functional causal models with generative neural networks. In *Explainable and interpretable models in computer vision and machine learning*, pages 39–80. Springer, 2018.

Stephen Jay Gould and Elisabeth S. Vrba. Exaptation–a missing term in the science of form. *Paleobiology*, 8(1):4–15, 1982.

Lee Graham. *Exaptation and functional shift in evolutionary computing.* PhD thesis, Carleton University, December 2008.

John J Grefenstette. Parallel adaptive algorithms for function optimization. *Vanderbilt University, Nashville, TN, Tech. Rep. CS-81-19*, 1981.

John J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on systems, man, and cybernetics*, 16(1):122–128, 1986.

John J Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In Lawrence Davis, editor, *Genetic algorithms and simulated annealing*, pages 42–60. Morgan Kaufman Publishers, Inc., Los Altos, CA, 1987.

A Gupta, YS Ong, B Da, L Feng, and SD Handoko. Measuring complementarity between function landscapes in evolutionary multitasking. In *IEEE Congress on Evolutionary Computation*, 2016a.

Abhishek Gupta and Yew-Soon Ong. Genetic transfer or population diversification? deciphering the secret ingredients of evolutionary multitask optimization. *arXiv preprint arXiv:1607.05390*, 2016.

Abhishek Gupta, Yew Soon Ong, B Da, L Feng, and Stephanus D Handoko. Landscape synergy in evolutionary multitasking. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 3076–3083. IEEE, 2016b.

Abhishek Gupta, Yew-Soon Ong, and Liang Feng. Multifactorial evolution. *IEEE Transactions on Evolutionary Computation*, 20(3):343–357, 2016c.

Abhishek Gupta, Yew-Soon Ong, Liang Feng, and Kay Chen Tan. Multiobjective multifactorial optimization in evolutionary multitasking. *IEEE transactions on cybernetics*, 2016d.

Abhishek Gupta, Bingshui Da, Yuan Yuan, and Yew-Soon Ong. On the emerging notion of evolutionary multitasking: A computational analog of cognitive multitasking. In *Recent Advances in Evolutionary Multi-objective Optimization*, pages 139–157. Springer, 2017.

Abhishek Gupta, Yew-Soon Ong, and Liang Feng. Insights on transfer optimization: Because experience is the best teacher. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2(1):51–64, 2018.

Abhishek Gupta, Lei Zhou, Yew-Soon Ong, Zefeng Chen, and Yaqing Hou. Half a dozen real-world applications of evolutionary multitasking, and more. *IEEE Computational Intelligence Magazine*, 17(2):49–66, 2022.

Nikolaus Hansen. The cma evolution strategy: A tutorial. *arXiv preprint arXiv:1604.00772*, 2016.

Nikolaus Hansen, Anne Auger, Raymond Ros, Steffen Finck, and Petr Pošík. Comparing results of 31 algorithms from the black-box optimization benchmarking bbob-2009. In *Proceedings of the 12th annual conference companion on Genetic and evolutionary computation*, pages 1689–1696, 2010.

Tomohiro Harada. Mathematical model of asynchronous parallel evolutionary algorithm to analyze influence of evaluation time bias. In *2019 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, pages 1–8. IEEE, 2019.

Tomohiro Harada. Search progress dependent parent selection for avoiding evaluation time bias in asynchronous parallel multi-objective evolutionary algorithms. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1013–1020. IEEE, 2020a.

Tomohiro Harada. A study on efficient asynchronous parallel multi-objective evolutionary algorithm with waiting time limitation. In *International Conference on the Theory and Practice of Natural Computing*, pages 121–132. Springer, 2020b.

Tomohiro Harada and Enrique Alba. Parallel genetic algorithms: a useful survey. *ACM Computing Surveys (CSUR)*, 53(4):1–39, 2020.

Tomohiro Harada and Keiki Takadama. Asynchronous evaluation based genetic programming: Comparison of asynchronous and synchronous evaluation and its analysis. In *European Conference on Genetic Programming*, pages 241–252. Springer, 2013.

Tomohiro Harada and Keiki Takadama. Asynchronously evolving solutions with excessively

different evaluation time by reference-based evaluation. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 911–918, 2014.

Tomohiro Harada and Keiki Takadama. Analysis of semi-asynchronous multi-objective evolutionary algorithm with different asynchronies. *Soft Computing*, 24(4):2917–2939, 2020.

Tomohiro Harada, Misaki Kaidan, and Ruck Thawonmas. Comparison of synchronous and asynchronous parallelization of extreme surrogate-assisted multi-objective evolutionary algorithm. *Natural Computing*, pages 1–31, 2020.

Emma Hart and Kevin Sim. On the life-long learning capabilities of a NELLI*: A hyper-heuristic optimisation system. In Thomas Bartz-Beielstein, Jürgen Branke, Bogdan Filipič, and Jim Smith, editors, *Parallel Problem Solving from Nature – PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 282–291. Springer, 2014. ISBN 978-3-319-10761-5.

Daniel L Hartl and Andrew G Clark. *Principles of Population Genetics*. Sinauer, Sunderland, Massachusetts, 2007.

Edward Haslam, Bing Xue, and Mengjie Zhang. Further investigation on genetic programming with transfer learning for symbolic regression. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 3598–3605. IEEE, 2016.

Mark Hauschild and Martin Pelikan. An introduction and survey of estimation of distribution algorithms. *Swarm and evolutionary computation*, 1(3):111–128, 2011.

Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2021.

T. Helmuth, L. Spector, and J. Matheson. Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, October 2015. ISSN 1089-778X. doi: 10.1109/TEVC.2014.2362729.

G. Hendrickson and W. H. Schroeder. Transfer of training in learning to hit a submerged target. *Journal of Educational Psychology*, 32(3):205, September 1941. ISSN 1939-2176. doi: 10.1037/h0056643. URL `https://psycnet.apa.org/fulltext/1941-03730-001.pdf`. Publisher: US: Warwick & York.

Alfredo G Hernandez-Diaz, Carlos A Coello Coello, Fátima Pérez, Rafael Caballero, Julián Molina, and Luis V Santana-Quintero. Seeding the initial population of a multi-objective evolutionary algorithm using gradient-based information. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 1617–1624. IEEE, 2008.

Y-C Ho and David L Pepyne. Simple explanation of the no free lunch theorem of optimization. *Cybernetics and Systems Analysis*, 38(2):292–298, 2002a.

Yu-Chi Ho and David L Pepyne. Simple explanation of the no-free-lunch theorem and its implications. *Journal of optimization theory and applications*, 115(3):549–570, 2002b.

EBCH Hopper and Brian CH Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128(1):34–57, 2001.

Joost Huizinga and Jeff Clune. Evolving Multimodal Robot Behavior via Many Stepping Stones with the Combinatorial Multi-Objective Evolutionary Algorithm. *arXiv:1807.03392 [cs]*, July 2018. URL `http://arxiv.org/abs/1807.03392`. arXiv: 1807.03392.

Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36(1):267–306, 2009.

Marketa Illetskova, Alex R Bertels, Joshua M Tuggle, Adam Harter, Samuel Richter, Daniel R Tauritz, Samuel Mulder, Denis Bueno, Michelle Leger, and William M Siever.

Improving performance of cdcl sat solvers by automated design of variable selection heuristics. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.

Muhammad Iqbal, Will N Browne, and Mengjie Zhang. Reusing building blocks of extracted knowledge to solve complex, large-scale boolean problems. *IEEE Transactions on Evolutionary Computation*, 18(4):465–480, 2014.

Muhammad Iqbal, Mengjie Zhang, and Bing Xue. Improving classification on images by extracting and transferring knowledge in genetic programming. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 3582–3589. IEEE, 2016.

François Jacob. Evolution and tinkering. *Science*, 196(4295):1161–1166, 1977.

Domagoj Jakobović, Marin Golub, and Marko Čupić. Asynchronous and implicitly parallel evolutionary computation models. *Soft Computing*, 18(6):1225–1236, 2014.

Thomas Jansen. Analysing stochastic search heuristics operating on a fixed budget. In *Theory of Evolutionary Computation*, pages 249–270. Springer, 2020.

Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Multitask visual learning using genetic programming. *Evolutionary computation*, 16(4):439–459, 2008.

Wojciech Jaśkowski, Krzysztof Krawiec, and Bartosz Wieloch. Cross-task code reuse in genetic programming applied to visual learning. *International Journal of Applied Mathematics and Computer Science*, 24(1):183–197, 2014.

Xinfang Ji, Yong Zhang, Dunwei Gong, Xiaoyan Sun, and Yinan Guo. Multisurrogate-assisted multitasking particle swarm optimization for expensive multimodal problems. *IEEE Transactions on Cybernetics*, 2021.

Min Jiang, Zhongqiang Huang, Liming Qiu, Wenzhen Huang, and Gary G Yen. Transfer learning-based dynamic multiobjective optimization algorithms. *IEEE Transactions on Evolutionary Computation*, 22(4):501–514, 2017.

Siwei Jiang, Chi Xu, Abhishek Gupta, Liang Feng, Yew-Soon Ong, Allan Nengsheng Zhang, and Puay Siew Tan. Complex and intelligent systems in manufacturing. *IEEE Potentials*, 35(4):23–28, 2016.

Yaochu Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft computing*, 9(1):3–12, 2005.

Yaochu Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 2011.

Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *ICGA*, volume 95, pages 184–192, 1995.

Nathaniel R Kamrath, Aaron Scott Pope, and Daniel R Tauritz. The automated design of local optimizers for memetic algorithms employing supportive coevolution. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 1889–1897, 2020.

Nadav Kashtan, Elad Noor, and Uri Alon. Varying environments can speed up evolution. *Proceedings of the National Academy of Sciences*, 104(34):13711–13716, 2007.

Paul Kaufmann and Marco Platzner. Advanced techniques for the creation and propagation of modules in cartesian genetic programming. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08*, page 1219, Atlanta, GA, USA, 2008. ACM Press. ISBN 978-1-60558-130-9. doi: 10.1145/1389095.1389334. URL `http://portal.acm.org/citation.cfm?doid=1389095.1389334`.

Saeid Kazemzadeh Azad. Seeding the initial population with feasible solutions in metaheuristic optimization of steel trusses. *Engineering Optimization*, 50(1):89–105, 2018.

Borhan Kazimipour, Xiaodong Li, and A Kai Qin. A review of population initialization techniques for evolutionary algorithms. In *2014 IEEE congress on evolutionary computation (CEC)*, pages 2585–2592. IEEE, 2014.

Stephen Kelly and Malcolm I. Heywood. Multi-task learning in atari video games with emergent tangled program graphs. In *GECCO '17: Proceedings of the 2017 Conference on Genetic and Evolutionary Computation*, pages 195–202, Berlin, Germany, 2017. ACM.

William G Kennedy and Kenneth De Jong. Characteristics of long-term learning in soar and its application to the utility problem. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 337–344, 2003.

Pascal Kerschke. Comprehensive feature-based landscape analysis of continuous and constrained optimization problems using the r-package flacco. *arXiv preprint arXiv:1708.05258*, 2017.

Pascal Kerschke, Mike Preuss, Carlos Hernández, Oliver Schütze, Jian-Qiao Sun, Christian Grimme, Günter Rudolph, Bernd Bischl, and Heike Trautmann. Cell mapping techniques for exploratory landscape analysis. In *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V*, pages 115–131. Springer, 2014.

M. M. Khan, G. M. Khan, and J. F. Miller. Efficient representation of Recurrent Neural Networks for markovian/non-markovian non-linear control problems. In *2010 10th International Conference on Intelligent Systems Design and Applications*, pages 615–620, November 2010. doi: 10.1109/ISDA.2010.5687197.

Jinwoo Kim. *Hierarchical asynchronous genetic algorithms for parallel/distributed simulation-based optimization.* PhD thesis, The University of Arizona, 1994.

Byung-Il Koh, Alan D George, Raphael T Haftka, and Benjamin J Fregly. Parallel asynchronous particle swarm optimization. *International Journal for Numerical Methods in Engineering*, 67(4):578–595, 2006.

Richard E Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, pages 700–705, 1997.

Kostas Kouvaris, Jeff Clune, Loizos Kounios, Markus Brede, and Richard A Watson. How evolution learns to generalise: Using the principles of learning theory to understand the evolution of developmental organisation. *PLoS computational biology*, 13(4):e1005358, 2017.

John R Koza. *Genetic programming II, automatic discovery of reusable subprograms*. MIT Press, Cambridge, MA, 1994a.

John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994b.

Krzysztof Krawiec and Bartosz Wieloch. Automatic generation and exploitation of related problems in genetic programming. In *IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2010.

Kentarou Kurashige, Toshio Fukuda, and Haruo Hoshino. Reusing primitive and acquired motion knowledge for gait generation of a six-legged robot using genetic programming. *Journal of Intelligent and Robotic Systems*, 38(1):121–134, 2003.

John Laird, Paul Rosenbloom, and Allen Newell. *Universal subgoaling and chunking: The automatic generation and learning of goal hierarchies*. Kluwer Academic Publishers, Boston, 1986.

John E Laird. *The Soar Cognitive Architecture*. MIT Press, 2012.

John E Laird, Allen Newell, and Paul S Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40, 2017.

Pat Langley. Learning to search: From weak methods to domain-specific heuristics. *Cognitive Science*, 9(2):217–260, 1985.

Pedro Larrañaga and Jose A Lozano. *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media, 2001.

Minh Nghia Le, Yew Soon Ong, Stefan Menzel, Chun-Wei Seah, and Bernhard Sendhoff. Multi co-objective evolutionary optimization: Cross surrogate augmentation for computationally expensive problems. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8. IEEE, 2012.

Sarah Leberman and Lex McDonald. *The Transfer of Learning: Participants' Perspectives of Adult Education and Training*. CRC Press, February 2016. ISBN 978-1-317-01366-2. Google-Books-ID: z7CXCwAAQBAJ.

Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

Johannes Lengler. Drift analysis. In *Theory of Evolutionary Computation*, pages 89–131. Springer, 2020.

Richard E. Lenski, Charles Ofria, Robert T. Pennock, and Christoph Adami. The evolutionary origin of complex features. *Nature*, 423:139–144, May 2003.

Genghui Li, Qiuzhen Lin, and Weifeng Gao. Multifactorial optimization via explicit multipopulation evolutionary framework. *Information Sciences*, 512:1555–1570, 2020.

Jian-Yu Li, Ke-Jing Du, Zhi-Hui Zhan, Hua Wang, and Jun Zhang. Multi-criteria differential evolution: treating multitask optimization as multi-criteria optimization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 183–184, 2021.

Wei Li and Jinbo Li. Covariance matrix adaptation evolutionary algorithm for multi-task optimization. In *Bio-Inspired Computing: Theories and Applications: 15th International Conference, BIC-TA 2020, Qingdao, China, October 23-25, 2020, Revised Selected Papers*, volume 1363, page 25. Springer Nature, 2021.

Rung-Tzuo Liaw and Chuan-Kang Ting. Evolutionary many-tasking based on biocoenosis through symbiosis: A framework and benchmark problems. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 2266–2273. IEEE, 2017.

D Lim, YS Ong, A Gupta, CK Goh, and PS Dutta. Towards a new praxis in optinformatics targeting knowledge re-use in evolutionary computation: simultaneous problem learning and optimization. *Evolutionary Intelligence*, 9(4):203–220, 2016.

Pu Liu, Francis Lau, Michael J Lewis, and Cho-li Wang. A new asynchronous parallel evolutionary algorithm for function optimization. In *Parallel Problem Solving from Nature—PPSN VII*, pages 401–410. Springer, 2002.

Shengcai Liu, Ke Tang, and Xin Yao. Experience-based optimization: A coevolutionary approach. *arXiv preprint arXiv:1703.09865*, 2017.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

Sushil J Louis and John McDonnell. Learning with case-injected genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 8(4):316–328, 2004.

Sean Luke. Ecj then and now. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 1223–1230, 2017.

Sean Luke and Liviu Panait. Is the perfect the enemy of the good? In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 820–828, 2002.

Sean Luke and AKM Talukder. Is the meta-ea a viable optimization method? In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1533–1540. ACM, 2013.

Xiaoliang Ma, Yongjin Zheng, Zexuan Zhu, Xiaodong Li, Lei Wang, Yutao Qi, and Junshan Yang. Improving evolutionary multitasking optimization by leveraging inter-task gene

similarity and mirror transformation. *IEEE Computational Intelligence Magazine*, 16(4): 38–53, 2021.

Norman MacLeod. The role of phylogeny in quantitative paleobiological data analysis. *Paleobiology*, 27(2):226–240, 2001.

Shaul Markovitch and Paul D. Scott. The Role of Forgetting in Learning. In *Machine Learning Proceedings 1988*, pages 459–465. Elsevier, 1988. ISBN 978-0-934613-64-4. doi: 10.1016/B978-0-934613-64-4.50052-9. URL `http://linkinghub.elsevier.com/retrieve/pii/B9780934613644500529`.

Matthew A. Martin, Alex R. Bertels, and Daniel R. Tauritz. Asynchronous parallel evolutionary algorithms: Leveraging heterogeneous fitness evaluation times for scalability and elitist parsimony pressure. In *Companion Proceedings of the 2015 Conference on Genetic and Evolutionary Computation (GECCO'15)*, pages 1429–1430, Madrid, Spain, 2015. ACM.

Aritz D Martinez, Javier Del Ser, Eneko Osaba, and Francisco Herrera. Adaptive multifactorial evolutionary optimization for multitask reinforcement learning. *IEEE Transactions on Evolutionary Computation*, 26(2):233–247, 2021.

Emília P Martins. Adaptation and the comparative method. *Trends in Ecology & Evolution*, 15(7):296–299, 2000.

S.R. Maxwell. Experiments with a coroutine execution model for genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 413–417, 1994. doi: 10.1109/ICEC.1994.349915.

Deborah G Mayo. *Error and the growth of experimental knowledge*. University of Chicago Press, 1996.

Florian Mazière, Pierre Delisle, Caroline Gagné, and Michaël Krajecki. An asynchronous parallel evolutionary algorithm for solving large instances of the multi-objective qap. In *Heuristics for Optimization and Learning*, pages 69–85. Springer, 2020.

James McDermott. When and why metaheuristics researchers can ignore "no free lunch" theorems. *SN Computer Science*, 1(1):1–18, 2020.

James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 791–798, 2012.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.

Yi Mei and Mengjie Zhang. A comprehensive analysis on reusability of gp-evolved job shop dispatching rules. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 3590–3597. IEEE, 2016.

JJ Merelo, Antonio Miguel Mora, Carlos M Fernandes, and Anna I Esparcia-Alcázar. Designing and testing a pool-based evolutionary algorithm. *Natural Computing*, 12(2):149–162, 2013.

Olaf Mersmann, Bernd Bischl, Heike Trautmann, Mike Preuss, Claus Weihs, and Günter Rudolph. Exploratory landscape analysis. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 829–836, 2011.

Zbigniew Michalewicz and David B Fogel. *How to solve it: modern heuristics*. Springer Science & Business Media, 2013.

J.F. Miller and S.L. Smith. Redundancy and computational efficiency in Cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, April

2006. ISSN 1089-778X. doi: 10.1109/TEVC.2006.871253. URL http://ieeexplore.ieee.org/document/1613935/.

Julian F Miller. Cartesian genetic programming. In *Cartesian Genetic Programming*, pages 17–34. Springer, 2011.

Julian F Miller and Peter Thomson. Cartesian Genetic Programming. In *Proceedings of the Third European Conference on Genetic Programming*, volume 1802 of *LCNS*, pages 121–132. Springer-Verlag, 2000.

Jaina Mistry, Sara Chuguransky, Lowri Williams, Matloob Qureshi, Gustavo A Salazar, Erik LL Sonnhammer, Silvio CE Tosatto, Lisanna Paladin, Shriya Raj, Lorna J Richardson, et al. Pfam: The protein families database in 2021. *Nucleic acids research*, 49(D1): D412–D419, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Candia Morgan. "the language of mathematics": towards a critical analysis of mathematics texts. *For the Learning of Mathematics*, 16(3):2–10, 1996.

David E Moriarty and Risto Miikkulainen. Forming neural networks through efficient and adaptive coevolution. *Evolutionary computation*, 5(4):373–399, 1997.

J. B. Mouret and S. Doncieux. Evolving modular neural-networks through exaptation. In *Eleventh IEEE Congress on Evolutionary Computation*, Trondheim, Norway, 2009.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Heinz Mühlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In *Proc. 2nd Int. Conf. on Parallel Problem Solving from Nature, 1992*. Elsevier, 1992.

Luca Mussi, Youssef SG Nashed, and Stefano Cagnoni. Gpu-based asynchronous particle swarm optimization. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 1555–1562. ACM, 2011.

John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

Allen Newell. The knowledge level. *Artificial intelligence*, 18(1):87–127, 1982.

Allen Newell and Herbert A Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

A Nguyen, J Yosinski, and J Clune. Understanding innovation engines: Automated creativity and improved stochastic optimization via deep learning. *Evolutionary Computation*, 2016.

Anh Nguyen, Jason Yosinski, and Jeff Clune. Innovation engines: Automated creativity and improved stochastic optimization via deep learning. In *GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation*, pages 959–966, Madrid, Spain, 2015a. ACM.

Anh Mai Nguyen, Jason Yosinski, and Jeff Clune. Innovation engines: Automated creativity and improved stochastic optimization via deep learning. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 959–966, 2015b.

Emma Stensby Norstein, Kai Olav Ellefsen, and Kyrre Glette. Open-ended search for environments and adapted agents using map-elites. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 651–666. Springer, 2022.

Mariusz Nowostawski and Riccardo Poli. Parallel genetic algorithm taxonomy. In *Third International Conference on Knowledge-Based Intelligent Information Engineering Systems*, pages 88–92. IEEE, 1999.

Gabriela Ochoa, Francisco Chicano, Renato Tinós, and Darrell Whitley. Tunnelling crossover networks. In *GECCO '15: Proceedings of the 2015 Conference on Genetic and Evolutionary Computation*, pages 449–456, Madrid, Spain, 2015. ACM.

Jernej Olenšek, Tadej Tuma, Janez Puhan, and Árpád Bűrmen. A new asynchronous parallel global optimization method based on simulated annealing and differential evolution. *Applied Soft Computing*, 11(1):1481–1489, 2011.

Stephen Oman and Pádraig Cunningham. Using case retrieval to seed genetic algorithms. *International Journal of Computational Intelligence and Applications*, 1(01):71–82, 2001.

Yew-Soon Ong and Abhishek Gupta. Evolutionary multitasking: a computer science view of cognitive multitasking. *Cognitive Computation*, 8(2):125–142, 2016.

Christine A Orengo and Janet M Thornton. Protein families and their evolution—a structural perspective. *Annu. Rev. Biochem.*, 74:867–900, 2005.

Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 10:1345–1359, October 2010.

Matt Parker and Gary B Parker. Using a queue genetic algorithm to evolve xpilot control strategies on a distributed system. In *2006 IEEE International Conference on Evolutionary Computation*, pages 1202–1207. IEEE, 2006.

Maryam Parsa, John P Mitchell, Catherine D Schuman, Robert M Patton, Thomas E Potok, and Kaushik Roy. Bayesian multi-objective hyperparameter optimization for accurate, fast, and efficient neural network accelerator design. *Frontiers in neuroscience*, page 667, 2020.

Merav Parter, Nadav Kashtan, and Uri Alon. Facilitated variation: how evolution learns from past environments to generalize to new environments. *PLoS computational biology*, 4(11):e1000206, 2008.

Harold Pashler. Dual-task interference in simple tasks: data and theory. *Psychological bulletin*, 116(2):220, 1994.

P Victer Paul, A Ramalingam, R Baskaran, P Dhavachelvan, K Vivekanandan, R Subramanian, and VSK Venkatachalapathy. Performance analyses on population seeding techniques for genetic algorithms. *International Journal of Engineering and Technology (IJET)*, 5(3):2993–3000, 2013.

P Victer Paul, A Ramalingam, Ramachandran Baskaran, P Dhavachelvan, K Vivekanandan, and R Subramanian. A new population seeding technique for permutation-coded genetic algorithm: Service transfer approach. *Journal of Computational Science*, 5(2):277–297, 2014.

Judea Pearl. *Causality*. Cambridge university press, 2009.

Steven F Perry, Richard JA Wilson, Christian Straus, Michael B Harris, and John E Remmers. Which came first, the lung or the breath? *Comparative Biochemistry and Physiology Part A: Molecular & Integrative Physiology*, 129(1):37–47, 2001.

Aaron Scott Pope, Daniel R Tauritz, and Melissa Turcotte. Automated design of tailored link prediction heuristics for applications in enterprise network security. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 1634–1642, 2019.

Mitchell A Potter, R Paul Wiegand, H Joseph Blumenthal, and Donald A Sofge. Effects of experience bias when seeding with prior results. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2730–2737. IEEE, 2005.

Richard O Prum and Alan H Brush. The evolutionary origin and diversification of feathers. *The Quarterly review of biology*, 77(3):261–295, 2002.

Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019a.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019b.

Connie Loggia Ramsey and John J Grefenstette. Case-based initialization of genetic algorithms. In *ICGA*, pages 84–91. Citeseer, 1993.

Khaled Rasheed and Brian D Davison. Effect of global parallelism on the behavior of a steady state genetic algorithm for design optimization. In *IEEE Congress on Evolutionary Computation (CEC) 1999*, volume 1. IEEE, 1999.

Raphael-Elias Reisch, Jens Weber, Christoph Laroque, and Christian Schröder. Asynchronous optimization techniques for distributed computing applications. In *Proceedings of the 48th Annual Simulation Symposium*, pages 49–56. Society for Computer Simulation International, 2015.

Samuel N Richter and Daniel R Tauritz. The automated design of probabilistic selection methods for evolutionary algorithms. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 1545–1552, 2018.

Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 130, page 136. Citeseer, 2015.

Thomas Philip Runarsson. An asynchronous parallel evolution strategy. *International Journal of Computational Intelligence and Applications*, 3(04):381–394, 2003.

Jacob Russin, Randall C O'Reilly, and Yoshua Bengio. Deep learning needs a prefrontal cortex. *Work Bridging AI Cogn Sci*, 107:603–616, 2020.

Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick,

Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

Ramon Sagarna and Yew-Soon Ong. Concurrently searching branches in software tests generation through multitask evolution. In *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pages 1–8. IEEE, 2016.

Suhana Mohd Said and Mitsutoshi Nakamura. Master-slave asynchronous evolutionary hybrid algorithm and its application in vanets routing optimization. In *3rd International Conference on Advanced Applied Informatics (IIAI AAI'14)*, pages 960–965. IEEE, 2014.

Maria Salamó and Maite López-Sánchez. Adaptive case-based reasoning using retention and forgetting strategies. *Knowledge-Based Systems*, 24(2):230–247, 2011.

Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

Ralf Salomon. The influence of different coding schemes on the computational complexity of genetic algorithms in function optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 227–235. Springer, 1996.

Carolina Salto and Enrique Alba. Adapting distributed evolutionary algorithms to heterogeneous hardware. In *Transactions on Computational Collective Intelligence XIX*, pages 103–125. Springer, 2015.

Aliyu Sani Sambo, R Muhammad Atif Azad, Yevgeniya Kovalchuk, Vivek Padmanaabhan Indramohan, and Hanifa Shah. Leveraging asynchronous parallel computing to produce simple genetic programming computational models. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 521–528, 2020.

Claudio Sanhueza, Francia Jiménez, Regina Berretta, and Pablo Moscato. Pasmoqap: A

parallel asynchronous memetic algorithm for solving the multi-objective quadratic assignment problem. In *2017 IEEE congress on evolutionary computation (CEC)*, pages 1103–1110. IEEE, 2017.

Jayshree Sarma and Kenneth De Jong. Generation gap methods. *Evolutionary computation*, 1:205–211, 2000.

Kumara Sastry and David E Goldberg. On extended compact genetic algorithm. In *Late-Breaking Paper at the Genetic and Evolutionary Computation Conference*, pages 352–359, 2000.

Bernhard Schölkopf, Francesco Locatello, Stefan Bauer, Nan Rosemary Ke, Nal Kalchbrenner, Anirudh Goyal, and Yoshua Bengio. Toward causal representation learning. *Proceedings of the IEEE*, 109(5):612–634, 2021.

Chris Schumacher, Michael D Vose, L Darrell Whitley, et al. The no free lunch and problem description length. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 565–570. Citeseer, 2001.

Eric Scott, Camrynn Fausey, Karen Jones, Geoff Warner, and Hahnemann Ortiz. Automated discovery of tax schemes using genetic algorithms. In *Tax Policy Conference*, forthcoming.

Eric O. Scott and Jeffrey K. Bassett. Learning genetic representations for classes of real-valued optimization problems. In *GECCO '15: Companion Proceedings of the 2015 Conference on Genetic and Evolutionary Computation*, pages 1075–1082, Madrid, Spain, 2015. ACM.

Eric O. Scott and Kenneth A. De Jong. Understanding simple asynchronous evolutionary algorithms. In *FOGA'15: Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 85–98, New York, NY, 2015. ACM.

Eric O. Scott and Kenneth A. De Jong. Landscape features for computationally expensive

evaluation functions: Revisiting the problem of noise. In *International Conference on Parallel Problem Solving from Nature*, pages 952–961. Springer, 2016a.

Eric O. Scott and Kenneth A. De Jong. Evaluation-time bias in quasi-generational and steady-state asynchronous evolutionary algorithms. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 845–852. ACM, 2016b.

Eric O. Scott and Kenneth A. De Jong. Multitask evolution with cartesian genetic programming. In *GECCO '17: Companion Proceedings of the 2017 Conference on Genetic and Evolutionary Computation*, pages 255–256, Berlin, Germany, 2017. ACM.

Eric O. Scott and Kenneth A. De Jong. Toward learning neural network encodings for continuous optimization problems. In *Companion Proceedings of the 2017 Conference on Genetic and Evolutionary Computation*, GECCO '18, New York, NY, USA, 2018. ACM.

Eric O Scott and Sean Luke. Ecj at 20: toward a general metaheuristics toolkit. In *Proceedings of the genetic and evolutionary computation conference companion*, pages 1391–1398, 2019.

Eric O Scott, Mark Coletti, Catherine D Schuman, Bill Kay, Shruti R Kulkarni, Maryam Parsa, and Kenneth A De Jong. Avoiding excess computation in asynchronous evolutionary algorithms. In *UK Workshop on Computational Intelligence*, pages 71–82. Springer, 2021.

Eric O Scott, Mark Coletti, Catherine D Schuman, Bill Kay, Shruti R Kulkarni, Maryam Parsa, Chathika Gunaratne, and Kenneth A De Jong. Avoiding excess computation in asynchronous evolutionary algorithms. *Expert System*, in press.

Jimmy Secretan, Nicholas Beato, David B D'Ambrosio, Adelein Rodriguez, Adam Campbell, Jeremiah T Folsom-Kovarik, and Kenneth O Stanley. Picbreeder: A case study in collaborative evolutionary exploration of design space. *Evolutionary Computation*, 19(3): 373–403, 2011.

Gregory Seront. External concepts reuse in genetic programming. In *working notes for the AAAI Symposium on Genetic programming*, pages 94–98, 1995.

Joan Serrà, Dídac Surís, Marius Miron, and Alexandros Karatzoglou. Overcoming catastrophic forgetting with hard attention to the task. *arXiv:1801.01423 [cs, stat]*, January 2018. URL `http://arxiv.org/abs/1801.01423`. arXiv: 1801.01423.

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015.

Tao Shi, Hui Ma, and Gang Chen. Seeding-based multi-objective evolutionary algorithms for multi-cloud composite applications deployment. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 240–247. IEEE, 2020a.

Tao Shi, Hui Ma, and Gang Chen. Divide and conquer: Seeding strategies for multi-objective multi-cloud composite applications deployment. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, pages 317–318, 2020b.

Yang Shu, Zhi Kou, Zhangjie Cao, Jianmin Wang, and Mingsheng Long. Zoo-tuning: Adaptive transfer from a zoo of models. In *International Conference on Machine Learning*, pages 9626–9637. PMLR, 2021.

Luís F Simões, Dario Izzo, Evert Haasdijk, and Agoston Endre Eiben. Self-adaptive genotype-phenotype maps: neural networks as a meta-representation. In *International Conference on Parallel Problem Solving from Nature*, pages 110–119. Springer, 2014.

Zbigniew Skolicki. *An analysis of island models in evolutionary computation*. PhD thesis, George Mason University, 2007.

Stacey DeWitt Smith. Using phylogenetics to detect pollinator-mediated floral evolution. *New Phytologist*, 188(2):354–363, 2010.

Kate A Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys (CSUR)*, 41(1):6, 2008.

Nicolás Soca, José Luis Blengio, Martín Pedemonte, and Pablo Ezzatti. Pugace, a cellular evolutionary algorithm framework on gpus. In *IEEE Congress on evolutionary computation*, pages 1–8. IEEE, 2010.

Jacob Soderlund, Darwin Vickers, and Alan Blair. Parallel hierarchical evolution of string library functions. In *International Conference on Parallel Problem Solving from Nature*, pages 281–291. Springer, 2016.

Jorge A Soria Alcaraz, Gabriela Ochoa, Martin Carpio, and Hector Puga. Evolvability metrics in adaptive operator selection. In *Proceedings of the 2014 annual conference on genetic and evolutionary computation*, pages 1327–1334, 2014.

Kenneth O Stanley and Joel Lehman. *Why Greatness Cannot Be Planned*. Springer, 2015.

Timothy J Stanley and Trevor N Mudge. A parallel genetic algorithm for multiobjective microprocessor design. In *Proceedings of the 6th International Conference on Genetic Algorithms (ICGA)*, pages 597–604, San Francisco, CA, 1995. Morgan Kaufmann.

Jörg Stork, Agoston E Eiben, and Thomas Bartz-Beielstein. A new taxonomy of global optimization algorithms. *Natural Computing*, pages 1–24, 2020.

Thomas Stützle. Parallelization strategies for ant colony optimization. In *Parallel Problem Solving from Nature—PPSN V*, pages 722–731. Springer, 1998.

Felipe Petroski Such, Vashisht Madhavan, Rosanne Liu, Rui Wang, Pablo Samuel Castro, Yulun Li, Jiale Zhi, Ludwig Schubert, Marc G Bellemare, Jeff Clune, et al. An atari model zoo for analyzing, visualizing, and comparing deep reinforcement learning agents. *arXiv preprint arXiv:1812.07069*, 2018.

Gerald Jay Sussman. *A computer model of skill acquisition*, volume 1. American Elsevier Publishing Company New York, 1975.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. *Advances in neural information processing systems*, 26, 2013.

Kiyoharu Tagawa and Hirokazu Takeuchi. Dynamic implementation techniques of concurrent differential evolutions for multi-core CPUs. In *14th International Conference on Software Engineering, Parallel and Distributed Systems*, Dubai, United Arab Emirates, 2015.

El-Ghazali Talbi and Hervé Meunier. Hierarchical parallel approach for gsm mobile network design. *Journal of Parallel and Distributed Computing*, 66(2):274–290, 2006.

Zedong Tang, Maoguo Gong, Yue Wu, AK Qin, and Kay Chen Tan. A multifactorial optimization framework based on adaptive intertask coordinate system. *IEEE Transactions on Cybernetics*, 2021.

Daniel R Tauritz and John Woodward. Hyper-heuristics tutorial. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 685–719, 2018.

Mohammad-H Tayarani, Xiu Yao, and Hao Xu. Meta-heuristic algorithms in car engine design: a literature survey. *IEEE Transaction on Evolutionary Computation*, 19(5):609–629, 2015.

Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. *Journal of Machine Learning Research*, 10(7), 2009.

Matthew E Taylor and Peter Stone. An introduction to intertask transfer for reinforcement learning. *Ai Magazine*, 32(1):15, 2011.

Renato Tinós, Darrell Whitley, and Francisco Chicano. Partition crossover for pseudo-boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, pages 137–149, 2015.

A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, October 1950.

W Van Geit, Erik De Schutter, and Pablo Achard. Automated neuron model optimization techniques: A review. *Biological Cybernetics*, 99(4-5):241–251, 2008.

Vesselin K. Vassilev and Julian F. Miller. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, Lecture Notes in Computer Science, pages 252–263. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-46406-8.

Siva Venkadesh, Alexander O Komendantov, Stanislav Listopad, Eric O Scott, Kenneth De Jong, Jeffrey L Krichmar, and Giorgio A Ascoli. Evolving simple models of diverse intrinsic dynamics in hippocampal neuron types. *Frontiers in neuroinformatics*, 12:8, 2018.

Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. Parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *Journal of Aerospace Computing, Information, and Communication*, 3(3):123–137, 2006.

Jürgen Wakunda and Andreas Zell. Median-selection for parallel steady-state evolution strategies. In *International Conference on Parallel Problem Solving from Nature*, pages 405–414. Springer, 2000.

Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Poet: open-ended coevolution of environments and their optimized solutions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 142–151, 2019.

Richard A Watson. *Compositional evolution: the impact of sex, symbiosis and modularity on the gradualist framework of evolution*. Mit Press, 2006.

Richard A Watson and Thomas Jansen. A building-block royal road where crossover is

provably essential. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1452–1459, 2007.

Richard A Watson and Eörs Szathmáry. How can evolution learn? *Trends in ecology & evolution*, 31(2):147–157, 2016.

Richard A Watson, Günter P Wagner, Mihaela Pavlicev, Daniel M Weinreich, and Rob Mills. The evolution of phenotypic correlations and "developmental memory". *Evolution*, 68(4):1124–1138, 2014.

Shixian Wen and Laurent Itti. Overcoming catastrophic forgetting problem by weight consolidation and long-term memory. *arXiv:1805.07441 [cs, stat]*, May 2018. URL `http://arxiv.org/abs/1805.07441`. arXiv: 1805.07441.

Yu-Wei Wen and Chuan-Kang Ting. Learning ensemble of decision trees through multifactorial genetic programming. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 5293–5300. IEEE, 2016.

David R White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. Better gp benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.

Darrell Whitley. The genitor algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In *Proceedings of the third international conference on Genetic algorithms*, pages 116–121, 1989.

Darrell Whitley. A gray box manifesto for evolutionary combinatorial optimization. *ACM SIGEVOlution*, 12(1):3–5, 2019.

Darrell Whitley and Jean Paul Watson. Complexity theory and the no free lunch theorem. In *Search Methodologies*, pages 317–339. Springer, 2005.

Darrell Whitley, Keith Mathias, and Patrick Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms. In *ICGA*, volume 91, pages 77–84. Citeseer, 1991.

Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between optima: partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 915–922, 2009.

Darrell Whitley, Doug Hains, and Adele Howe. A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. In *International Conference on Parallel Problem Solving from Nature*, pages 566–575. Springer, 2010.

Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics*, 1(6):80–83, 1945.

D. G. Wilson, Julian F. Miller, Sylvain Cussat-Blanc, and Hervé Luga. Positional Cartesian Genetic Programming. *arXiv:1810.04119 [cs]*, October 2018. URL `http://arxiv.org/abs/1810.04119`. arXiv: 1810.04119.

David H Wolpert. The existence of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1391–1420, 1996a.

David H Wolpert. The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7):1341–1390, 1996b.

David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

David H Wolpert, William G Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

Sewall Wright. The roles of mutation, inbreeding, crossbreeding, and selection in evolution. In *Proceedings of the Sixth International Congress on Genetics*, pages 355–366, 1932.

H Wu. What's sophisticated about elementary mathematics. *American Educator*, 33(3): 4–14, 2009.

Qi Xia, Winson Ye, Zeyi Tao, Jindi Wu, and Qun Li. A survey of federated learning for edge computing: Research problems and solutions. *High-Confidence Computing*, page 100008, 2021.

Hang Xu, Ning Kang, Gengwei Zhang, Chuanlong Xie, Xiaodan Liang, and Zhenguo Li. Nasoa: Towards faster task-oriented online fine-tuning with a zoo of models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5097–5106, 2021a.

Qingzheng Xu, Na Wang, Lei Wang, Wei Li, and Qian Sun. Multi-task optimization and multi-task evolutionary computation in the past five years: A brief review. *Mathematics*, 9(8):864, 2021b.

Xiaoming Xue, Cuie Yang, Yao Hu, Kai Zhang, Yiu-ming Cheung, Linqi Song, and Kay Chen Tan. Evolutionary sequential transfer optimization for objective-heterogeneous problems. *IEEE Transactions on Evolutionary Computation*, 2021.

Mouadh Yagoubi. *Optimisation évolutionnaire multi-objectif parallèle: application à la combustion Diesel*. PhD thesis, Université Paris Sud-Paris XI, 2012.

Mouadh Yagoubi, Ludovic Thobois, and Marc Schoenauer. An asynchronous steady-state nsga-ii algorithm for multi-objective optimization of diesel combustion. In H. Rodrigues, editor, *Proceedings of the 2nd International Conference on Engineering Optimization*, volume 2010, page 77, 2010.

Mouadh Yagoubi, Ludovic Thobois, and Marc Schoenauer. Asynchronous evolutionary multi-objective algorithms with heterogeneous evaluation costs. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 21–28. IEEE, 2011.

Jun Yi, Wei Zhang, Junren Bai, Wei Zhou, and Lizhong Yao. Multifactorial evolutionary algorithm based on improved dynamical decomposition for many-objective optimization problems. *IEEE Transactions on Evolutionary Computation*, 26(2):334–348, 2021.

Tian-Li Yu, David E Goldberg, Kumara Sastry, Claudio F Lima, and Martin Pelikan. Dependency structure matrix, genetic algorithms, and effective recombination. *Evolutionary computation*, 17(4):595–626, 2009.

Tianhe Yu, Deirdre Quillen, Zhanpeng He, Ryan Julian, Karol Hausman, Chelsea Finn, and Sergey Levine. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning*, pages 1094–1100. PMLR, 2020.

Tina Yu and Julian Miller. Neutrality and the Evolvability of Boolean Function Landscape. In Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming*, Lecture Notes in Computer Science, pages 204–217. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-45355-0.

Tina Yu and Julian Miller. Finding Needles in Haystacks Is Not Hard with Neutrality. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea Tettamanzi, editors, *Genetic Programming*, Lecture Notes in Computer Science, pages 13–25. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-45984-2.

Yuan Yuan, Yew-Soon Ong, Abhishek Gupta, Puay Siew Tan, and Hua Xu. Evolutionary multitasking in permutation-based combinatorial optimization problems: Realization with tsp, qap, lop, and jsp. In *Region 10 Conference (TENCON), 2016 IEEE*, pages 3157–3164. IEEE, 2016.

Yuan Yuan, Yew-Soon Ong, Liang Feng, A. K. Qin, Abhishek Gupta, Bingshui Da, Qingfu Zhang, Kay Chen Tan, Yaochu Jin, and Hisao Ishibuchi. Evolutionary multitasking for multiobjective continuous optimization: Benchmark problems, performance metrics and baseline results, 2017.

Linda Zagzebski. The inescapability of gettier problems. *The Philosophical Quarterly (1950-)*, 44(174):65–73, 1994.

Ahmed Bin Zaman, Kenneth De Jong, and Amarda Shehu. Using subpopulation eas to map molecular structure landscapes. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 960–967, 2019.

Amir R Zamir, Alexander Sax, William Shen, Leonidas J Guibas, Jitendra Malik, and Silvio Savarese. Taskonomy: Disentangling task transfer learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3712–3722, 2018.

Alexandru-Ciprian Zăvoianu, Edwin Lughofer, Werner Koppelstätter, Günther Weidenholzer, Wolfgang Amrhein, and Erich Peter Klement. On the performance of master-slave parallelization methods for multi-objective evolutionary algorithms. In *Artificial Intelligence and Soft Computing*, pages 122–134. Springer, 2013a.

Alexandru-Ciprian Zăvoianu, Edwin Lughofer, Werner Koppelstätter, Günther Weidenholzer, Wolfgang Amrhein, and Erich Peter Klement. On the performance of master-slave parallelization methods for multi-objective evolutionary algorithms. In *International Conference on Artificial Intelligence and Soft Computing*, pages 122–134. Springer, 2013b.

Alexandru-Ciprian Zăvoianu, Edwin Lughofer, Werner Koppelstätter, Günther Weidenholzer, Wolfgang Amrhein, and Erich Peter Klement. Performance comparison of generational and steady-state asynchronous multi-objective evolutionary algorithms for computationally-intensive problems. *Knowledge-Based Systems*, 2015.

Bernard P Zeigler and Jinwoo Kim. Asynchronous genetic algorithms on parallel computers. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 660. Morgan Kaufmann Publishers Inc., 1993.

Fangfang Zhang, Su Nguyen, Yi Mei, and Mengjie Zhang. Adaptive multitask genetic programming for dynamic job shop scheduling. In *Genetic Programming for Production Scheduling*, pages 271–290. Springer, 2021a.

Fangfang Zhang, Su Nguyen, Yi Mei, and Mengjie Zhang. Multitask learning in hyper-heuristic domain with dynamic production scheduling. In *Genetic Programming for Production Scheduling*, pages 249–269. Springer, 2021b.

Fangfang Zhang, Su Nguyen, Yi Mei, and Mengjie Zhang. Surrogate-assisted multitask genetic programming for learning scheduling heuristics. In *Genetic Programming for Production Scheduling*, pages 291–311. Springer, 2021c.

Jun Zhang, Weien Zhou, Xianqi Chen, Wen Yao, and Lu Cao. Multisource selective transfer framework in multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 24(3):424–438, 2019.

Xiaolong Zheng, Yu Lei, Maoguo Gong, and Zedong Tang. Multifactorial brain storm optimization algorithm. In *Bio-Inspired Computing-Theories and Applications*, pages 47–53. Springer, 2016.

Xiaolong Zheng, Yu Lei, A Kai Qin, Deyun Zhou, Jiao Shi, and Maoguo Gong. Differential evolutionary multi-task optimization. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 1914–1921. IEEE, 2019.

Lei Zhou, Liang Feng, Jinghui Zhong, Yew-Soon Ong, Zexuan Zhu, and Edwin Sha. Evolutionary multitasking in combinatorial search spaces: A case study in capacitated vehicle routing problem. In *Computational Intelligence (SSCI), 2016 IEEE Symposium Series on*, pages 1–8. IEEE, 2016.

Xinyun Zou, Eric Scott, Alexander Johnson, Kexin Chen, Douglas Nitz, Kenneth De Jong, and Jeffrey Krichmar. Neuroevolution of a recurrent neural network for spatial and working memory in a simulated robotic environment. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 289–290, 2021.

Biography

Eric Scott is a Lead Artificial Intelligence Engineer at MITRE Corporation in McLean, Virginia, where he directs research into machine learning and optimization for applications that benefit the United States government and its many mission areas. Eric is author of over twenty peer-reviewed publications covering evolutionary algorithms, artificial intelligence, and agent-based modeling. He graduated from Andrews Academy in Berrien Springs, Michigan in 2006, and completed a double Bachelors of Science degree in Computer Science and Mathematics at Andrews University in 2011. During that time he also interned as an undergraduate research fellow at the Santa Fe Institute in New Mexico—where he first began research into evolutionary knowledge transfer. He received a Masters of Science degree in Computer Science from George Mason University in 2015, working there as a graduate research assistant from 2011 to 2019. Eric joined MITRE in 2013 as a graduate intern, and transitioned to full-time employment as an AI Engineer in 2019. He lives in Herndon, Virginia with his wife, sister-in-law, and two black cats who have been his trusty companions throughout graduate school.