USING APPROXIMATE DYNAMIC PROGRAMMING TO ADAPT A MILITARY FORCE MIX

by

Jason Alan Southerland
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Systems Engineering/Operations Research

Committee:

_____	Dr. Andrew Loerch, Dissertation Director
_____	Dr. Carlotta Domeniconi, Committee Member
_____	Dr. Rajesh Ganesan, Committee Member
_____	Dr. John Shortle, Committee Member
_____	Dr. Ariela Sofer, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date:_____	Spring Semester 2017 George Mason University Fairfax, VA

Using Approximate Dynamic Programming to Adapt a Military Force Mix

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

by

Jason Alan Southerland
Master of Science
George Mason University, 2008
Bachelor of Arts
University of Texas at Austin, 2004

Director: Andrew Loerch, Associate Professor
Department of Systems Engineering/Operations Research

Summer Semester 2017
George Mason University
Fairfax, VA

DEDICATION

This is dedicated to all of my family, without whose example I would never have thought to undertake something like this dissertation.

ACKNOWLEDGEMENTS

I would like to thank the many friends, relatives, and supporters who have made this happen. I owe special thanks to Dr. Andrew Loerch for orienting me on the right path; to Dr. Rajesh Ganesan for his guidance and mentorship since I first enrolled at George Mason in 2006; and to the Dr. Shortle and Dr. Domeniconi for their support and guidance throughout the dissertation process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

ABSTRACT

USING APPROXIMATE DYNAMIC PROGRAMMING TO ADAPT A MILITARY FORCE MIX

Jason Alan Southerland, Ph.D.

George Mason University, 2017

Dissertation Director: Dr. Andrew Loerch

Adaptation, "the ability to bring about timely and effective adjustment or change in response to the surrounding environment" (Defense Science Board, 2010), is critical to maintaining system performance in dynamic environments. The global security environment is one such dynamic environment and military forces must adapt to the evolving global security environment to remain relevant.

Military force adaptation can occur at several levels. A few examples include: militaries may improve the performance of existing systems or acquire new systems; militaries may change the mix of personnel or equipment in the designs of particular unit types; or they may opt to change the overall mix of units within a military force. We focus on the latter: the mix of units within a military force. We refer to this mix as a force structure or force mix. Though others may refer to the mix of personnel and equipment in

a particular unit type as force structure (Loerch, 2007), for clarity we refer to this as unit design.

In the United States military, the Secretary of Defense publishes strategic guidance which is then enacted by multiple entities: per Title 10, United States Code (United States Government, 2016), the military departments prepare and provide forces to the Combatant Commands who execute military missions. In enacting the Secretary's strategic guidance, there is a natural tension between meeting current and near-term mission requirements on one hand, and maintaining sufficient uncommitted forces to be able to respond to unforeseen crises, up to and including major contingencies.

As the dynamic security environment evolves, adversary capabilities improve. In response, strategic guidance may change, and in turn, Combatant Command mission requirements may increase or decrease. Despite these changes, the military departments must maintain their ability to provide relevant forces. One way to maintain this ability is to periodically review force structure in order to identify necessary changes in force mix.

Each year, the Department of the Army (hereafter referred to as the Army), conducts a comprehensive review of its force structure known as Total Army Analysis (United States Army, 1995). The goal of any TAA is to identify changes, within total personnel constraints mandated by Congress, to the existing force structure which maintain or improve the Army's ability to meet Combatant Command mission requirements while maintaining the ability to respond to unforeseen crises.

Analyzing the Army's force structure is no trivial task. In a typical TAA, the Army considers changes to the inventories of 150-200 unit types across three components

(Loerch, 2007). To manage this complexity, the Army divides its overall force structure into logical groupings of unit types, such as logistics or intelligence, and considers each logical grouping separately in what is known as a resourcing panel. Each resourcing panel is given guidance with respect to a total personnel change that panel must identify. The total of the guidance across the panels equates to the overall Congressionally-authorized change in personnel from the existing personnel total.

In this article, we describe an approximate dynamic programming (ADP) methodology that can be applied to support resourcing panel deliberations. Our methodology identifies valuable force structure changes within given constraints by using a simulation that models the occurrence of military missions; readiness requirements; and the management of Army units to meet both mission and readiness requirements.

Our approximate dynamic program applies a diffusion wavelet transform (DWT) value function approximation (VFA). We provide computational results that demonstrate performance superior to the application of a myopic heuristic and we examine the effect of approximation size, a critical DWT parameter, on simulation outcomes.

CHAPTER ONE: INTRODUCTION

1.1 Flexibility, Adaptability, and Robustness

Robustness is the ability of a system to sustain key capabilities irrespective of the environment [Deshmukh et. al, 2010]. This definition has two key components—the first being the ability of a system to sustain key capabilities and the other being "irrespective of the environment." An equivalent statement of the first portion of the definition of robustness would be "the ability of a system to operate as designed." The second portion of the definition, "irrespective of environment," implies designing a system to operate in many environments. Thus, if a system is operating as intended in one environment the system should still be able to perform as designed if the environment changes [Downey et. al, 2003].

If designing a system for robust performance requires ensuring the system can perform in a variety of environments, the design process must determine how changes in the environment affect system performance. By identifying how changes in the environment affect system performance, system designers can build in hedges to environmental change and thus ensure continued system performance. Designing for robust system performance thus requires two things—planning for changes in the environment and hedging against those changes. These two considerations motivate two key aspects of system design, adaptability and flexibility.

Adaptability is "the ability to bring about timely and effective adjustment or change in response to the surrounding environment [Defense Science Board, 2010]. Thus, adaptation is about changing a system in order to ensure continued performance of that system. Enabling adaptation through system design entails anticipating the possibility that changing the system might be necessary and preparing for that change as conditions warrant [Defense Science Board, 2010].

Roughly speaking, flexibility is the number of possible configurations of a system (adapted from [Gerwin, 1993]). In discussing adaptation, we highlighted the importance of change to maintain system performance. Flexibility, by providing a menu of possible changes to a system, enables response to changes in the environment. Thus, flexibility enables adaptation.

In this research we examined the ability of a military force to simultaneously satisfy stochastic mission requirements and maintain prescribed readiness levels to respond to crises by altering the mix of units in its inventory. In other words, we examined the ability of the military force to adapt to some unknown, and possibly changing, future mission set by implementing changes to its configuration. To achieve this we leverage the state-based decision learning of approximate dynamic programming.

1.2 Value Function Approximation with a Diffusion Wavelet Transform

To solve the military force mix problem with approximate dynamic programming, we used a diffusion wavelet transform (DWT) value function approximation (VFA). Compared to other VFA approaches, there is relatively little research on the properties of

DWT. The parameters of a VFA influence two factors: approximation convergence and solution quality.

Any number of VFA factors can influence convergence and solution quality, for example, length of exploration phase and method of exploration; alpha decay; and approximation method. One critical factor to DWT VFA, size of approximation, has not been researched with respect to convergence and solution quality. In this dissertation, we research the relationship between solution space size, approximation size, and the resultant approximation convergence and solution quality.

1.3 Summary

This research makes two contributions to the literature, one related to the application of our methodology to a challenging problem that receives little attention in the literature and the other related to the computational aspects of the solution. We developed a novel solution to the military force mix problem, and we investigated the relationship between solution space size and solution quality with application of the diffusion wavelet transform value function approximation.

This dissertation has seven chapters. Chapter two describes defense planning and programming. In chapter three we review the military force structure and fleet planning literature. Chapter four reviews approximate dynamic programming. Chapter five defines our methodology and chapter six describes our experimentation. Chapter seven concludes this dissertation.

CHAPTER TWO: THE MILITARY FORCE MIX PROBLEM IN CONTEXT

Determining the best mix of military forces is a complex problem whose solution must consider, among other things, a wide range of possible missions; fiscal and personnel availability constraints; and the time and resources necessary to prepare units to execute missions. In determining its force mix, the Army must decide how to manage the readiness of its units by determining the possible nature and timing of future contingencies. In other words, the Army must answer three key questions: "of what," "for what," and "for when" [Betts, 1995]. In this chapter, we discuss the laws defining roles and responsibilities of the Combatant Commands and the military departments, of which the Army is one; and how each plans for and executes those responsibilities. We conclude this chapter by discussing analyses of defense planning and their relation to our research.

2.1 Title 10, United States Code: Roles and Responsibilities

Title 10, United States Code [United States Government, 2012] establishes the laws of the United States concerning the performance of military mission. We summarize below the key organizations and the roles and responsibilities thereof that are relevant to our research.

The President of the United States is responsible for establishing combatant commands, both unified and specified. Unified combatant commands are military

commands which have "broad, continuing missions and are composed of forces from two or more military departments." These combatant commands are responsible for the *performance of military missions* (emphasis added).

The secretaries of the military departments assign forces to the combatant commands to perform missions directed by the Secretary of Defense. The Secretary of Defense both directs the conduct of missions and approves the assignment of forces. More colloquially, the combatant commands first request forces, and the Secretary of Defense then directs the military departments to *provide* forces. The roles of the combatant commands as force requestors and mission performers and the role of the military departments as force providers are critical for our research.

In order for the military departments to provide forces, they must carry out other responsibilities as prescribed in Title 10—organizing, equipping, and training forces among others. The Army in particular is responsible for "the preparation of land forces necessary for the effective prosecution of war except as otherwise assigned and…for the expansion of the peacetime components of the Army to meet the needs of war." Thus, the Army organizes, trains, and equips forces in order to provide those forces to the combatant commands for the performance of military missions.

2.2 Combatant Command Demand

Combatant commanders use two processes for submission of requests for forces (RFFs)—annual RFFs and emergent RFFs [United States Army, 2011]. Annual RFFs are submitted once each year and include requirements for operations and other military activities the Combatant Commander plans to execute in the coming year. Emergent

RFFs are submitted outside of this annual process and are used to request forces for uses unanticipated during submission of annual RFFs. Combatant commanders submit emergent RFFs as needs arise.

This request for forces process is not a rigid request-then-provide system. While the combatant commanders make requests for particular sorts of units, the military departments have a say in the types of units that they actually provide for use by the combatant commanders. The process contains a fair amount of flexibility in that the departments are able to recommend substitute unit types. Typically, the departments will recommend substitution based on a combination of factors, including availability of preferred units and the availability of suitable substitutes as measured by the ability of other units to perform the mission, and the degree of risk assumed by providing a potentially less capable unit type.

For example, a combatant commander may request a brigade combat team for an area security task as part of a larger stabilization operation. The Army might recommend an artillery unit to perform this task based on non-availability of brigade combat teams and the assessment that the field artillery unit can perform the task with minimal risk to the overall mission.

In addition to the RFF process, the Combatant Commands provide guidance to each military department, consolidated by the Secretary of Defense, to maintain a certain aggregate level of readiness within its forces. These readiness guidelines coupled with the forecast and emergent requests for forces represent aggregate demand for forces.

2.3 Army Force Structure and Unit Readiness

In this section, we discuss the units the Army prepares for provision to the combatant commands and the process through which the Army manages the preparation those units.

2.3.1 Force Structure

Force structure can refer to entities at varying resolutions. At the lowest level, force structure refers to the detailed makeup of personnel and equipment within a specific type of military unit [Loerch, 2007]. These specific types of military units, also known identified by standard requirements codes (SRCs), are the building blocks of the more aggregated force structure. Title 10 defines forces structure as "the set of units and organizations that exist" [United States Government, 2012] within the Army. From the Army's perspective, force structure is the portfolio of the various quantities of each SRC that exist within the Army. This portfolio details the units the Army prepares for provision to the combatant commands.

The tasks for which various units of force structure are designed range along a spectrum of specificity from highly specific to highly general. Some unit types, chemical response teams, for example, are designed to perform highly specified tasks. Other units, brigade combat teams, for example, are designed to perform myriad tasks. Unit types designed to perform a wide range of tasks can be more flexibly employed than units designed to perform fewer tasks.

2.3.2 Readiness and Deployment Management

In the United States military, readiness is the ability of forces to fight and meet the demands of the national military strategy. Readiness is derived from unit readiness

and joint readiness. Unit readiness is the ability of military units to deliver the outputs for which they are designed [The Joint Staff, 2012c]. Unlike joint readiness, ensuring unit readiness is the responsibility of the military departments.

Ensuring unit readiness entails preparing units for the missions they will perform under combatant command leadership. Preparing forces for the execution of military missions is a complex process of ensuring personnel and equipment are in the right place at the right time, and that a series of increasingly complex training exercises is successfully completed. The details of preparing units for mission execution are beyond the scope of this research. Some of the tasks required for this preparation are listed below.

The Army has defined varying degrees of preparation, known as readiness levels. Each readiness level corresponds to the achievement of various milestones with respect to manning, equipping, and training units. In reality, readiness progresses over time through discrete levels. The graph of a unit's readiness over time would resemble a step function. For our purposes, the readiness levels correspond roughly to the amount of time remaining until a unit is fully prepared for the mission assigned to it. At any given point in time, each unit in the Army has some level of readiness.

2.3.2.1 Static Readiness Management

In static readiness, units maintain a fixed level of readiness prior to preparations for deployment. This readiness model entails assessing two key variables—how long from mission notification will a unit take to be ready for mission execution; and to what extent should a unit prepare for a specific mission. In general, though there are

exceptions, units maintained at lower readiness take longer to prepare for mission execution with the ability to prepare for a broad set of missions upon notification while units maintained at higher readiness prepare faster for more specific missions.

In a resource-unconstrained world, all units would maintain the highest level of readiness. In reality, maintaining a total force at highest readiness is prohibitively expensive [George, 1999]. Maintaining units at lower readiness is less expensive than maintaining units at higher readiness. Thus, in general, static readiness management entails determining which units would prepare for specific missions and be ready to respond quickly to those missions and which units would be maintained at lower readiness with the flexibility to prepare, on notification, for any of an assortment of missions.

The following example demonstrates a static readiness management model— in response to fiscal pressures in the post-Cold War era, Senator John McCain proposed a readiness model known as "tiered readiness." Under tiered readiness, forces would be maintained, statically, at one of three levels of readiness. The levels corresponded to how early in a major war-fight units would be required to perform missions— forward-deployed and crisis response forces would respond earliest in campaigns; force buildup units would respond next; and conflict resolution forces would see campaigns through to their end [McCain, 1996].

2.3.2.2 Progressive Readiness Management

This discussion highlights two key variables that complicate the preparation of forces for mission execution—the nature of the missions for which the Army prepares

forces and the duration of those missions. Specifically, as relates to the nature of missions, suppose under the static model that units in high readiness prepared for a mission other than that for which it operated. That unit might still be able to deploy for this other mission, though in doing so, the combatant commander might inherit some risk associated with diminished proficiency. And as relates to mission duration, suppose the mission lasted longer than anticipated. As units spend more time deployed, their skills might atrophy, again saddling the combatant commander with unforeseen risk. Determining the nature and duration of potential missions is critically important, as the following example demonstrates.

In 2004, the Army realized that its static readiness management model was not well suited to the continued provision of forces for operations in Iraq and Afghanistan. The Army had been operating in Afghanistan for three years and the nature of operations in Iraq had evolved from major combat operations to post-hostility stabilization operations. The Army had not prepared its force to execute post-hostility stabilization operations and the requirement to do so was taking longer than expected. Given these realities the Army needed to devise a readiness model that would allow it to prepare forces for a diverse collection of missions over a long period of time with the resources currently in its inventory.

In response to this realization, the Army developed a readiness model known as Army Force Generation, or ARFORGEN [United States Army, 2011]. Rather than the static model through which the Army prepared units for specific missions, ARFORGEN, through a progressive readiness approach, provided the Army the flexibility to prepare

units for diverse missions as well as a mechanism for meeting the demands of sustained, long-duration operations.

In this progressive readiness model, units progress through three phases, thus the name progressive readiness. These phases— reset, train/ready, and available, correspond to varying levels of readiness, in increasing order. A broad overview of the ARFORGEN model is provided below.

In this model, units train during the train/ready phase initially on a broad range of general skills. At some point during the train/ready phase, typically not to exceed 90 days prior to entering the available phase, a unit will either receive a specific mission and be designated as a deployment expeditionary force (DEF) or will be designated as a contingency expeditionary force (CEF). In either case, after this designation, units then prepare for their assigned or contingency mission. Note here that preparing for this mission requires more specific training than the general training during the early portions of the train/ready phase. This process enables the Army to prepare units first for a broad range of potential missions, then to focus unit training on specific missions as their scheduled deployment (in the case of DEF units) or entry into the available pool (in the case of CEF units) approaches. In the event that demand for forces exceeds the number of units in the available phase of their cycle, the Army can deploy units in their train/ready phase.

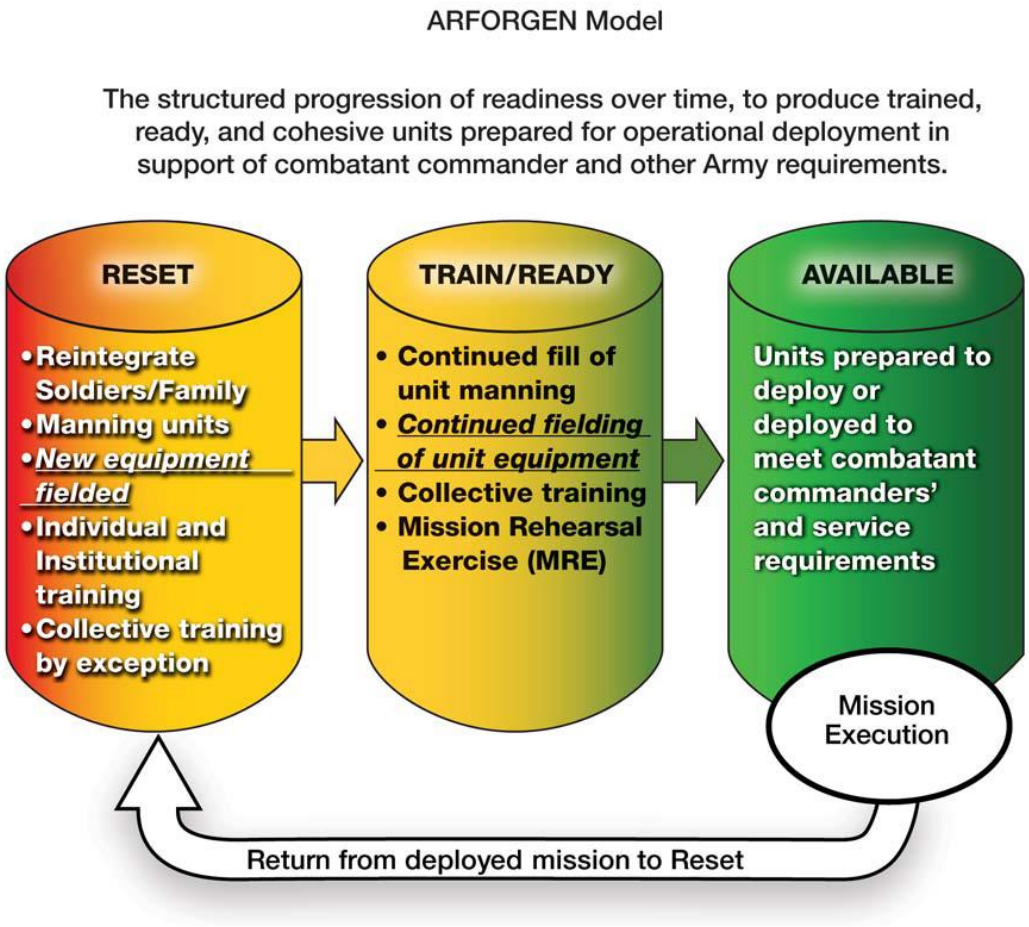

Figure 1: Army Force Generation Model [United States Army, 2011]

This ability to deploy units in their train/ready phase provides the Army with flexibility to meet unforeseen or, to use the language of combatant command requests for forces, emergent requirements. Another key flexibility under ARFORGEN is the periodic review of deployment lengths. The Army has, in the past, changed its maximum deployment length to meet forecasted requests for forces (see, for example, [McIlvaine, 2012]). This ability to review deployment policy helps ensure the Army is able to

continue to meet its Title 10 responsibilities as a preparer of forces within the force structure at its disposal.

2.4 From Strategy to Resources: Defense Planning and Programming

The system through which the Department of Defense determines how to allocate its finite resources is known as the Defense Planning, Programming, Budgeting, and Execution System (PPBES). Each component of PPBES encompasses a different time horizon, from the present to some point in the future, in decreasing order of duration. That is, planning includes deliberation from the present further into the future than programming, programming considers further out than budgeting, and so on. Budgeting and execution only concern decisions in the short term. We will not discuss budgeting and execution further.

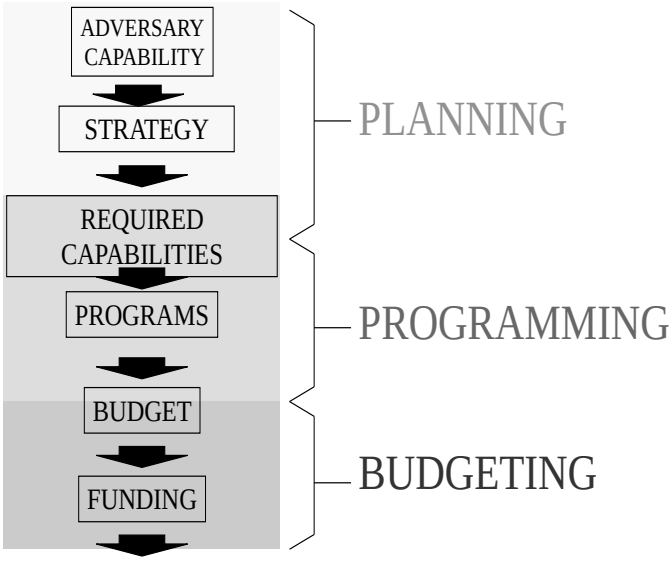

Figure 2: Components of Planning, Programming, and Budgeting [United States Army Force Management School, 2010]

2.4.1 Planning

The Department of Defense (DOD) conducts strategic planning to examine the military posture as it relates to national security objectives and resource constraints [United States Army, 1994]. DOD uses strategic planning to develop the National Military Strategy and determines the resources required to execute that strategy. Defense strategic planning produces many documents which serve the purpose of advising DOD senior leadership and providing direction to the services for planning.

One of the many documents DOD produces during its strategic planning efforts is the Defense Planning Guidance (DPG). The DPG is the primary means of providing planning direction to the services. The DPG presents the Secretary of Defense's plan for developing and employing future forces and reflects long-range plans and priorities for the Department of Defense [United States Army, 1994].

2.4.2 Programming

Every year, each military department submits to the Office of the Secretary of Defense (OSD) a program objective memorandum (POM), also known as a program. OSD then reviews these inputs, directs changes to the departments, and submits a final DOD program to the Congress.

The program translates planning decisions, programming guidance, and congressional guidance into a comprehensive allocation of forces, manpower, and funds [United States Army, 1994]. This program submission contains a proposed resource allocation for the next year, known as a budget estimate, and the four subsequent years.

Note here that the program contains a proposed allocation of forces. We discuss in the next section the analytical construct, known as an integrated security construct, used to determine this allocation of forces.

2.4.3 Bridging the Gap from Planning to Programming: The Integrated Security Construct

During the planning phase, based on the defense strategy and other senior leader priorities, planners and analysts develop a collection of scenarios, each of which represents a likely or significant challenge the military might face in the future. Examples of scenarios include—a major stabilization operation; extending support to civil authorities in response to a catastrophic event in the United States; and deterring and defeating regional aggressors [Department of Defense, 2010]. Typically, the resources required to succeed in a scenario and the duration of those resource requirements are determined during scenario development and carried forward as static inputs for follow-on analyses. These scenarios provide the foundation for analysis during the programming phase.

In older planning efforts planners would typically only develop scenarios to analyze major combat operations. The implicit assumption in this approach was the belief that if the military could meet the requirements of a collection of major regional conflicts, the ability to succeed in other operations was "lesser-included." With the end of the Cold War and the necessary evolution in defense planning post-Cold War, the "lesser included" assumption was subjected to scrutiny and abandoned for the purposes of defense planning. Thus, post-Cold war defense planning began to include detailed

analysis of operations other than major regional conflicts or major combat operations [DuBois, 1999].

The current paradigm for identifying scenarios for defense planning is the spectrum of conflict. The spectrum of conflict characterizes military conflict based on an ascending scale of violence in which military forces operate, from stable peace on one end to general war on the other [United States Army, 2011]. In any point along the spectrum of conflict, military forces conduct operations to reduce violence and establish conditions that advance national strategic goals [United States Army, 2011].

A useful refinement of the spectrum of conflict, operational themes, describes the nature of the dominant type of operations military forces conduct to reduce violence and achieve national strategic objectives [United States Army 2011]. For defense planning purposes, operational themes help define the types of operations that are considered for scenario analysis. Some types of operations and their associated operational themes are detailed below.

Peacetime military engagement	*Limited intervention*	*Peace operations*	*Irregular warfare*
• Multinational training events and exercises • Security assistance • Joint combined exchange training • Recovery operations • Arms control • Counterdrug activities	• Noncombatant evacuation operations • Strike • Raid • Show of force • Foreign humanitarian assistance • Consequence management • Sanction enforcement • Elimination of weapons of mass destruction	• Peacekeeping • Peace enforcement • Peacemaking • Peace building • Conflict prevention	• Foreign internal defense • Support to insurgency • Counterinsurgency • Combating terrorism • Unconventional warfare

Figure 3: Operational themes and associated types of operations [United States Army, 2011]

The current approach to scenario planning is the integrated security construct (ISC). An ISC is a combination of scenarios that represents one potential collection of future challenges, arrayed over a prescribed period of time in the future. An ISC essentially selects some number of individual scenarios and places those scenarios in some order of occurrence. Since the resources required for any scenario and the duration of each scenario is a pre-determined input, constructing an ISC provides planners and analysts a time series of resource requirements for further analysis. A notional ISC is depicted below.

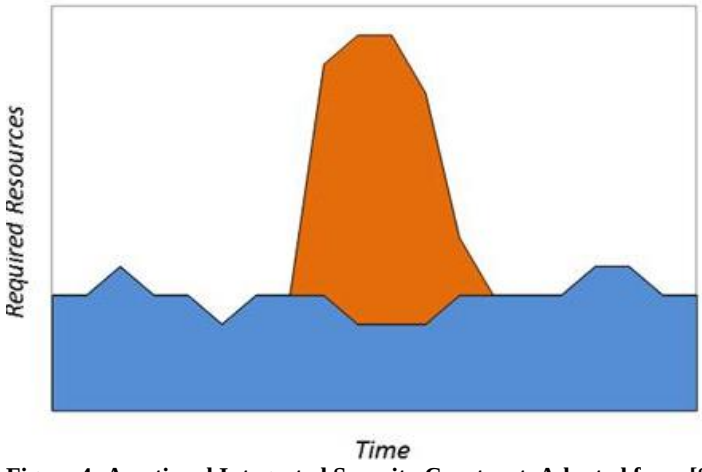

Figure 4: A notional Integrated Security Construct. Adapted from [Stoddard et. al., 2011]

2.5 Total Army Analysis (TAA)

Total Army Analysis is the Army's analytical venue for determining its force structure. The objective of TAA is to determine and justify a program force structure consistent with Defense Planning Guidance and other Army plans [United States Army,

1995]. While force structure is but part of the entire Army program, determining the program force structure is of critical importance for other elements of the program such as personnel and equipment decisions.

TAA includes activities spanning the spectrum from planning to programming. Many of the analyses conducted by the Army during the planning phase of TAA inform other DOD efforts, particularly the campaign analyses (of which, more later), which often form the basis for Army requirements in DOD planning documents.

2.5.1 TAA Planning: Determining Force Structure Requirements

Army analysts determine force structure requirements in one of two ways—by direct simulation or by use of allocation rules [Loerch, 2007]. In some cases, the numbers of large combat forces, such as brigade combat teams, required for a given mission are specified by the Department of Defense outside the scope of TAA. In these cases, the Army will still conduct direct simulation to inform the analysis of forces determined through allocation rules.

Given the large number of types of units, analysts use direct simulation to determine requirements for only a limited subset of unit types. These directly determined unit types are typically combat forces, such as brigade combat teams. The remainder of unit types are then determined via allocation rules, which we discuss in more detail later.

Direct simulation takes on one of two forms—subject matter expert-informed, tabletop exercises; and computational, theater combat modeling. The form of simulation is a function of the availability of higher resolution computer models. In general, computer simulation models are only available for high intensity conflict and in recent

years, for stability operations. In either case, analysts iterate through numerous instances of the simulation until they determine a combat force that can achieve specified objectives and within specified parameters such as timelines and casualty thresholds.

During the conduct of these simulations, analysts ensure that results used to inform allocation rules are recorded. These results include, but are not limited to, ammunition expenditure, geographic areas of responsibility, combat organization, and casualties [Crain, 2007].

Forces not determined through direct combat simulation are determined using rules of allocation, of which there are two types—existence rules and workload rules. Existence rules specify that for every unit of type X in existence, some number of units of type Y are required. Workload rules specify requirements such as the ratio of units required to handle various volumes of ammunition expenditure [Loerch, 2007].

Taken together, the forces determined through direct combat simulation and by means of allocation rules constitute the force requirements for a single engagement. TAA then takes these requirements from engagements specified by the Department of Defense (as discussed in section 2.4.3) to create a time-series of force requirements over some specified period of time. Often this force structure requirement exceeds Congressionally-mandated ceiling on personnel and other resources [Loerch, 2007]. In these cases, it is necessary to select a portfolio of units that minimizes the risk incurred over the specified planning scenario within these personnel and other resource constraints. In the resourcing phase of TAA, Army analysts perform precisely this function.

2.5.2 TAA Programming: Resourcing a Force Structure

After determining the force structure requirements, TAA shifts its focus to determining which elements in the force structure will be funded in future years, an exercise known as resourcing the force structure. This resourcing process takes place over several weeks and involves groups of individuals, typically colonels and their civilian equivalents, performing qualitative analysis of the required force structure [Loerch and Coblentz, 2002] and [United States Army, 1995]. This analysis includes assessments of force affordability, supportability, and executability [United States Army, 1995]. This phase of qualitative analysis may take under advisement quantitative analyses that consider risks and trade-offs across competing force structure options (see for example [Helms, 2012]), though this type of input is neither mandatory nor habitual.

The final result of TAA is a force structure recommendation for each year of the program objective memorandum.

2.6 Limitations in Defense Planning

Perhaps the greatest limiting factor in any force provision analysis is the construction of demand functions which drive force structure requirements and readiness policy decisions. We discussed in section 2.4.3 the current method for constructing demand functions, the integrated security construct. Critiques of such an approach focus on two distinct, but interrelated dimensions of the ISC approach—the process for constructing a single ISC and the number of futures considered for analysis. As we will discuss further, addressing issues related to ISC construction can ameliorate the limitations related to the number of futures considered for analysis.

We discussed in section 2.4.3 the process used to construct an ISC. There are four key variables which drive the development of an ISC demand function—the nature of events which will occur; the duration of these events; the timing of the events; and which resources are required to succeed in these events. Recall from our previous discussion that these inputs are pre-determined.

Such a process, which uses pre-determined scenarios, drawn from a small set of possible scenarios is subject to bias, placing undue attention on specific events, such as conflict with a particular adversary (at the expense of others) [Lempert et. al., 2003]. Frequently, this attention to specific events focuses analysis on the performance of *desired* (emphasis added) strategies.

The manner in which these ISCs focus on specific adversaries is relevant to the discussion of the theme of our research, robustness. By specifying not only the adversaries to be faced in the future, but also the timing of the interventions against those adversaries, policy makers can limit the effect of these interventions on force structure decisions, a criticism leveled by Davis [2002] among others.

For example, consider a planning construct that specifies adversaries X and Y. By constructing an ISC that ensures that interventions against these adversaries do not overlap, policy makers thus ensure that force structure requirements (to the extent that such a thing exists) are less than would be indicated if those interventions overlapped. Since we do not have a crystal ball, we cannot know for sure if those interventions would overlap or even if they would occur, but ignoring the possibility of simultaneous intervention limits the range of potential scenarios to which the force could intervene,

thus limiting a priori the robustness of potential responses. Now, it may be that based on some value judgment, responding to both events would not be desirable, but the current approach to constructing futures (and specifying strategies) does not support making such an assertion.

Limiting the scenarios used for analysis and specifying the timing of those scenarios necessarily limits the generalizability of any claims concerning the military's ability to execute the strategy. In particular, this prescriptive approach to analysis precludes discussion of robustness. Since robustness concerns the ability of a system to operate in any (though for practical purposes many) scenario, and the current analytic paradigm considers only one scenario, analysts cannot presently make statements about the military's ability to execute the strategy beyond the single scenario under consideration.

This criticism of narrowly focused analysis was put forth by Davis [2002] among others, and in recent years, DoD has taken steps to address this criticism. For example, the 2010 Quadrennial Defense Review [Department of Defense, 2010] considers three potential futures. While this is a positive step, it does not go far enough. While no method can possibly simulate and subject to analysis all possible futures which could confront the DoD, any method that seeks to make generalizable statements about the ability to execute the defense strategy must consider many possible futures. While the magic number of potential futures is likely beyond our grasp, given the present state of computing power available, it is certainly possible to analyze a great many more timelines of scenarios than currently under consideration.

2.7 Summary

In this chapter we described the roles and responsibilities of the Combatant Commands and military departments. As a force provider, the Army has a responsibility to determine the future composition of its force. Defense PPBE provides a venue for the Army to plan for its future force. However, current defense planning provides only limited ways to consider uncertainty in the nature and timing of potential future operations.

In the next chapter we review military force planning literature. Given the relative paucity of the force planning literature, we also review private sector fleet planning literature.

CHAPTER THREE: LITERATURE REVIEW: MILITARY FORCE MIX AND RELATED PROBLEMS

In the previous chapter we discussed the system by which combatant commanders request forces as well as the process by which the Army manages the resources it provides to those combatant commanders. We also discussed the planning processes in use to support the execution of this supply and demand system. In this chapter we discuss the current state of methods the Army and the Joint Staff use to support force structure decisions, methods used to analyze smaller scale force structure problems, and methods used to study a useful, nearly-analogous problem: the fleet size and mix vehicle routing problem.

3.1 Military Force Structure Analyses

As noted by [Carter et. al., 1997] (among others), and as we discussed in section 2.4.3 the Cold War and indeed post-Gulf War force structure analysis paradigm was to size and shape the armed forces to conduct some specified number of major regional conflicts and assume that the capabilities required to execute other types of operations were a lesser-included subset of those needed to succeed on these major regional conflicts.

In a series of planning exercises in 1996 and 1997, known as Dynamic Commitment, the defense analytic community began to relax the lesser-included assumption and considered the force size and force structure implications of so-called

"small-scale contingencies." Carter et. al. [1997] describe in detail the Dynamic Commitment series of planning exercises. We summarize this discussion below.

Dynamic Commitment began by noting that post-Gulf War military commitments were placing a significant demand on U.S. military forces, though none of these commitments was for a major regional conflict. Continuing with the assumption that the nature of operations in the near-future would reflect the nature of those in the recent past, planners developed a series of planning vignettes and associated force lists. The force structure analysis proceeded by randomly distributing this collection of vignettes over some notional future timeline and determining via seminar the response to each event in sequence.

In order to investigate the sensitivity of like results to the assumption that the near-future will look much like the recent past, the Center for Army analysis conducted a series of analyses to look at both the nature of smaller scale contingencies and the force structure required to succeed in those smaller scale contingencies.

The first of these studies, Stochastic Analysis for Deployments and Excursions (SADE) [DuBois and Kastner, 2000] sought to apply queuing theory to the stochastic modeling of the occurrence of smaller-scale contingencies. The methodology determined the inter-arrival times for all events and the event-type and duration distributions for six types of operations then simulated the occurrence of operations using these distributions to determine the distribution over time of the number and nature of simultaneous operations.

The second of these studies, Stochastic Analysis of Resources for Deployments and Excursions (SARDE) [DuBois, 1999] built on the SADE analysis to examine the force structure implications of a future in which the Army would be involved in numerous, simultaneous smaller-scale contingencies. In SARDE, analysts determined for each type of operation a mission, task-organized force (MTOF), a list of force structure elements required to succeed in each type of operation modeled in SADE. By applying the MTOF to the SADE simulations, analysts were then able to determine, for each resource type, a probability distribution of the number of each unit type used simultaneously.

[Loerch and Coblentz, 2002] built on the SADE/SARDE suite of analyses to develop a method for analyzing force structure decisions in the context of both major regional conflicts and smaller-scale contingencies. Their analysis proposed a two-stage, recourse stochastic program. The first stage considered force structure decisions with respect to the deterministic requirements of major regional conflicts; the second stage assigns units to tasks in the smaller-scale contingencies. The first stage of the model is used to determine an end-strength-constrained feasible solution while the second stage minimizes the expected value of the risk associated with an optimal allocation of the feasible force structure to the tasks in the SSCs.

The Army developed Marathon in response to the realization that readiness policies influence the resources needed to meet anticipated demands. Marathon has two methods of analyzing force structure: capacity analysis and requirements analysis.

Capacity analysis provides descriptive demand satisfaction and unit utilization data given a specified structure, fixed demands, and a pre-determined readiness policy. Marathon requirements analysis determines the force structure required to meet some single collection of demands based on three inputs— a time series list of demands by resource type; a time-based readiness policy; and a matrix of component proportions for each resource type.

The logic by which Marathon generates a force structure based on these inputs is described below. For further information on Marathon functionality see [Spoon, 2012] and [Spoon, 2011].

The Randomly Generated Requirements Informed by Past Operational Deployments (RANGER IPOD) [Helms, 2012] methodology was designed to address the single-future limitation in current Army force structure analysis. Essentially, RANGER IPOD creates many demand profiles using data-informed stochastic processes and identifies one "ideal" force structure for each demand profile using the Marathon requirements routine. The methodology then compares the performance of these numerous force structures against another set of stochastic process-based demand profiles to determine which of the generated force structures is likely to be most effective across a range of potential demand profiles. In other words, RANGER IPOD searches for a force structure that will enable the Army to have a robust ability to provide forces.

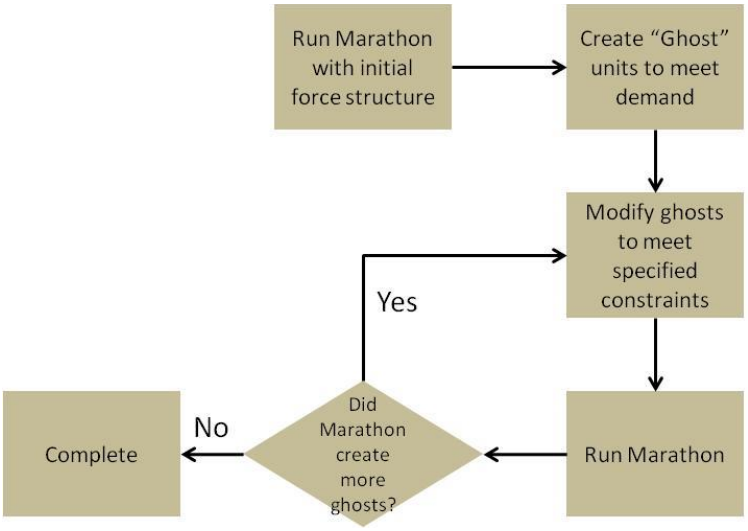

Figure 5: Marathon Requirements Analysis Logic

The Joint Staff uses a suite of tools to analyze Joint Force sufficiency. The Capabilities to Forces Integration Tool (CFIT) [The Joint Staff, 2012a] is a mechanism for representing combatant command requests for forces for individual events. The CFIT Force Management Tool (CFORM) [The Joint Staff, 2012b] contains force structure demand data, force management actions, and readiness policy attributes for US forces on a future of specified events over a prescribed timeline. The functionality in the Mitigation Options Selection Tool (MOST) [The Joint Staff, 2012d] allows service force structure planners and analysts to determine how to supply forces to prescribed demands given constrained resource inventory.

[Southerland and Loerch, 2014] developed an integer program to determine how to reduce inventories in Army force structure to meet some directed overall personnel

reduction target. The data for the optimization are derived from multiple instances of Marathon with varied inventories.

[Checco, 2015] describes a "multi-stage optimization model to determine dynamic force size…while accounting for uncertainty of future demand." The optimization allows for the overall size of the force to change in response to conditions, and considers multiple classes of manpower. Most other methods treat overall force size as a fixed constraint, and focus on only military manpower.

3.2 Other Military Force Structure Analyses

[Wojtaszek and Wesolkowski, 2012] surveys the military fleet composition literature and describes three classes of fleet composition problems—finding the best fleet; determining how well a particular fleet will perform ; and determining the best schedule for retiring and acquiring fleet platforms. In this section, we review their findings with respect to two of these problems—finding the best fleet and determining the best schedule for retiring and acquiring fleet platforms.

3.2.1 Finding an Optimal Fleet

Approaches to finding a single, optimal mix of resources to accomplish some mission have generally focused on finding a mix of vehicles, for example armored vehicles in the British Army or tactical vehicles in the Canadian Armed Forces. The methods used to find these fleets are generally static, in that they consider a fixed collection of scenarios to identify a static set of requirements. We discuss here some linear and non-linear programming approaches as well as heuristic search methods to identifying optimal vehicle fleet mixes.

[Walmsley and Hearn, 2004] describe a mixed integer linear program approach to informing an armored vehicle fleet mix decision. The approach first determines total fleet requirements by mapping platforms to roles, roles operational units, and operational units to deployment types. Each platform is rated as either compliant or insufficient with respect to these three requirements for various scenarios. The MILPs either determine (1) a feasible fleet that complies with all requirements or (2) a fleet that maximizes compliance given some cost constraint.

[Ghanmi et. al., 2010], like [Walmsley and Hearn, 2004], uses a requirements-based approach to optimizing a vehicle mix. A MILP is used to determine minimum cost fleets that meet all requirements and a mixed integer non-linear program is used to find a maximally effective mix that meets cost constraints.

[Stuive at. al., 2010] describe an evolutionary algorithm, the non-dominated sorted genetic algorithm II (NSGA II), in a multi-objective optimization framework to find a Pareto curve describing the cost and performance characteristics of candidate vehicle fleets. Other problems solved using evolutionary algorithms are discussed in [Mazurek and Wesolkowski, 2009] and [Whitacre at. al., 2007].

3.2.2. Finding an Optimal Fleet Adaptation Schedule

The literature contains discussions of two broad approaches to determining how best to adapt a fleet over time. The first approach is to determine which changes are feasible over some given time frame and from these feasible changes, adapt the fleet as well as possible given some constraining factors such as budget. The other approach is to identify a single, target fleet and determine how to achieve this target over time.

[Barlow et. al., 2007] discusses a methodology for minimizing a risk metric over time while changing the configuration of a particular force for some scenario or future. The constraints are "discretized into a fixed number of changes per time period," and the set of all possible changes are then enumerated, creating a directed graph. The outcome of each possible change is determined through simulation and the optimal solution found by finding the shortest path, representing the minimum risk, through the directed graph.

[Brown et. al., 1991] describes a U.S. Army helicopter fleet modernization model. Fleet modernization occurs via new production, modifying parameters of existing production, upgrading platforms currently in the fleet, and retiring old platforms. The modernization model uses a mixed-integer linear program to determine a fleet mix for each year of a 20 to 30 year planning horizon. For each year in that planning horizon, the fleet must adhere to specified performance standards, such as fleet age and proportion of platforms at various levels of technology, without violating budgetary constraints.

[Wesolkowski et. al., 2009] uses a Non-dominated Sorting Genetic Algorithm II to identify potential aircraft fleet mixes. Multiple scenarios are generated using the Stochastic Fleet Estimation (SaFE) methodology [Wesolkowski and Billyard, 2008]. Fleets are then generated for each scenario using the genetic algorithm. Candidate fleets are evaluated for their robustness, in terms of the number of generated scenarios that can be completely satisfied by each fleet, and their adaptability, in terms of the number of other scenarios that can be fully satisfied by adapting the fleet within specified budgetary or other constraints.

In SaFE-Robust (SaFER), [Wesolkowski and Wojtaszek, 2012a] search for task start times that minimize the total fleet cost required to meet all tasks. SaFE for Steady-State Tasking (SaFESST) [Wesolkowski and Wojtaszek, 2012b] uses an evolutionary fleet scheduling model to determine the performance of a specified fleet given specified platform to task matching.

[Wojtaszek and Wesolkowski, 2011] describes a method to search for Pareto-optimal platform to task assignment that can meet all requirements in a number of stochastically generated scenarios. Solutions are evaluated in a multi-objective optimization framework. Three objectives are proposed—minimal fleet cost; minimal total task duration; and maximum flexibility in accomplishing tasks within specified time windows. Here, flexibility is defined as the number of "subsets of platforms contained in a fleet that can accomplish the task within the time window." Thus, fleets with more ways to service tasks are more flexible than fleets with fewer task-servicing options.

Rather than specify a single heuristic to generate solutions to the fleet mix problem, [Shaffi et al, 2011] relates a method to find good heuristics. A learning classifier system (LCS) applies heuristics to scenarios. By extracting scenario features and evaluating the goodness of the heuristics the LCS learns the conditions under which various heuristics perform well by relating heuristic performance with the extracted scenario features.

[Abbas et. al., 2008] describes a simulation-based methodology for determining how to adapt a fleet of platforms. The Resource Planning under Time Constraints (RPTC) model identifies for a series of scenarios, a list of non-dominated solutions, where each

solution lists some portfolio of platforms necessary to meet all requirements in a scenario. These solutions are then clustered and these clusters are further clustered and ordered based on the maximum number of platforms in any solution within the first set of clusters. This ordering then informs candidate fleet adaptations, referred to as the "capability evolution network."

3.3 Fleet Mix Studies

In this section, we discuss literature concerning the fleet size and mix vehicle routing problem (FSMVRP). Generally speaking, the FSMVRP is concerned with the numbers and varieties of platforms in a vehicle fleet and managing that fleet of heterogeneous resources to meet some demand for those resources.

While not directly related to force structure analysis, the FSMVRP provides a useful analogy for identifying approaches to improving military force structure analysis. As we discussed in chapter two, the problem of finding good force structures involves three factors—supply, demand, and policy. The problem of sizing and shaping a fleet of resources is, at its most basic, no different. Some collection of customers (demand) need to be serviced by some collection of resources (supply) and rules (policies) exist, such as in laws limiting the number of hours a truck can stay on the road at any given time (see, for example, [Kok, et al, 2010]. Given these similarities it is prudent to review approaches to finding optimal fleets.

[Golden et al, 1984] was the first to fully describe the FSMVRP. The objective of the problem is to minimize some cost function, typically composed of both fixed and variable costs, by determining not only how many of each of numerous types of vehicles

to purchase, but also how to route said vehicle in order to meet some known demand. The fixed costs are generally associated with fleet acquisition while the variable costs are generally associated with routing vehicles to meet customer demand.

[Jabali et al, 2012] suggests a useful delineation of approaches to the problem as either operational or strategic. In operational studies, the goal is to find some mix of vehicles that can meet current, day-to-day customer demand. This demand is generally fixed and known. Strategic fleet mix studies, on the other hand, are focused on longer-term shaping of a vehicle fleet to meet a range of potential future demands. Unlike operational studies, strategic studies do not assume a fixed, known collection of customers that require service.

Our research is focused on a strategic analysis of Army force structure. As such, strategic fleet mix studies are most relevant to our research, but are less prevalent in the literature than operational fleet mix studies [Jabali et al, 2012]. Approaches to solving fleet mix problems at the operational level are nonetheless informative. In the following sections we first review approaches to solving operational fleet mix problems, discussing later the few strategic fleet mix studies we uncovered in our review of the literature.

3.3.1 Operational FSMVRP Formulations and Solution Methods

Solution methods to the operational FSMVRP can be summarized as following one of three approaches— heuristic search and combinations thereof; heuristic search augmented by a search-controlling metaheuristic; and various sorts of inequality and column generation approaches within an integer programming framework. We discuss

below several unique methods for each of the three general classes of solution methodology.

[Golden et al, 1984] describes a series of heuristics for solving the FSMVRP as well as methods for determining a lower bound on the solution and an underestimate of the optimal solution. The heuristic methods include several savings heuristics, several giant tour algorithms, as well as an improvement algorithm. The savings algorithms begin by identifying some feasible set of routing sub-tours and determining the savings of combining sub-tours with overlapping demands [Clarke and Wright, 1964]. The giant tour algorithms create a solution in two phases—the first phase creates a single (giant) tour that satisfies all customer demands and the second phase partitions the giant tour into smaller tours satisfying the problem constraints. The improvement algorithm uses an initial tour solution and tries to improve the tours by exchanging edges between tours [Golden et al, 1980].

[Ulusoy, 1985] describes a variant on the giant tour approach to find a minimal cost fleet and routing of said fleet. A giant tour is constructed and from this giant tour is constructed a network with nodes corresponding to arcs on the giant tour and arcs corresponding to single-vehicle feasible sub-tours. This network is then used to find a shortest path collection of sub-tours, which is then subjected to a no-cost arc, tour-cost improvement algorithm.

[Bookbinder and Reece, 1988] describe a non-linear, mixed integer program with Benders decomposition for solving the FSMVRP. The fleet size and mix is determined

via an assignment problem in the context of multiple depot distribution. By assigning customers to depots, a fleet is determined for each depot.

[Salhi and Rand, 1993] uses a heuristic search method to iteratively improve the utilization of vehicles and thus potentially reduce the number of vehicles required to service some demand. The algorithm uses a combination of reallocating customers within vehicle routes, combining routes for a single thus reducing the number of required vehicles, or by splitting routes.

Building on this work and the savings heuristics in [Golden et al, 1984], [Liu and Shen, 1999] describe a heuristic approach to solving the FSMVRP with time windows. Specifically, an insertion heuristic is used and the savings heuristic applied to determine the cost savings of inserting customers into already existing routes while maintaining time window and other constraint feasibility.

[Braysy et al, 2009] uses a three-phase, heuristic-based approach. In the first phase, an initial solution is found using a savings heuristic. The second phase tries to improve on the initial solution using a route elimination heuristic. And the third phase uses a general local search to improve further the second phase solutions.

[Osman and Salhi, 1996] describe a tabu search metaheuristic used to search for a near-optimal solution. Neighborhoods around an initial solution are determined using customer re-allocation techniques, defined as shift and interchange. The best changes in a neighborhood are then identified and compared to the initial solution. This process continues until some pre-defined number of iterations passes without improving on the then best solution. Other metaheuristic methods include [Liu et al, 2008] (genetic

algorithm), [Brandao, 2009] (tabu search), and [Repoussis and Tarantilis, 2010] (tabu search augmented by adaptive memory programming).

[Taillard, 1999] uses a heuristic column generation approach to solving the problem for a heterogeneous fleet of vehicles. The method begins by solving a series of *homogeneous* vehicle routing problems using an adaptive memory process driven by a tabu search [Taillard, 1993]. The set of solutions is then used to generate columns in a matrix identifying which customers are served by which vehicles. This matrix is then used to solve a Boolean linear program, the decision variables of which determine which tours for each vehicle type to select as part of the final solution.

[Choi and Tcha, 2007] extend the column generation approach to solve exactly a linear programming relaxation of an integer program. Columns are generated by way of dynamic programming techniques for the vehicle routes. This approach is the first we can find to use any sort of dynamic programming approach to generate potential solutions to the FSMVRP.

[Yaman, 2006] identifies a difficulty in evaluating heuristic solutions to the FCMVRP due to the "huge gap" in lower bound solutions. The proposed solution to this problem is to formulate a number of valid inequalities with lifting of those inequalities within an IP framework. [Pessoa et al, 2007] describes a branch, cut, and price algorithm, also within an IP framework.

[Simao, 2009] determines the value of various trucking resources from an approximate dynamic programming (ADP) model. The ADP manages the servicing of

uncertain demands. Outputs from this model are then used to determine the marginal value of drivers. These marginal values were then used to inform fleet mix decisions.

3.3.2 Strategic FSMVRP Formulations and Solution Methods

[List et al, 2006] details a robust, stochastic optimization methodology for determining how invest in radioactive waste disposal equipment given uncertainty. Uncertainty arises from variation in the timing and magnitude of waste disposal demands as well as in readiness of vehicles to dispose of that waste. These uncertainties manifest in the constraints of the formulation.

The goal in [Alvarez et al, 2011] is to find a solution that remains near-optimal when the problem parameters are subjected to changes. To achieve this, a mixed integer program is used to find solutions and then subjected to parameter variation based on the robust optimization method of [Bertsemas and Sim, 2003].

[Cambini and Riccardi, 2009] defines three factors that contribute to a fleet mix decisions—the ability to meet demand surges; the time to complete service tasks; and capability hierarchy. In this framework capability hierarchy refers to the possibility that some candidate platforms in the fleet can perform platform-unique tasks as well as tasks performed by other platforms. This definition is consistent with our definition of flexibility. Typically, the more flexible platforms are more expensive than less flexible vehicles. The solution to this problem is solved using a recursive optimization formulation.

[Cortes et al, 2011] optimizes the daily dispatch of operators within a simulation framework. The results of the simulation are then used to adjust fleet performance curves.

These performance turns are then used in off-line decision making to develop long-term policies.

Describing research by [Francis and Smilowitz, 2006], [Jabali et al, 2012] discusses a continuous approximation method to solving the strategic fleet mix problem. A notional demand scenario is modeled as a circle which has to be partitioned into rings and those rings partitioned into routes. Each ring can only be serviced by one type of vehicle while each route within a ring must be served by a single vehicle. Thus, the fleet size and mix is determined by finding an optimal, or near-optimal, partitioning of the circle into rings and routes by means of a mixed integer, non-linear program.

[Cheon et al, 2012] model a railcar fleet mix problem as a "long-term capacity expansion problem." The solution to this problem is determined over three stages. The first stage determines likely requirements for railcars over some time frame. The second part models the servicing of these requirements. The third, and final, stage of the model then determines a fleet management plan to include procurement decisions and other managerial tools such as railcar modification and transfer. All three stages of the model are integrated into a mixed integer linear program.

3.4 Discussion

The current approach to analyzing force structure has limitations. The primary limitation in force structure analysis arises from the fixed-future, fixed-force structure approach to analysis. That is, not only does the approach we have discussed assume a fixed future, the mix of units available to respond to that future is also fixed over the duration of the future.

This fixed-force structure approach runs counter to several realities inherent in defense planning and programming as well as counter to robust response to an uncertain environment—the final product of the programming phase of PPBE, the program objective memorandum, details a force structure for each year of the Future Years Defense Program [United States Army, 2011]; the military, and in particular the Army, adapts the mix of units in its force structure to respond to current and potential future challenges ; and theoretically fixing force structure limits the ability to utilize one dimension of flexible response to uncertainty, thus inhibiting modeling of adaptation to ensure robust system performance.

While it is important to note for the sake of completeness that a program force structure is actually one force structure per year over many years, it is more instructive to focus on the implications of modeling a single force structure over some planning horizon.

The theme of our research is ensuring robust response in an uncertain future through flexibility-enabled adaptation. So a natural question to pose is, “To what extent can the current approach to modeling consider robust response?” The answer, in short, is only in a limited manner. Much like in our discussion of the limitations in representing readiness management, assuming a fixed force structure over the planning horizon cannot consider how changing the force structure over time in response to changing conditions ensures an ability to provide forces when requested in support of the defense strategy. This suggests that a failure to capture force mix adaptation is another critical limitation of the current approach to analysis.

In this section we described literature on both military and civilian fleet sizing problems. While the dynamics in the problems facing those sizing civilian fleets are different than the dynamics facing military decision makers the methods used to model civilian systems and solutions to their problems are nonetheless informative.

Some of the system models we described here take a short-term, deterministic approach to solving for appropriate fleets. Others take longer term or stochastic approaches to solving this problem. The literature is richest for the short-term, deterministic models (operational models in the parlance of [Jabali et al, 2012]).

In either case, many of the solutions to fleet sizing problems involve heuristics, meta-heuristics, and in one case, hyper-heuristics. Heuristic approaches include algorithms for constructing solutions and approaches used to inform optimization models. Meta-heuristic methods build on these heuristics and utilize, among others, variations on the tabu search and genetic algorithms.

While the literature is richest for operational fleet sizing, there does exist a broad array of methods and models for strategic sizing and shaping of fleets. Many of these methods use an optimization framework, such as integer programs both linear and non-linear, and stochastic programs. Other solution methods include the use of simulations to determine fleet component value.

3.5 Summary

We have been thus far unable to identify any methods that use a combination of simulation and dynamic programming to determine how to adapt a fleet over time in the face of uncertain demand for fleet resources. This approach to fleet sizing and shaping is

the focus of our research. In the next chapter we describe our models for fleet demand, fleet deployment management, and fleet mix over time. Our overarching approach is an approximate dynamic programming model informed by a fleet deployment management simulation given uncertain resource demand modeled by a collection of stochastic processes. We discuss the methodology in chapter 5, but we first discuss approximate dynamic programming.

CHAPTER FOUR: APPROXIMATE DYNAMIC PROGRAMMING AND THE DIFFUSION WAVELET TRANSFORM

In this chapter we discuss the approach underpinning our methodology—approximate dynamic programming (ADP) with value function approximation (VFA). We review the ADP literature and describe the diffusion wavelet transform (DWT) VFA. We conclude the chapter with a discussion of the current state of DWT research, including present limitations in that research.

4.1 Dynamic Programming Overview

We define and describe here the field of approximate dynamic programming. This discussion is adapted from [Powell, 2011], [Balakrishna, 2009] and [Gosavi, 2003]. Other useful references include [Denardo, 2003], [Bellman, 2003], and [Busoniu, 2010] among others.

4.1.1 Dynamic Programming

Dynamic programming is a collection of techniques used to solve sequential decision making problems, often under conditions of uncertainty. Unlike mathematical programming approaches, which solve all periods in multi-period decision models simultaneously, dynamic programming solves these problems using a recursive formulation (Balakrishna, 2009). This recursive formulation is characterized by a statement of Bellman's Equation suitably modified to the needs of the problem at hand [Powell, 2011]—

Equation 1

$$V_t(S_t) = \max_{x_t \in X_t} (C_t(S_t, x_t) + \gamma E\{V_{t+1}(S_{t+1})|S_t\})$$

In the above equation, variables are indexed by time t, S is the state, V is the value, x is the decision variable, C is the one-step contribution, and γ is a discount factor.

Dynamic programs are defined by state variables; decision variables; a state transition function; an objective function; and, optionally, exogenous information processes. State variables describe the information upon which any decision is to be made, the information that must be known to make a 'good' decision. The goodness of any decision is determined by the objective function. The transition function describes how the state changes given any decision and any relevant exogenous processes. Dynamic programming models with discrete states and decisions belong to a class of models known as Markov Decision Processes.

Dynamic programs may seek to identify optimal decision over either a finite or an infinite horizon. State transitions may be either deterministic, in which case there is no exogenous process acting on the state transition, or stochastic. In any case the goal of a dynamic programming is to associate with any state the optimal decision to make in that state. Through its recursive formulation, a dynamic program finds an optimal decision given a current state by considering both the immediate contribution of making any decision and the potential future contributions that arise from having made that decision.

4.1.2 Markov Decision Processes

Dynamic programming is one solution method to a class of problems known as Markov or semi-Markov decision process (MDP and SMDP). MDP and SMDP are related to Markov and semi-Markov processes, the distinction being state transitions in the decision processes are influenced by decisions (often in a system control context) in some system whereas state transitions in the non-decision processes are not influenced by external control mechanisms.

Markov processes and MDP are characterized by three properties—the jump property, that is transitions between states occur regularly; the memoryless property, that is the transition from any state depends only on the current state, and does not depend on the states visited prior to the current state, depending; and the unit time property, that is all state transitions occur after unit time [Gosavi, 2003]. The semi-Markov process and SMDP are a more general class of problem than the MP and MDP in that the unit time property is relaxed. Specifically, the transition times are generally distributed random variables [Gosavi, 2003].

4.1.3 The Curses of Dimensionality and Modeling

As we discussed in the previous section Bellman's equation provides a compact, recursive solution to dynamic programs. In order to solve a dynamic program using Bellman's equation transition probability and transition reward matrices [Gosavi, 2003] must be specified. In the case of the transition probability matrix, the probability of transitioning from each state to every other state must be specified for every possible decision. To specify a transition reward matrix, a value must be specified for each state transition/action pair.

In either case, specifying the data needed to solve a dynamic program exactly using Bellman's equation is infeasible and the challenges therein, known as the curses of modeling and dimensionality, have been widely acknowledged in the literature (see for example, [Gosavi, 2003] and [Powell, 2011]).

The curse of modeling arises from the difficulty of *modeling* the relevant dynamics of the system in question. In dynamic programming parlance, specifying the state transitions, transition rewards, and transition times, given various decisions is subject to the curse of modeling. In complex, stochastic systems, specifying a model of that system is a challenging task [Gosavi, 2003].

A related challenge to the curse of modeling is the curses of dimensionality. Whereas the curse of modeling relates to the ability to specify a model, the curses of dimensionality relate to the computational feasibility of solving exactly a dynamic program. The computational burden of solving a dynamic program is related to three factors—the size of the state space; the size of the outcome space; and the size of the decision space [Powell, 2011]. For example, consider a system with 1,000 states, each of which is reachable from any other state via any one of ten decisions. In order to fully specify the transition function, a transition probability for 1 million potential transitions would have to be specified. In many practical cases, storing and recursing through the matrices necessary to solve exactly a dynamic program outstrips a computer's capacity to store and process such information.

4.2 Approximate dynamic programming

Research into working through the challenges posed by the curses of modeling and dimensionality has led to the development of approximate dynamic programming techniques. Unlike dynamic programming, which through Bellman's equation guarantees optimality, approximate dynamic programming techniques can provide only nearly-optimal solutions [Gosavi, 2003]. In this section we will discuss the approximate dynamic programming techniques most relevant to our methodology and how those techniques address the curses we discussed in the previous section.

4.2.1 Solutions to the curses of modeling and dimensionality

As we discussed in the section on dynamic programming, in order to find an exact solution using Bellman's equation a complete model, including transition probability matrices and transition reward matrices, is required. Research into solutions to the non-availability of explicit models has led to the development of so-called "model-free" methods. Model-free methods do not require an analytical form of the objective function, instead relying simply on the values of the objective function [Gosavi, 2003]. One particular model-free method is of interest to our research—simulation-based optimization. In simulation-based optimization, state transitions are generated through a simulation. Note here that we have mentioned the use of optimization to mitigate the need to explicitly represent state transitions, and acknowledge that model-free methods do not explicitly represent the objective function, but we have not yet discussed how to integrate this simulation into some value-based representation of the objective function. The solutions of most relevance to our research bridge the gap between solving the curses of modeling and dimensionality.

In general, there are three classes of solutions to the curses of dimensionality—state-space reduction via aggregation; function fitting; and function interpolation [Gosavi, 2003] [Balakrishna, 2009]. Aggregation reduces the complexity of a dynamic programming problem by combining multiple states into a single state. Frequently, this aggregation allows for exact DP solutions, which can then be disaggregated to obtain approximate solutions to the original problem [Powell, 2011]. Function fitting reduces the dimensionality of the problem by fitting model parameters to state space variables to generate an objective function. Function fitting methods include regression, among others [Balakrishna, 2009]. Function interpolation methods also reduce dimensionality by storing only a small number of representative value functions and interpolating to determine all others [Balakrishna, 2009]. Together, function fitting and interpolation reduce the dimensionality and provide a value-based representation of the objective function. Thus, function fitting and interpolation address both the curses of modeling and dimensionality. For more information on approximate methods, see [Busoniu et al, 2010].

4.2.2 The post-decision state variable

Before we discuss methods for approximating the value function it is important to first discuss a subtlety in the expression of Bellman's equation. Note in equation 5.1that this version of Bellman's equation requires evaluating the decision x that maximizes the expectation of some other value. In order to find this optimal decision, we would need to explicitly calculate this expectation for each decision. Calculating this expectation is frequently intractable [Powell, 2011], due in part to the curse of modeling. The solution

to this challenge lies in re-expressing Bellman's equation in terms of a post-decision state variable.

To understand what constitutes a post-decision state variable, consider the state transition from one decision cycle to the next. Two factors influence this transition—the decision made in the initial state, and any exogenous processes. The logic behind the post-decision state is to separate these two factors and apply only the influence of the decision on the current state, thus the "post-decision" state. In this modeling framework, the system operates just like any other MDP (or SMDP) except that observations are made after each decision. The state transition equations are expressed as follows:

Equation 2

$$S_t^x = S^{M,x}(S_{t,}x_t)$$

Equation 3

$$\boldsymbol{S_{t+1} = S^{M,W}(S_t^x, W_{t+1})}$$

In the above equations, S_t^x is the post-decision state at time t given decision x and S_{t+1} is the pre-decision state at time t+1 given post-decision state S_t^x and exogenous information W_{t+1}.

Given this discussion, we can express the relationship between the value of the pre-decision and post-decision states as follows [Powell, 2011]:

Equation 4

$$V_{t-1}^{x}(S_{t-1}^{x}) = E\{V_t(S_t)|S_{t-1}^{x}\}$$

Equation 5

$$V_t(S_t) = \max_{x_t \in X_t}(C_t(S_{t,}x_t) + \gamma V_t^{x}(S_t^{x}))$$

Equation 6

$$V_{t-1}^{x}(S_{t-1}^{x}) = E\{\max_{x_t \in X_t}(C_t(S_{t,}x_t) + \gamma V_t^{x}(S_t^{x}))|S_{t-1}^{x}\}$$

With this form of Bellman's equation we now have the expectation of a deterministic quantity.

4.2.3 Value function approximation methods

Consider a hypothetical situation in which all state transitions are known and deterministic. Computational feasibility aside, we could simply enumerate every possible path forward in a network and determine with certainty the value of being in any state at any time. Now consider a situation in which the state transitions are stochastic. If we were to use a simulation to drive our ADP, each iteration through the simulation would contain a different path forward through the states. From any given state, any number of paths forward would be possible, and we could estimate the value of being in that state for any of these paths. Each of these estimates would likely differ. The challenge we would then face would be how best to incorporate the collection of sample value function estimates. ADP uses an iterative value updating process driven by these sample value estimates. The iterative update process is described by the equation below—

Equation 7

$$V^n(S^n) = (1 - \alpha^n) * V^{n-1}(S^n) + \alpha^n \hat{v}^n$$

In this equation the estimated value at iteration n of being in state S, denoted by $V^n(S^n)$, is the weighted sum of two quantities—the previous estimated value of being in state S, denoted by $V^{n-1}(S^n)$; and the estimated value from the current iteration, denoted by $\hat{v}^n$. The choice of the weighting parameter, α^n, is of critical importance to ensuring the quality of any value function approximation. For further research on the selection of this parameter, see [George, 2006].

Given this iterative update method, it is important to discuss how to determine the within iteration estimates, the $\hat{v}^n$. The first step in this determination is to decide on a functional form, such as a linear or non-linear function. Once this form is specified, the parameters that define the function over the state-space or, in the case of function interpolation, the relevant portion of the state-space must be specified. This specification can come from regression or any other function fitting routine. One such routine, the diffusion wavelet transform, and its associated functional form are of particular interest to our research. The discussion that follows is adapted from [Balakrishna, 2009].

The diffusion wavelet transform represents a function by identifying "the best scaling and orthogonal basis functions." The functional form of this transform, in the context of value function approximation, is as follows—

Equation 8

$$\bar{V}(S|\theta) = \sum_{k=-\infty}^{\infty} c_{(j_0,k)}\phi_{(j_0,k)}(S) + \sum_{j=j_0}^{\infty}\sum_{k=-\infty}^{\infty} d_{(j,k)}w_{(j,k)}(S)$$

where $\phi_{(j_0,k)}$ is the scaling function, $w_{(j,k)}$ are the wavelet functions, and

Equation 9

$$c_{(j_0,k)} = \frac{1}{m}\sum_{i=1}^{m} \bar{V}_i(S)\,\phi_{(j_0,k)}(S)$$

Equation 10

$$d_{(j,k)} = \frac{1}{m}\sum_{i=1}^{m} \bar{V}_i(S)\,w_{(j,k)}(S)$$

Given this functional form, the following algorithm can be used to approximate the value function in an ADP framework (from [Balakrishna, 2009]).

Step 0: Initialization

0a: Initialize $V(S^0)$, S^0, and set n=0.

0b: Set a limit N_s, on the initial number of states to visit in order to initiate the function approximation scheme

Step 1: Obtain a sample path ω^n.

Step 2: Update S^nif n > 0.

Step 3: If N_s distinct states have not yet been visited, go to step 3e.

3a: If N_s distinct states are being visited for the first time, denote this set of states by S_{N_s}, and go to step 3b, else go to step 3c.

3b: Obtain the basis functions and the corresponding coefficients for the sample of N_s states. Go to step 3d.

3c: Obtain $V^{n-1}(S^n)$, the approximate value of being in current state S^n.

3d: Determine the next state, $S^{n+1} = S^M(S^n, x, \omega^n)$. If $S^{n+1} \in S^n$ then read $\bar{V}^{n-1}(S^{n+1})$ from the stored sample. Set $V^{n-1}(S^{n+1}) = \bar{V}^{n-1}(S^{n+1})$. Set $\phi^n = \phi^{n-1}$ and $\psi^n = \psi^{n-1}$. Go to step 3e. Else add S^{n+1} to the set S_{N_s}, as the last element and remove the first element of the set S_{N_s}. Using the updated state space sample, determine the basis functions, $\phi^n$and ψ^n, and update the coefficients, $c^n_{(j_0,k)}$ and $d^n_{(j,k)}$. Obtain $\bar{V}^{n-1}(S^{n+1})$ and set $V^{n-1}(S^{n+1}) = \bar{V}^{n-1}(S^{n+1})$. Go to step 3e.

3e: Solve

Equation 11

$$\hat{v}^n = \min_{x \in X} \left(C(S^n, x) + \gamma V^{n-1}\left(S^M(S^n, x, \omega^n)\right)\right)$$

and let x^n be the value of x that solves this equation.

3f: Update the value function.

Equation 12

$$V^n(S^n) = (1 - \alpha^n) * V^{n-1}(S^n) + \alpha^n \hat{v}^n$$

Step 4: Find the next state, $S^{n+1} = S^M(S^n, x, \omega^n)$.

Step 5: Set n=n+1. If n < N, go to step 1.

4.3 Discussion

Note that in step 3e of the preceding algorithm, we are told to solve an equation by finding the action, x, that minimizes the equation's value. However, step 3e does not specify how to solve the equation. This raises a critical question—"How should we solve the equation to find the optimal action?" To solve the equation, we must be able to estimate the value, $V^{n-1}\left(S^M(S^n, x, \omega^n)\right)$, for some set of actions, X. One possible solution is to estimate the value for all possible $x \in X$. However, given the curse of dimensionality, estimating the value for all possible decisions is likely computationally intractable for most problems.

[DeGregory, 2014] solves equation 11 by solving for only a subset of the decision space. The method describes a pre-process that fixes values for some of the decision variables. After setting these values, [DeGregory, 2014] uses information from the state variable to develop a small number of candidate solutions. The solution carried forward in the approximation is the best solution among this smaller list of candidates.

The solution-space reduction method seems a practical approach for solving problems without running afoul of the dimensionality curses. However, many questions remain about this approach, and particularly with respect to the DWT VFA. In particular, in the course of our research we identified two questions that warrant further investigation—"For how many decisions should the value be approximated?" and "How

does this number of candidate solutions relate to the size, N_s (from step 3, above), of the approximation?" These two questions provide the motivation for the computational aspect of this dissertation.

4.4 Summary

In this chapter we reviewed the literature associated with our computational approach to solving the problem described in chapter 2. We identified two unanswered questions relating to the diffusion wavelet transform approach to approximating value functions. These two questions provide the basis for the computational aspect of our research. In chapter six we discuss the experimentation that provided us with insight regarding these questions. In the next chapter we describe our methodology for applying DWT DVA to solve the military force mix problem.

CHAPTER FIVE: METHODOLOGY

Recall that our problem is to determine how best to sequentially adapt the mix of capabilities in the Army's force structure. To solve this problem, we developed an approximate dynamic programming formulation. We executed the formulation using a Python-based simulation with a value function approximation. In this chapter, we describe the simulation and its relationship to the DP formulation and value function approximation.

5.1 Simulation

To model the force structure adaptation problem we developed a supply and demand model in Python. In this model, as military missions occur, demand for forces increases; military units represent the supply; and force generation policies represent the rules by which units of supply are matched to demands.

5.1.1 Representing Military Units as Supply

Units of supply represent individual deployable elements of a given unit type. Each unit is represented by a supply object in the simulation. Each object is characterized by a collection of data, some of which are static and others dynamic. Static data include the unit type and the unit name. Dynamic data include location, which can include "Home" or a specific mission; status, which include "Non-Deployable", "Deployable", and "Deployed"; and cycle time. We update these three dynamic parameters for each unit

at the beginning of every time step. We discuss these dynamic data in further detail later, in our discussion of policies.

5.1.2 Representing Military Missions

In this methodology, military missions are divided into classes. Examples of classes of military missions include humanitarian assistance, peace enforcement, and homeland defense. Each class of mission is represented by a force list. Each force list, l, is characterized by a deterministic duration parameter, d_l which may be deterministic or stochastic; a frequency parameter, λ_l; and a list of unit types with associated required quantities. Each mission type has one, deterministic list of required quantities. An example force list is depicted below.

Table 1: Example Force List

Mission	Enforce Peace
Frequency	14
Duration	3

Unit type	**Quantity**
Tanks	2
Infantry	3
Logistics	1

The occurrence of each type of mission, l, is modeled as a Poisson process with parameter λ_l. By simulating the Poisson process for each mission type, we generate a list

of starting times for each mission. Coupling these starting times with the associated frequency parameters allows us to visualize the missions over time. Adding the quantities demanded for the various unit types allows us to create a demand vector for each unit type for each time step in the simulation.

5.1.3 Governing Supply with Policies

Policies govern two aspects of supply behavior—the progression through various states of readiness for each unit; and rules governing the assignment of units of supply to missions.

Readiness policies are specified through three parameters-- “Not Deployable”; “Cycle Max”; and “Deployment.” “Not Deployable” specifies the amount of time a unit remains in “Non-Deployable” status before entering “Deployable” status. “Cycle Max” describes how long a unit may remain at home before reverting to “Non-Deployable” status and is the sum of the amounts of time a unit may be in either a “Non-Deployable” or “Deployable” status. For example, if a unit has a “Cycle Max” of 24 months and a “Not Deployable” parameter of 9 months, that unit may remain in “Deployable” status for fifteen months before returning to “Non-Deployable” status.

The “Deployment” parameter dictates for how long a unit may be assigned to a mission before returning to the “Home” location and to a “Non-Deployable” status. In cases where a unit is assigned to a mission that ends before the unit’s remaining “Deployment” time reaches zero, that unit will return to the “Home” location and enter “Non-Deployable” status

While is it feasible and reasonable to specify different policies for different unit types, in practice this is rarely done for units managed on a rotational policy. Thus, for the purposes of this research, we allowed for the specification of a single readiness policy.

The rules for assigning units of supply to mission take account of the unit types required, the priority of the mission, and the relative cycle time of all deployable units of supply. The assignment of units to demands follows a myopic, greedy heuristic. Thus, for each given unit type, missions are sorted in priority order, deployable units of supply are ordered descending by cycle time, and the two lists are matched until either all mission requirements have been satisfied or no deployable units of supply remain.

5.1.4 Putting the pieces together

A single iteration of the simulation can be described by its demand, supply, and policies. At the beginning of any iteration, supply and demand are initialized.

Demand initialization executes, for each mission class, the Poisson process. The result is a data structure detailing the unit requirements for every unit type, for every time step in the simulation. As the simulation progresses, we track mission requirement satisfaction. This tracking is discussed in greater detail in section 6.2.

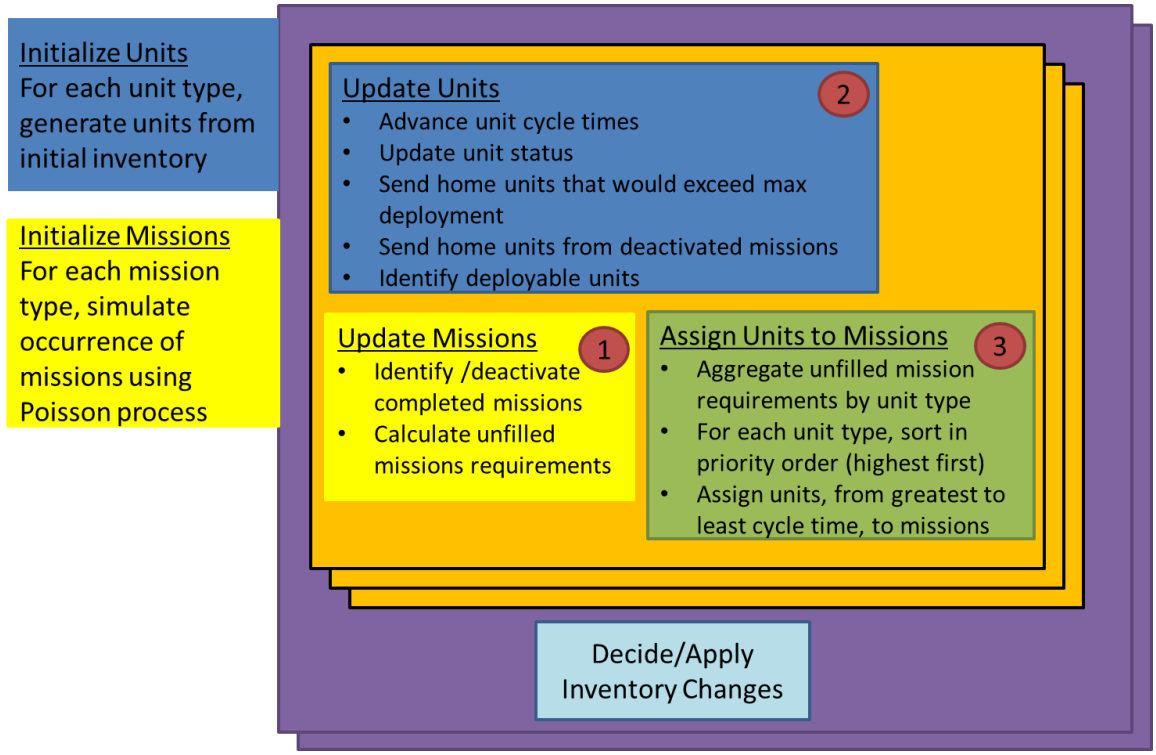

Figure 6: Simulation Overview

Supply initialization takes as input an initial supply portfolio which details the initial quantities of units for each unit type. For each unit type, the created units' readiness cycle times are evenly distributed across the readiness cycle. For example, if the "Cycle Max" parameter is 24 months, and a unit type has three units, the units' readiness will be 0, 8, and 16 months into the 24 month cycle. To ensure sufficiently realistic starting conditions for performance measurement, we specify a burn-in period during which no performance data are recorded. We describe these performance data in greater detail in section 5.2.4.

After initialization, the simulation executes the same procedure for every time step. The total number of time steps and the length of each time step are specified as

parameters. At the start of each time step, we update each unit of supply and update demands.

To update supply, we advance the cycle time of each unit by the specified time step parameter. Given this advance, we then update the status of each unit, as dictated by the policy parameters discussed in section 5.1.3. Finally, we determine which units of supply are eligible to be assigned to demands.

After updating the supply based on the specified policy parameters, we then update mission requirements. Missions that begin during the time step are activated. We then calculate the total number of units of supply assigned to each active mission. We use this calculation to determine the list of unsatisfied demands.

Finally, we assign deployable units of supply to demand, using the myopic, greedy heuristic as described in section 5.1.3.

5.2 Dynamic Programming Formulation

In the language of dynamic programming, the simulation, with one additional piece of functionality, the decision function, provides us a representation of the transition function. Recall from chapter 4 that the transition function defines transitions between states given exogenous processes and the decision function (see equations 4.2 and 4.3). In the simulation, missions and policies represent exogenous processes. We next describe the decision function and then describe the data collected in the simulation and their relationship to the state variable and the objective function.

5.2.1 Decision Function

As the simulation progresses from time-step to time-step, assigning ready units of supply to unsatisfied mission requirements, we developed a decision variable, X_t, to make, at regular intervals, changes to the portfolio of units types in the inventory. We define this interval via a decision interval parameter. Typically, as is the case with this research, the supply-demand interval is one month and the portfolio decision occurs every twelve months.

At each portfolio decision interval, we decide, for each unit type, the number of units to add or remove from the inventory, including the possibility of no changes to the inventory. We denote the decision for each unit type, u, as X_{tu}. Within the simulation, we defined a heuristic to apply the force structure decisions at each decision interval. For added units, we simply create a new supply object, and initialize the readiness cycle time of each unit to zero. For unit types that need to shrink their inventory, we find the least ready units with a "Home" location and set the status of the appropriate number of these units to "Inactivated."

This decision function, coupled with the functionality we discussed in section 5.1, provide a method for modeling changes to the system state variable. We next discuss our definition of the system state variable.

5.2.2 Decision Variable Constraints

The force structure portfolio decisions must meet a number of constraints. Specifically, three classes of constraint define the feasible region for the (annual) force structure decision—

1) Total change in personnel strength must not exceed some net change parameter,

2) Total new structure must not exceed an annual feasibility parameter,

3) and the range of potential changes for any unit are limited to that which can execute for a unit of its size class.

The third class of constraint provides us a way of representing a bit of reality in our methodology. At their most basic, these constraints ensure we do not grow too many large units, as defined by their echelon (company, battalion, brigade, in ascending order of size) beyond that which the Army can feasibly execute. In general, larger units are composed of a number of smaller units. This collection of smaller units may cover a wide range of capabilities, thus making these larger units more complex to build. For example, an armor brigade combat team is composed of five different types of battalions, each with different classes of personnel, and is thus much more difficult to build than an infantry company, which is composed overwhelmingly of infantry personnel.

5.2.3 State Variable

Within the simulation, we needed to identify a collection of variables that would change between force structure decision intervals and whose association with some notion of value would allow us to discriminate among potential force structure decisions. Ultimately, we decided to model the system state as a vector. This vector contains one entry for each unit type and describes the ability of each unit types to meet both readiness requirements and forecast mission requirements. The value for each unit type consists of three components: the fixed, pre-specified readiness requirements; a relation between

forecast and current mission requirements; and a relation between the current and theoretical maximum number of deployable units.

To define the state variable we define the components as follows:

1. Readiness requirements for unit type u: R_u
2. Mission requirements: Q_{ut}
3. Maximum deployable units: Y_u ,where

   Y_u=$\frac{CycleMax-MinDeployable}{CycleMax} * (Inventory_{u,T-1} + X_{Tu})$

4. Currently deployed units: C_u

Then, $S_u = R_u + \max_{t\in[T,T+w)} Q_{ut} - Q_{uT} - (Y_u - C_u)$ and S=(S_u). In this equation, $\max_{t\in[T,T+w)} Q_{ut} - Q_{uT}$ accounts for changes in mission requirements in some window while $Y_u - C_u$ accounts for forecast changes in deployments in that same window. Taken as a whole, the system state is a forecast for each unit type of its ability to satisfy readiness and forecast mission requirements.

5.2.4 Objective Function

As is typical of a dynamic program, our objective function is defined as a recursive relationship, namely the value of any given state given some decision is the sum of the immediate contribution of the decisions given the current state and the value of the subsequent state given the decision (see equation 4.1).

Given this recursive formulation, the contribution function given the current state and the decision function is thus the critical element of the objective function. Our contribution function is the weighted sum of two elements, demand satisfaction and force

readiness relative to some specified requirement, $C_T = w_r * r_T + w_d * d_T$, where $w_r + w_d = 1$.

Demand satisfaction is the first of two components in the contribution function. At the end of every time step in the simulation, we record the total number of units of each type assigned to each mission demand and weight both this satisfaction and the requirement by the total number of personnel in the relevant unit type. As the simulation progresses from decision epoch to decision epoch, we are thus able to calculate the total satisfaction across all missions in the decision interval. This total satisfaction is expressed as—

$$\frac{Total\ Personnel\ Months\ Satisfied}{Total\ Personnel\ Months\ Required}.$$

For example, if a mission requires three 20-person units for three months, the total requirement would by 180 person-months; if two-thirds of this demand is met, demand satisfaction is 120 person-months.

At each force structure decision epoch, the mission requirement satisfaction component of the contribution is $d_T = \frac{\sum_m Sat_m}{\sum_m Req_m}$, where m is the set of all missions that were active at any point in the decision interval.

The second component of the contribution function is readiness. At each mission assignment interval in the simulation we record the number of units that would be deployable within some window. For each unit type, we compare this number to a specified readiness requirement, specified as a number of units of each type that must be deployable within that same window. Thus, for each time step, we are able to calculate

the total force readiness relative to a requirement—$r_t = \frac{\sum_u \min(Ready_{ut}, R_u) * Size_u}{\sum_u R_u * Size_u}$, where u is the set of unit types and $Size_u$ is the number of personnel in a single unit of type u. The readiness component of the contribution is the minimum readiness over the period between the current and subsequent force structure decision, $r_T = \min_{t \in (T, T+1]} r_t$.

5.2.5 Exogenous Information Processes

The exogenous information in our formulation affects two components of our system state as described in section 5.2.3. The mission simulation affects component 2 of the system state, or mission requirements. The overall application of the simulation, advancing unit cycles and assigning units to missions, affects component 4 of the system state, or the number of units currently deployed at the time a unit inventory decision is made.

5.3 Value Function Approximation Overview

The goal of an approximate dynamic program is to find an optimal, or near-optimal, mapping of decisions to states. In this research, we use the Diffusion Wavelet value function approximation to determine this mapping. The details of the Diffusion Wavelet scheme are given in section 4.2.3 and we will not repeat those details here. Instead, we describe the process by which we initialize and leverage simulation data for the approximation.

5.3.1 Iteration and Convergence

Within the approximation scheme, we endeavor to reach a point at which successive approximations do not vary greatly and of which we are reasonably satisfied that solutions to Bellman's equation arising from the approximation are near-optimal. In

practical terms, we address each of these considerations with separate mechanisms within our methodology.

We use equation 4.7 to update our value function approximation. This equation determines the current approximated value as the weighted sum of the value estimate from the current iteration and the estimate from the most recent iteration. The weighting factor, α, is critical in reaching a convergent approximation. If we are careless with how we select this factor, we may experience a phenomenon known as *apparent convergence* [Powell, 2011]. In apparent convergence, the approximation may reach a period of stable approximations which mimic convergence but are merely locally stable. In these situations, the approximation is likely to break out of this period of stability to better approximations. One potential cause of this is a weighting factor that decreases too quickly and thus exerts an unwanted influence on the approximation.

The other obstacle to reaching a convergent, near-optimal approximation is sufficiently searching the state space to provide us confidence the solutions arising from our approximation are, in fact, near-optimal. In the next section, we discuss how we overcome this obstacle.

5.4 Implementation of Dynamic Programming

As we discussed above, to find a near-optimal policy we proceed through a number of simulation iterations. This iteration progresses through two phases—an exploration phase and an exploitation phase. The goal of the exploration phase is to search the decision space to provide a foundation of data for the exploitation phase.

In the exploration phase, force structure decisions are made by one of two mechanisms—an exploratory mechanism and function approximation-based mechanism. In the exploratory mechanism, a random, feasible (as described in section 5.2.2) decision is taken. As the simulation progresses, we track the decisions taken and the realizations of the contributions. At the end of the iteration, we use these contributions to estimate, post-hoc, the value of taking the various decisions in the relevant states. The random decisions during the exploration phase ensure we have searched the decision space.

With the function approximation-based mechanism, we use the current value function approximation to make a force structure decision. Specifically, we use the value function approximation to determine which decision maximizes Bellman's equation.

The exploration phase occurs over some pre-defined number of iterations. An iteration uses the exploratory mechanism with probability, p, where p is determined from a decreasing function whose value is close to 1 early in the exploration phase and reaches 0 at the end of the exploration phase. Within a single iteration, the decision mechanism is constant.

The exploitation phase uses the same mechanics as the exploration phase with one key modification—the probability of taking a random decision, i.e., exploring, is zero. The exploitation phase continues for a fixed number of iterations or until the value function approximation converges.

5.5 Summary

In this chapter, we described our methodology and model formulation for solving the Army force structure adaptation problem. In the next chapter, we describe a case

study through which we demonstrate the feasibility of the methodology for solving the problem.

CHAPTER SIX: EXPERIMENTATION

Recall the two objectives of this research: to demonstrate the feasibility of the approximate dynamic programming approach to solving the force structure adaptation problem and to examine the effect of value function approximation size on solution quality. Our base hypothesis is that, within the range we tested, larger approximations will yield better solutions. In this chapter, we describe the experimentation we conducted to achieve these objectives, and test and refine the base hypothesis.

6.1 Experimentation Overview

Before we could test the solution quality of the DWT VFA, we first had to train the VFA. Thus, the each experiment occurred over three distinct phases—an exploration phase, in which we initiated the value function approximation; a learning phase, in which we trained the value function approximation; and a "learnt" phase in which we assessed the solution quality of the VFA by applying the approximation in an approximate dynamic programming context against a number of test scenarios.

6.1.1 Exploration and a Decision Heuristic

One challenge we had to overcome in this research was the sheer size of the decision space. As we describe below, we simulated a force with twenty different unit types. Based on this and application of the decision space constraints, our decision space

had 804,651 possible decisions. This number of potential decisions would far outstrip the computational ability of our system to estimate the value of every post-decision state.

To overcome this decision space challenge, we developed a heuristic to drastically reduce the number of decisions we evaluated. To develop our reduced decision space, we used the state space, which indicates relative surplus or shortage of inventory, to rank order the 20 unit types. We then selected the five unit types with the greatest projected surplus and five unit types with the greatest projected shortage as our candidates for reduction or growth respectively.

Decision space constraints limited the total growth or reduction of each unit type to one unit per decision, and our total number of new unit could not exceed three. Applying these constraints to our two candidate populations reduced our decision space to 226 decisions to be evaluated.

During exploration simulations, we selected randomly from this list of 226 decisions. For exploitation simulations, we evaluated these decisions using the diffusion wavelet transform value function approximation (DWT VFA) and chose the decision that took us to the most valuable post-decision state. Given the approximation sizes we describe below, we were able to evaluate all candidate decisions with one single application of the DWT VFA.

6.1.2 Learning Phase Description

Before we describe the details of the learning phase, we begin with some terminology. When we use the word “scenario,” we mean an arraying over time of the simulated mission occurrences with their associated force requirements. A simulation

dynamically applies specified force generation parameters to the inventory of unit, assigns units to the scenario missions, and adjusts unit inventories by applying the appropriate decision function. A single iteration, also called a state visit, is the application of a single force structure inventory decision at a state. Thus, each simulation is composed of one or more iterations.

Each simulation is defined by four major factors-- the scenario; the inventory of units; force generation parameters; and a decision function. We created each scenario by simulating the occurrence of eight mission types. The Poisson parameter and force requirements for each mission type are described below. We started each simulation with the same initial inventory, across 20 unit types of the same size, which is described below. We applied a 24 month force generation cycle: each unit was deployable 6 months after the beginning of a cycle and deployments were capped at 9 months. Decisions in the simulation were random, as described above, or derived from the VFA-based optimization described in chapter 5.

Table 2:Mission Requirements

Unit Type	Mission 1	Mission 2	Mission 3	Mission 4	Mission 5	Mission 6	Mission 7	Mission 8
A			3	1			3	3
B			2	1			2	2
C	2	2	6	2	1		6	6
D		1	3	1		1	3	3
E		1	3	1			3	3
F								
G	1	2	3	1	1	1	3	3
H	1	2	3	1		1	3	3
I	1	1	3	1	1	1	3	3
J	1	1	2	1		1	2	2
K	1	1	3	2	1	1	4	4
L		1		1	1			
M			4	4			4	4
N			2	1			2	2
O								
P								
Q			3	1			3	3
R				1			2	2
S			3	1			3	3
T		1	3	1			1	1
Frequency	27	90	110	162	60	90	45	27
Duration	6	36	12	15	6	18	1	2

The learning phase had three distinct sub-phases—an initialization phase, intended to initialize the VFA by visiting many states with random decisions; a transition phase, during which we slowly transitioned from random decisions to VFA-based decisions; and an exploitation phase, during which we used VFA exclusively to make inventory decisions.

Table 3: Inventories and Readiness Requirements

Unit Type	Initial Inventory	Alt Inventory 1	Alt Inventory 2	Readiness Requirement
A	9	8	2	6
B	8	0	5	3
C	9	0	13	7
D	6	14	16	3
E	9	0	16	3
F	5	4	12	10
G	10	5	12	3
H	10	18	1	3
I	9	5	11	3
J	12	5	12	2
K	10	18	6	5
L	5	7	11	1
M	11	16	3	2
N	8	12	2	3
O	5	9	12	2
P	5	5	12	3
Q	7	15	4	3
R	8	1	8	3
S	10	7	3	6
T	7	14	2	1

In the process of determining exactly how to structure the learning phase (beyond the three phases and the application of decision functions therein), we identified two potential options. Each option lent itself to studying different aspects of the problem and each option had the potential to run afoul of the curse of dimensionality. The first option, to specify phase transition criteria and run simulations until all transition/termination criteria had been met would have allowed us to study the convergence properties of the value function approximation in greater detail, at the potential cost of having to run orders of magnitude more simulations than time would permit. The second option, to

specify phase lengths a priori and run for a fixed number of simulations would allow us to assess solution quality with reasonable controls and within manageable time.

After much consideration and informal experimentation, we opted to specify phase lengths a priori. All results discussed in this dissertation are based on an initialization phase of 5 million iterations, a transition phase of 2 million iterations, and an exploitation phase of up to 5 million iterations (though in all cases we ran only 2 million iterations).

In each case, we observed the mean squared error of the gradient and monitored this MSE for signs of convergence. [DeGregory, 2014] describes convergence within a band. We applied the convergence with band approach to identify our stopping conditions. For each measurement of the squared error of the gradient, we calculated a mean (MSE) and plus-or-minus one standard deviation band, using the previous 500 observations of the squared error. We then counted the number of consecutive MSE measurements that fell within its band.

6.1.3 Assessing the Impact of Approximation Size on Solution Quality

To assess the impact of approximation size we needed some way to compare outcomes against some common feature. Tc achieve this, we generated 1,000 scenarios using the Poisson processes described above. We then simulated each scenario applying the base force structure and force generation rules described above, applying the DWT value function approximations that resulted from the learning phase of each test case. This simulation allowed us to compare solution quality within the DWT VFA.

However, to truly demonstrate goodness of the DWT VFA approach, we needed to compare not only within variations on the DWT VFA, but with other approximate dynamic program solution methods within the literature. To determine the potential goodness of a VFA approach, [Powell, 2011] suggested asking, "Does a value function add value?" In this context, he makes repeated reference to a myopic policy. Given this discussion, we decided that comparing outcomes with our DWT VFA approach to outcomes applying a myopic heuristic would be useful in establishing the goodness, or added value, or our DWT VFA approach. Put another way, the myopic policy would serve as an external control on statements of solution quality.

Having devised a way to establish the added value of our approach, we then needed to determine a way to mitigate the possibility that any added value we identified was an artifact of the setup of our study. In considering this need, we identified two critical factors that might lead to misrepresenting the goodness of our approach—each of the 1,000 scenarios we simulated started with the same inventory; and stated preferences for outcomes measured in the objective function were fixed. In other words, we needed to ensure that our statements of goodness were robust to varying initial conditions and to changes in decision-maker preferences.

To investigate the robustness to initial conditions, we randomly generated two additional force structures, listed above as "Alternate 1" and "Alternate 2", each with the same total number of units as the structure listed as "Initial." We then performed the simulation of the same 1,000 scenarios, substituting the initial structure with each alternative. We did not re-learn a value function approximation because the initial state of

the simulation would be captured by the state space representation that resulted from the learning phase.

To investigate the robustness we decided to reverse the weights. Since state values are determined by the objective function, which is a function of the weights, we needed to relearn the value function approximation using the new weights. Thus, we re-executed the learning phase of our model for multiple approximation sizes. We then re-simulated the 1,000 test scenarios for each combination of approximation and initial inventory. In total, we learned 10 value function approximations, which are summarized below.

Table 4: Approximations Learned

Experiment	w_d	w_r	250	375	500	625
1	0.7	0.3	x	x	x	x
2	0.3	0.7	x			x
3	0.1	0.9	x			x
4	0.9	0.1	x			x

6.1.4 Computational Time

The first consideration we examined during our experimentation was computational time. As we simulated the 1,000 scenarios for each approximation, we noted the amount of time needed to simulate the scenarios to completion. We ran the simulations on a PC running 64-bit Windows 7 with 32 gigabytes of RAM and a 3GHz Intel Xeon 5760 processor. Although the processor had a dual core, we did not use any

multi-threading to reduce computational time. The average simulation times for each of the four approximations across the 1,000 scenarios are listed below.

Table 5: Average Simulation Time Over 1,000 Simulations

Size	Avg. Simulation Time (sec)
250	5.4
375	9.1
500	14.6
625	22.5

It is clear from the data above that approximation size is a critical factor in determining computational time. During development of the program, our speed benchmarks indicated that about 80% of the computational time for any simulation was due to QR factorizations required to calculate basis and scaling functions in the diffusion wavelet transform.

We should note here that with the diffusion wavelet transfer, the simulations times are a function of the approximation size and not a function of the length of the state space vector. In our case, the state space vector had 20 entries. A single simulation for a problem with 40, 50, or any arbitrarily larger (or smaller) number of entries would take the same amount of time. This independence from problem size is part of the appeal of the DWT VFA.

However, computational resources are finite, so we should choose an approximation size that balances computational burden with solution quality. At the speeds listed above, running the 100,000 simulations with a 625 state approximation for the third sub-phase of our learning phase took about 26 days while the same number of simulations with a 250 state approximation took less than one week. So, when considering practical application of this methodology, we need to assess to what extent, if any, the additional computational time improves simulation outcomes. The remainder of this chapter presents our analysis of simulation outcomes.

6.2 Learnt Phase: First Experiment

Our first experiment utilized an objective function weighted 70% for satisfying current mission requirements and 30% maintaining readiness for future contingencies. As discussed above, we first learned four value function approximations. We then executed the learnt phase for each value function approximation and compared the quality of decisions applying these approximations to the quality of decisions applying a myopic heuristic.

6.2.1 Solution Quality

To assess solution quality, we tracked two data point from each scenario: total mission requirements satisfied over the 20 year simulation and average readiness over the same period. Initial inspection of these data indicated that for each o each of the four DWT value function approximations outperformed the myopic heuristic in all scenarios for both data points. Given this finding, we decided to use the ratios of mission

satisfaction and readiness relative to the myopic heuristic as our modeling metrics. We thus tracked percent increase of mission requirement satisfaction and readiness.

Given the experimental setup described in section 6.1, we decided to treat our experiment like a fixed-effect model with a complete block design. In this setup, each of the four DWT VFAs (denoted by i below) was a treatment and each of the 1,000 scenarios (denoted by j below) was a block. With this setup, we modeled each observation as: $y_{ij} = \mu + \tau_i + \beta_j + \epsilon_{ij}$, where μ is the overall mean, $\tau_i$is the mean for treatment i, β_j is the mean for block j, and $\epsilon_{ij}$is an error term.

We first set out to determine to what extent approximation size contributes to better satisfaction of mission requirements. We needed to determine if the treatment means were different. To this end, we proposed the following hypothesis:

$$H_0: \tau_{250} = \tau_{375} = \tau_{500} = \tau_{625}$$

$$H_1: \tau_k \neq \tau_l \text{ for some k,l} \in 250, 375, 500, 625$$

To test this hypothesis we conducted an analysis of variance. With an F-statistic of 405 and 3 degrees of freedom, the ANOVA results indicated that we could reject our null hypothesis at the α=.01 level of significance.

Since we rejected the null hypothesis of equal treatment means, we then needed to determine which treatment means differed significantly. We performed Duncan's multiple range test. This test allowed us to differentiate between individual treatment means. The results of this test indicated that each of the four treatment means was significantly different from the other means. Further, the results indicated the following

order, from best to worst, of the four treatments: 625, 500, 250, 375. The individual treatment means from Duncan's multiple range test are detailed in below.

Table 6: Experiment One, Mission Requirement Improvement Relative to a Myopic Heuristic

Approximation	Base Inventory	Alt Inventory 1	Alt Inventory 2
250	1.8%	4.3%	3.9%
375	1.6%	3.4%	3.9%
500	2.1%	7.7%	3.1%
625	3.2%	13.3%	4.8%

We then repeated the experiment for each of the two alternate initial inventories. For alternate inventory 1, we were able to reject the null hypothesis of equal treatment means at the α=.01 level of significance (F=2,613 and df=3). Duncan's multiple range test indicated all treatment means being significantly different from other treatment means with the same order as with the base inventory. Similarly, for alternate inventory 2, we were able to reject the null hypothesis of equal treatment means at the α=.01 level of significance (F=158 and df=3). However, Duncan's multiple range test indicated that approximations with 250 and 375 states did not produce significantly different improvements in mission requirement satisfaction and the approximation with 500 states produced the worst improvements in mission requirement satisfaction of all treatments. The order of means was 625, 250 and 375, 500.

We conducted the same statistical tests to determine if the treatment means for improved readiness were significantly different. For each inventory, we were able to reject the null hypothesis of equal treatment means. The ANOVA for treatment means for the base inventory had an F statistic of 1,497; for the first alternate inventory 2,172; and for the second alternate inventory 2,801. Duncan's test indicated significant differences between all means. The mean readiness improvement data for each of the three inventories and four approximations are listed below.

Table 7: Experiment One, Readiness Improvement Relative to a Myopic Heuristic

Approximation	Base Inventory	Alt Inventory 1	Alt Inventory 2
250	13.6%	12.6%	13.0%
375	12.8%	12.1%	12.0%
500	9.8%	7.6%	7.8%
625	16.9%	15.1%	16.7%

6.2.2 Discussion

Our results indicate that for both metrics, improved mission requirement satisfaction and readiness, the size of the approximation has a statistically significant effect on those improvements. For both metrics, the largest approximation outperformed the other approximations. However, the results are less clear for the other three approximation sizes. The smallest approximation yielded the second best improvements in readiness for all three inventories. The smallest approximation also outperformed

larger approximations with respect to meeting mission requirement for one of the inventories. Given the computational cost of larger approximations, this ambiguity indicates that it might be computationally prudent to rely on the smallest approximation we tested, 250 states. For the analyses we discuss in the next section, we compared only the approximations using 250 and 625 states.

6.3 Learnt Phase, Experiment 2: Assessing the Robustness of Findings to Alternative Preferences

To examine the robustness of our findings to alternative preferences, we decided to reverse the weights in the objective function: the weight for readiness changed to .7 and the weight for meeting mission requirements changed to .3. We conducted the same statistical tests with the same null hypotheses, namely equal treatment means. For each of the three inventories we tested, we were able to reject the null hypothesis of equal treatment means for both the readiness and mission requirements metrics. The relevant ANOVA test statistics are depicted below.

Table 8: Experiment Two, F-Statistics for Comparing Equality of Means

Inventory	Mission Requirement	Readiness
Base	27	4,492
Alt 1	12.5	5,728
Alt 2	101.5	5,809

Closer inspection of the data revealed an interesting finding. For both metrics, the smaller approximation yielded better outcomes. The mean outcomes for both metrics are listed below.

Table 9: Experiment Two, Performance Improvements Relative to a Myopic Heuristic

Mission requirements			
Approximation	**Base Inventory**	**Alt Inventory 1**	**Alt Inventory 2**
250	3.5%	14.9%	5.1%
625	3.2%	14.4%	4.3%

Readiness			
Approximation	**Base Inventory**	**Alt Inventory 1**	**Alt Inventory 2**
250	19.7%	17.3%	19.0%
625	14.4%	12.2%	13.8%

This result was rather unexpected. However, a few factors may bear on this observation. First, the readiness requirements are deterministic. And in general, readiness requirements are greater than mission requirements. It may be the case that weighting the generally larger, deterministic factor more heavily than stochastic factors greatly simplifies the structure of the state space to value space mapping. Plus, in our methodology, we use a simplifying heuristic to assign units to missions. A method that attempts to optimize both structure and assignment of units might benefit from a larger approximation. However, such a method is beyond the scope of this research.

6.4 Learnt Phase, Experiments 3 and 4: Further testing with more extreme weights

Based on the above observations, we hypothesized that the reversal of performance should hold for other alternate weights. To test the hypothesis, we conducted two additional experiments with more extreme weights on the two factors: weights .9 for readiness, .1 for mission requirements and .1 for readiness, .9 for mission requirements.

Table 10: Experiment Three, F-Statistics Comparing Equality of Means

Inventory	Mission Requirement	Readiness
Base	56	695
Alt 1	190	603
Alt 2	7.9	749

Table 11: Experiment Three, Performance Improvement Relative to a Myopic Heuristic

Mission requirements			
Approximation	**Base Inventory**	**Alt Inventory 1**	**Alt Inventory 2**
250	3.8%	15.7%	5.4%
625	3.3%	13.9%	5.1%

Readiness			
Approximation	**Base Inventory**	**Alt Inventory 1**	**Alt Inventory 2**
250	19.6%	17.2%	19.0%

625	17.4%	15.4%	17.1%

The results, detailed below, were somewhat surprising. As expected, and consistent with our new hypothesis, the experiment that weighted readiness, the deterministic factor, more heavily, had better results with the smaller (250 state) approximation. However, the other experiment, which weighted mission requirements more heavily, also had better results with the 250 state approximation.

Table 12: Experiment Four, F Statistics for testing equality of treatment means

Inventory	Mission Requirement	Readiness
Base	76	864
Alt 1	1,012	1,332
Alt 2	0.03	738

Table 13: Experiment Four, Performance Improvement Relative to a Myopic Heuristic

Mission requirements			
Approximation	**Base Inventory**	**Alt Inventory 1**	**Alt Inventory 2**
250	3.2%	13.9%	5.1%
625	2.8%	9.5%	5.1%

Readiness			
Approximation	**Base Inventory**	**Alt Inventory 1**	**Alt Inventory 2**

250	19.9%	18.0%	19.0%
625	17.4%	15.4%	17.1%

6.5 Discussion

We began the discussion of our results by highlighting the computational burden associated with different approximation sizes. Within the range we tested, we observed run times of up to 26 days to complete 2 million state visits over 100,000 simulations. This run time is significant. Our problem size, 20 unit types, is comparable to many of the problems the Army might face in shaping its force structure. For example, a problem of the sort we modeled in this research could be applied to many of the branch-specific resourcing decisions encountered in a typical Total Army Analysis. And a brigade mix problem would also have in the neighborhood of 20 unit types. So our method should be applicable to many interesting force structure problems.

However, overall problem size is multiplicative in the number of unit types. To apply this method to the entirety of the Army would require significant additional research. We did not investigate different approaches to exploring and exploiting the state space. There may be significant dependencies between either of these and solution quality associated with the DWT VFA. We noted that the computation of the DWT VFA is a function of only approximation size and not the length of the state space vector. But, for larger problems, we may need to explore for longer to ensure sampling of the state space; we cannot know without more research for how many additional simulations we

might need to explore or exploit the state space. Due to this uncertainty, we must still be mindful of computational time.

Our results indicate that for 3 of the 4 experiments we conducted, the smallest approximation yields the best results for both mission requirement satisfaction and future readiness. It may be the case that for approximations larger than we tested we might see an improvement in these two metrics. However, the design of our experiments, fixed effects, does not allow us to infer anything about the performance of approximations whose sizes lie outside the values we tested. Had we randomly selected approximation sizes and learned many more approximations, we might have been able to infer performance for some arbitrary approximation size. But given our earlier discussion about computing time, such a design would be impractical. We might be able to mitigate some of the computational burden through multiple computing threads. Multi-threading might make larger approximations more feasible for testing, though any gains in computational power are linear in the number of processors, so multi-threading is likely not a computational cure-all.

CHAPTER 7: CONCLUSION

[Powell, 2011] tells us that the future of ADP is likely a "collection of fairly specific problem classes with well-defined structures." This point serves as a reminder that we probably should not paint too broadly any findings arising from an ADP solution. Rather, we might be prudent to extend our findings only to problems with similar structures. After all, ADP exists as a field to find computationally tractable, yet demonstrably better solutions to massive problems with unique structures.

In this chapter we discuss the contributions of this research from both a methodological and application perspective as well as areas for further research.

7.1 Application contributions

The primary objective of our research was to demonstrate the application of approximate dynamic programming to the military force mix adaptation problem. We accomplished this objective by developing a simulation-based, sequential decision making model where decision in the model were determined by applying the diffusion wavelet transform as a value function approximation in an approximate dynamic programming formulation.

The United States Army annually revises its force structure. Despite the computational challenges we discussed earlier, the methodology we developed is capable of prescribing alternative decisions within the annual decision cycle. With sufficient lead

time, our methodology can also support numerous sensitivity analyses. All of these are possible without any recourse to parallel processing. Any effort to expand this methodology to include parallel processing would allow for analysis of sensitivity to more parameters and possibly to provide alternative force structure recommendations more rapidly, as decisions are being made.

There is a relative lack of research on military force mix problems. The most frequent approach we identified in the literature applies heuristic search of the decision space (for example, [Mazurek and Wesolkowski, 2009] and [Helms, 2012]). Fully stochastic approaches include some two-stage stochastic programs, [Loerch and Coblentz, 2002] and [Checco, 2015] among them. This dissertation is the first instance we have identified that applies ADP to solve the military force mix problem. This approach extends the problem by identifying the conditions that necessitate changing the fore mix, as expressed through the ADP state variable, and selecting a good set of changes to make to the mix, as expressed through the ADP decision variable. We now refer to the military force mix problem as the military force mix adaptation problem.

One demonstrated strength of the DWT VFA is its scalability to larger problems. We demonstrated our methodology on a sample problem of non-trivial size. With a methodology that can apply to force mix problems of up to 20 unit types, we can use this methodology for a wide array of problems: brigade mix problems; branch resourcing problems within the United States Army's Total Army Analysis; and likely force mix problems for many nations' armed forces. With further development and research, we might be able to develop ADP solutions to the full force mix adaptation problem.

7.2 Methodological contributions

The methodological objective of our research was to investigate the effect of the size of the diffusion wavelet transform value function approximation on solution quality. This research advanced the field in three critical ways: we examined approximation size as relates to solution quality; we established superior performance of the DWT VFA relative to a myopic heuristic; and applied our DWT VFA research to the largest problem yet investigated in the literature.

We examined for the first time the effect of approximation size on solution quality and established superior performance of the DWT VFA to a myopic heuristic. Two previous efforts, [Balakrishna, 2009] and [DeGregory, 2014], studied solution quality with application of the DWT VFA. Both established the superiority of VFA solutions to a myopic heuristic, but neither addressed approximation size as a variable.

In this dissertation, we examined four discrete approximation sizes in four distinct experiments. Each experiment consisted of simulating 1,000 twenty-year scenarios for each of three different starting conditions. In each experiment, VFA identified more valuable decisions than the myopic heuristic for every scenario. The myopic heuristic never outperformed the value function approximation.

As we began this research, we expected to find that, within the range of approximation sizes we tested, solution quality would monotonically increase with approximation size. This proved not to be the case. Our first experiment showed the largest approximation, using 625 states, provided the greatest improvement in performance relative to the myopic heuristic. However, the smallest approximation, using 250 states, did not perform much worse. Given the added computational needed to apply

a DWT VFA with 625 states, we completed the next three experiments comparing only the 250-state and 625-state approximations. In each of these experiments, the 250-state approximation identified better solutions than the 625-state approximation, excepting one starting condition in one experiment in which the performances did not show a statistically significant difference.

7.3 Areas for future research

The first area we would pursue to improve this methodology would be to incorporate the ability to parallel process multiple simulations simultaneously. Doing so would provide immediate additional scalability to our methodology, though this scalability would not be a silver bullet. Parallel processing is linear in the number of processes while problem complexity is exponential in the number of unit types under consideration. However, parallel processing is critical to enabling other areas of potential research.

In addition to parallel processing, we identified three additional areas that warrant further study within the existing methodology: examining the effect of larger approximations on solution quality, which would be aided by parallel processing; examining the effect of imperfect knowledge or varying look-ahead on solution quality; and examining the effect on outcomes of changing the learning phase.

APPENDIX A: PYTHON CODE

Our program used ten modules to execute the functionality described in the main body of this dissertation. In this appendix we provide the code for the nine of the ten modules comprising our program. The tenth module was a general purpose controller which we modified frequently to execute our experiments in manageable chunks. We thus do not include the controller code in this appendix

A.1 Simulation

```
import Params as Par

import SupplyClass as SC

import DemandClass as DC

import StochDemand as SD

import MatchingFunctions as MF

import Initialize as Init

import ADP

import random

import time

import os

import pickle

import thread
```

```
import DW


errorcount=0
ObjectiveList=[]


def Simulation(SimLength, BurnIn,
DecisionInterval,Explore,alpha,SRCData,SupplyList,DemandList,ReadReq,ReadHist):
    #create lists of states and values if in exploration mode
    #these will be used throughout the exploration phase to initialize the state
tracker
    #for use in value function approximation during exploitation phase
    global errorcount


    StateTrack=[]
    ContTrack=[]
    StateValue=[]
    decIntervalSat=[]
    Deactivating=[]
    ValueList=[]
    bigV=[]
```

```
            for T in range(SimLength+1):

                #Update all supply
                SC.UpdateAllSupply(SupplyList)
                #Update all demands

DemandList,Deactivating=MF.UpdateAllDemand(T,DemandList,SupplyList)
                #Update units of supply assigned to deactiving demands
                SupplyList=SC.UpdateSupplyDeact(SupplyList,Deactivating)
                #Assign deployable supply to unfilled demands

DemandList,SupplyList=MF.AssignSupplyDemand(DemandList,SupplyList)

                if T%(int(DecisionInterval))==0:
                    if not T>=BurnIn:

s=MF.getSystemState(DemandList,ReadReq,SupplyList,SRCData,T,Par.lookAhead)
                    if not len(StateTrack)==0 and Explore=='Yes':
                        c=calcContribution(SRCData,decIntervalSat,ReadHist,ReadReq)
                        ContTrack.append(float(c))
                        #clear decision interval satisfaction record after determining
                        #contribution
```

```
                    decIntervalSat[:]=[]
                if T>=BurnIn:

SupplyList,SRCData,maxval=structureDecision(Explore,T,SupplyList,SRCData,s,ReadR
eq,decIntervalSat,Par.gamma)
                    if not Explore=='Yes':
                        ValueList.append(float(maxval))
                    #post-decision state

s=MF.getSystemState(DemandList,ReadReq,SupplyList,SRCData,T,Par.lookAhead)
                    StateTrack.append(s)

            ReadHist=SC.UpdateReadinessHistory(SRCData,SupplyList,ReadHist)

decIntervalSat=DC.trackRecentDemand(SRCData,DemandList,decIntervalSat)

            #after running through a single iteration of the simulation in exploration mode
            #add the state data to the state tracker for use in exploitation phase
            if Explore=='Yes':
                ValueList=CalcTotalCont(ContTrack,Par.gamma)
                StateValue=list(zip(StateTrack,ValueList))
                StateValue=StateValue[:Par.Observations]
```

```
            bigV=SC.calcBigV(StateTrack,Par.DWInitial)
            SC.UpdateErrorList(bigV[:Par.Observations],ValueList[:Par.Observations+1])
            SC.UpdateStateTracker(StateValue,alpha)

            if len(SC.ErrorList)>=10000:
                SC.saveErrorList(errorcount,500)
                errorcount+=1

        def
structureDecision(explore,Time,SupplyList,SRCData,State,ReadReq,decIntervalSat,disco
unt):
            #Find new structure,place result in variable y
            if explore=='Yes':
                #changed function call to RandDec from makeRandomDecision
                y=ADP.RandDec(SRCData,5,State)
                maxval=0
            else:

y,maxval=ADP.makeGoodDecision(State,SRCData,SupplyList,ReadReq,decIntervalSat,
discount)
            #Apply structure changes
            SupplyList,SRCData=SC.applyDecision(y,Time,SupplyList,SRCData)
```

```
    return SupplyList,SRCData,maxval


def calcContribution(SRCData,decIntervalSat,ReadHist,ReadReq):
    #Get demand satisfaction data over previous decision interval
    #Then calculate readiness contribution and roll them both up...
    z=DC.getSatHistory(SRCData,decIntervalSat,'Full')
    b=SC.getReadinessCont(ReadHist,ReadReq,SRCData)
    a=100*MF.CalcCont(b,z)



    return a


def RecurseValues(elist,discount):
    if not elist:
        return 0
    else:
        return elist[0]+discount*RecurseValues(elist[1:],discount)


def CalcTotalCont(elist,discount):
    ContList=[]
    for i in range(len(elist)):
        ContList.append(RecurseValues(elist[i:],discount))
```

```
        return ContList

    def writedatatofile(path):
        myfile=open(path+'\\perf.txt','w')
        for item in ObjectiveList:
            myfile.write(str(item)+'\n')
        myfile.close()

    def storeObjTracker(path):
        name=path+'//Obj.pkl'
        F=open(name,'wb')
        pickle.dump(ObjectiveList,F)
        F.close()

    def LearntExperiment(Type, SimLength, BurnIn,
DecisionInterval,SupplyList,SRCData,DemandList,ReadHist,ReadReq):

        ContTrack=[]
        decIntervalSat=[]
        Deactivating=[]
        ValueList=[]
```

```
        for T in range(SimLength+1):

            #Update all supply
            SC.UpdateAllSupply(SupplyList)
            #Update all demands

DemandList,Deactivating=MF.UpdateAllDemand(T,DemandList,SupplyList)
            #Update units of supply assigned to deactiving demands
            SupplyList=SC.UpdateSupplyDeact(SupplyList,Deactivating)
            #Assign deployable supply to unfilled demands

DemandList,SupplyList=MF.AssignSupplyDemand(DemandList,SupplyList)

            if T%(int(DecisionInterval))==0 and T>=BurnIn:
               if not T>=BurnIn+1:

s=MF.getSystemState(DemandList,ReadReq,SupplyList,SRCData,T,Par.lookAhead)
               if T>=BurnIn+1:
                  c=calcContribution(SRCData,decIntervalSat,ReadHist,ReadReq)
                  ContTrack.append(float(c))
```

```
            if T>=BurnIn:
                #Force structure decision

SupplyList,SRCData=LearntDecision(Type,T,SupplyList,SRCData,s,ReadReq,decIntervalSat)
                #post-decision state

s=MF.getSystemState(DemandList,ReadReq,SupplyList,SRCData,T,Par.lookAhead)
            #clear decision interval satisfaction record after determining contribution and making decision
            decIntervalSat[:]=[]

          ReadHist=SC.UpdateReadinessHistory(SRCData,SupplyList,ReadHist)

decIntervalSat=DC.trackRecentDemand(SRCData,DemandList,decIntervalSat)

        ValueList=CalcTotalCont(ContTrack,Par.gamma)
        Objective=ValueList[0]

        return Objective

    def LearntDecision(Type,T,SupplyList,SRCData,s,ReadReq,decIntervalSat):
```

```
            if Type=='VFA':

y,maxval=ADP.makeGoodDecision(s,SRCData,SupplyList,ReadReq,decIntervalSat,Par.
gamma)
            elif Type=='Myopic':
                #Need to make a myopic decision function
                y=ADP.MyopicDecision(s,SRCData,SupplyList,ReadReq,decIntervalSat)
            SupplyList,SRCData=SC.applyDecision(y,T,SupplyList,SRCData)
            return SupplyList,SRCData

        def ValidationExperiment(Type,path,count,SimLength,BurnIn,DecisionInterval):
            #initialize input data for a simulation
            fname=path+'Validate//dem_'+str(count)+'.pkl'

SupplyList,SRCData,DemandList,ReadHist,ReadReqt=Init.InitializeValidation(path,fna
me)

            if Type=='VFA':
                g='StateSpace_Final.pkl'
                Init.InitializeApprox(path,g,Par.LenStateTrack)

            #run a simulation with the relevant inputs
```

```
            obj=LearntExperiment(Type, SimLength, BurnIn,
DecisionInterval,SupplyList,SRCData,DemandList,ReadHist,ReadReqt)
            #append the simulation data to the relevant data structure
            ObjectiveList.append(obj)

        def InitIterations(numiters):

            InitialComplete=False
            iteration=1
            alpha=.95

            while iteration<=numiters and not InitialComplete==True:
                #Clear simulation inputs from previous iteration
                SupplyList=[]
                SRCData={}
                DemandList=[]
                ReadReq={}
                ReadHist={}
                #initialize supply structures for this iteration
                SupplyList,SRCData=SC.InitSupply('.//SupplyRecords.txt')
                #initialize demand structures for this iteration
                DemandList=SD.processDemandDirectory('.//DemFiles')
```

```
            #initialize readiness structures for this iteration
            ReadReq,ReadHist=SC.InitReadReqt('.//Readiness.txt')

Simulation(Par.SimLength,Par.BurnInPd,Par.DecInterval,'Yes',alpha,SRCData,SupplyLis
t,DemandList,ReadReq,ReadHist)
            if iteration%500==0:
               SC.TrimTracker()
               print 'Tracker',len(SC.StateTracker)
               SC.storeStateTracker('.//','19Init')
               print iteration, time.clock(),alpha
            iteration+=1
            alpha=alpha-.000001
         SC.StateTracker.sort(key=lambda k: (k['Value']),reverse=True)
         SC.storeStateTracker('.//','19Init')

      def TransitionIterations(numiters,numstates):

         iteration=1
         alpha=.7
         p=.8
         p0=.8
```

```
        Init.InitializeApprox('.//','StateSpace_73Init.pkl',numstates)


        while iteration<=numiters:
            SupplyList=[]
            SRCData={}
            DemandList=[]
            ReadReq={}
            ReadHist={}
            explorechance=random.random()
            #initialize supply structures for this iteration
            SupplyList,SRCData=SC.InitSupply('.//SupplyRecords.txt')
            #initialize demand structures for this iteration
            DemandList=SD.processDemandDirectory('.//DemFiles')
            #initialize readiness structures for this iteration
            ReadReq,ReadHist=SC.InitReadReqt('.//Readiness.txt')
            if explorechance<=p:

Simulation(Par.SimLength,Par.BurnInPd,Par.DecInterval,'Yes',alpha,SRCData,SupplyLis
t,DemandList,ReadReq,ReadHist)
            else:
```

```
Simulation(Par.SimLength,Par.BurnInPd,Par.DecInterval,'No',alpha,SRCData,SupplyList
,DemandList,ReadReq,ReadHist)
                if iteration%500==0:
                    print iteration, time.clock()
                iteration+=1
                alpha=alpha-.15/float(numiters)
                p=p-p0/float(numiters)

            def ExploitIterations(numiters,numstates):

                iteration=1
                alpha=.55

                Init.InitializeApprox('.//','StateSpace_TransAW19.pkl',numstates)

                while iteration<=numiters:
                    SupplyList=[]
                    SRCData={}
                    DemandList=[]
                    ReadReq={}
                    ReadHist={}
```

```
        #initialize supply structures for this iteration
        SupplyList,SRCData=SC.InitSupply('.//SupplyRecords.txt')
        #initialize demand structures for this iteration
        DemandList=SD.processDemandDirectory('.//DemFiles')
        #initialize readiness structures for this iteration
        ReadReq,ReadHist=SC.InitReadReqt('.//Readiness.txt')

Simulation(Par.SimLength,Par.BurnInPd,Par.DecInterval,'No',alpha,SRCData,SupplyList
,DemandList,ReadReq,ReadHist)
        if iteration%500==0:
            print iteration, time.clock()
            SC.storeStateTracker('.//','Final')
        iteration+=1
        alpha=alpha-.45/float(250000)
```

A.2 Matching Functions

```
    import SupplyClass as SC
    import DemandClass as DC
    import Params
    import time

    RecentPerf=[]
```

```
def UpdateDemandFill(DemandList,SupplyList):
    #make list of active demands

    ActiveDemandList=[obj for obj in DemandList if obj.status=='Active']

    for obj in ActiveDemandList:
        #make list of units assigned to current demand in the loop
        Units=[x for x in SupplyList if x.location==obj.name]
        #Number of units assigned
        count=len(Units)
        if count>=obj.qty+1:
            print 'More assigned than needed'
        obj.UpdateAssigned(count)
        #increment the total number of unit months satisfied
        obj.UpdateTotSat(obj.assigned)

    return DemandList


def UpdateAllDemand(Time,DemandList,SupplyList):
    #Deactivate Demands with finish = "Now"-1
```

```
        #Activate Demand with start = "Now"
        DemandList,Deactivating=DC.UpdateDemandStatus(Time,DemandList)
        #Update number of units assigned to each demand
        DemandList=UpdateDemandFill(DemandList,SupplyList)

        return DemandList,Deactivating

    def AssignSupplyDemand(DemandList,SupplyList):
        #Find unfilled, active demands
        DList=DC.FindUnfilledDemand(DemandList)

        #Find deployable supply
        SList=SC.FindDeployableSupply(SupplyList)

        #make list of SRCs with at least one unfilled demand
        q=DC.getUnfilledSRC(DList)
        #iterate through the list of SRCs
        for entry in q:
            #Make a list of deployable supply with the SRC, ordered by CycleTime,
descending
            Supp=[x for x in SList if x.SRC==entry]
            #Make a list of  unfilled demands with the SRC, ordered by priority
```

```
            Dem=[y for y in DList if y.SRC==entry]
            #for each entry in the Dem list
            for demand in Dem:
               #if supply list is not empty
               if Supp:
                  #determine number of units to assign
                  n=min(len(Supp),demand.qty-demand.assigned)
                  #assign units to demand
                  for q in range(n):
                     Supp[q].AssignSupply(demand.name)
                 #delete the assigned units of supply from the supply list
                  Supp[0:n]=[]

         return DemandList,SupplyList

      def CalcCont(Ready,Fill):

Cont=Ready*Params.MetricWeight['Ready']+Fill*Params.MetricWeight['Now']
         return Cont

      def getSystemState(DemandList,readReq,SupplyList,SRCData,t,window):
         #system state: combination of window-length demand forecast, readiness
```

```
        #requirements, and current readiness
        #1) Calculate max and current demand by SRC
        #2) Calculate max deployable and currently deployed supply
        #3) Calculate forecast surplus,shortage=readReq+maxDem-CurrDem-
(MaxDep-CurrDep)

        f=[]
        SRClist=[obj for obj in SRCData.keys()]

        maxDem=DC.getDemandLookAhead(t,window,DemandList,SRCData)
        currDem=DC.getDemandLookAhead(t,0,DemandList,SRCData)

        maxDep=SC.getMaxDeployable(SRCData)
        currDep=SC.getCurrDep(SupplyList,SRCData)

        for entry in SRClist:
            p=round(readReq[entry]+maxDem[entry]-currDem[entry]-(maxDep[entry]-
currDep[entry]),0)
            f.append((entry,p))
        f.sort()

        return f
```

```
def TopBottom(length,Forecast):
    Recent=sorted(Forecast,key=lambda x:x[1],reverse=True)
    t=Recent[:length]
    b=Recent[len(Forecast)-length:]

    top=[(item[0],1) for item in t]
    bottom=[(item[0],-1) for item in b]



    return top,bottom
```

A.3 Supply Class

```
import Params as Par
import DemandClass as DC
import ADP
import numpy as NP
import time
import pickle
import DW

StateTracker=[]
ErrorList=[]
```

```
densitycheck=[]
mu=0
sigma=0
mulist=[]
sigmalist=[]

maxNewStruct=3 #constrains the total new personnel in any year
EchelonMax={'Bde':1, 'Bn':2, 'Co':3} # lists the max any single SRC at this echelon can grow
TotalUnitGrowth=10
MaxPAXDelta=1200 #allows for some deviation from 0 end-strength growth

class Supply:
    def __init__(self, SRC, name, cycletime, location='Home'):
        self.SRC=SRC
        self.name=name
        self.cycletime=cycletime % Par.policy['CycleMax']
        self.location=location
        if self.cycletime<=Par.policy['NotDep']:
            self.status='NotDeployable'
        elif self.cycletime> Par.policy['NotDep'] and self.cycletime <= Par.policy['CycleMax']:
```

```
        self.status='Deployable'
    def UpdateStatus(self):
        if self.location == 'Home':
            if self.status == 'Deployable' and self.cycletime>Par.policy['CycleMax']:
                self.cycletime=0
                self.status='NotDeployable'
            elif self.status=='NotDeployable' and self.cycletime>Par.policy['NotDep']:
                self.status='Deployable'
        elif self.status=='Deployed' and self.cycletime>Par.policy['Deploy']:
            self.cycletime =0
            self.status='NotDeployable'
            self.location='Home'
    def UpdateDeact(self,Deactivating):
        if self.location in Deactivating:
            self.cycletime =0
            self.status='NotDeployable'
            self.location='Home'
    def AdvanceCycle(self):
        self.cycletime=int(self.cycletime+1)
    def AssignSupply(self,name):
        self.location=name
        self.status='Deployed'
```

```
            self.cycletime=0
        def DeactivateUnit(self):
            self.status='Inactive'
            self.location='None'

    def InitSupply(filename):
        #local list and dictionary for use within a single simulation
        SupplyList=[]
        SRCData={}
        SupIn = [line.split() for line in open(filename)]

        for i in range(len(SupIn)):
            #make dictionary of SRCs: inventory and associated unit sizes, echelon from input file
            SRCData[SupIn[i][0]]=[int(SupIn[i][1]),int(SupIn[i][2]),SupIn[i][3]]
            #create supply objects and populate a list with these objects
            for j in range(int(SupIn[i-1][1])):
                SupplyList.append(Supply(str(SupIn[i-1][0]),str(SupIn[i-1][0])+'_'+str(j+1),j*Par.policy['CycleMax']/int(SupIn[i-1][1]),))

        return SupplyList, SRCData
```

```
def UpdateAllSupply(SupplyList):
    for Obj in SupplyList:
        Obj.AdvanceCycle()
        Obj.UpdateStatus()
def UpdateSupplyDeact(SupplyList,Deactivating):
    for obj in SupplyList:
        obj.UpdateDeact(Deactivating)
    return SupplyList

def FindDeployableSupply(SupplyList):
    SupplyDep=[]

    SupplyDep=[x for x in SupplyList if x.status=='Deployable']
    SupplyDep=sorted(SupplyDep, key=lambda k: (k.cycletime))

    return SupplyDep

def createNewSupply(SRC,number,time,SupplyList):

    for i in range(int(number)):
        #create new units and append to SupplyList
        SupplyList.append(Supply(SRC,SRC+'_'+str(number)+'_'+str(time),0,))
```

```
    return SupplyList


def InactivateUnits(SRC,number,SupplyList):
    #make sorted list of names by cycletime for SRC
    dList=[x for x in SupplyList if x.SRC==SRC]
    dList=sorted(dList,key=lambda k: (k.cycletime))
    dList=dList[0:abs(int(number))-1]


    for obj in dList:
        #deactivate unit, then remove from SupplyList
        obj.DeactivateUnit()
        SupplyList.remove(obj)
    return SupplyList


def ApplyStructChanges(SRC,quantity,time,SupplyList):
    if quantity>=1:
        SupplyList=createNewSupply(SRC,quantity,time,SupplyList)
    elif quantity<=-1:
        SupplyList=InactivateUnits(SRC,quantity,SupplyList)
    else:
        return
    return SupplyList
```

```
def modSRCInventory(SRC,delta,SRCData):
    SRCData[SRC][0]=SRCData[SRC][0]+delta
    return SRCData

def InitReadReqt(filename):
    ReadReq={}
    ReadHist={}

    Input = [line.split() for line in open(filename)]

    for i in range(len(Input)):
        ReadReq[Input[i][0]]=int(Input[i][1])
        ReadHist[Input[i][0]]=[]

    return ReadReq,ReadHist

def getCurrentReadiness(SRCData,SupplyList):
    readCurr=[]

    #count number of units of each SRC ready to deploy within 'DepWindow'
months
```

```
        for SRC in SRCData:
            List=[x for x in SupplyList if x.SRC==SRC and x.location=='Home' and
x.cycletime>=(Par.policy['Deploy']-Par.DepWindow)]
            c=len(List)
            T=(SRC,c)
            readCurr.append(T)

        readCurr.sort()

        return readCurr

    def UpdateReadinessHistory(SRCData,SupplyList,ReadHist):
        #Make variable with current readiness stats for update

        c=getCurrentReadiness(SRCData,SupplyList)

        for entry in c:
            if len(ReadHist[entry[0]])<=(Par.DecInterval-1):
                #Grow the list for each SRC until it is DecInterval long
                ReadHist[entry[0]].append(entry[1])
            else:
```

```
                #once the list is DecInterval long, pop the oldest record, append the new
one
                ReadHist[entry[0]].pop(0)
                ReadHist[entry[0]].append(entry[1])
            return ReadHist

        def UpdateTotReady(olddata,newdata):
            #get list of keys (SRCs) from newdata
            SRClist=[obj for obj in olddata.keys()]

            #for each key, update value list with list items from newdata
            for item in SRClist:
                for entry in newdata[item]:
                    olddata[item].append(entry)

            return olddata

        def initTotReady(SRCData):
            datadict={}
            SRClist=[obj for obj in SRCData.keys()]

            for item in SRClist:
```

```
        datadict[item]=[]

    return datadict

def getReadinessCont(ReadHist,ReadReq,SRCData):
    ReadCont=[]
    Req=0

    #calculate the total personnel ready at each sample
    for i in range(Par.DecInterval):
        c=0
        for SRC in ReadHist:
            #take the minimum of required and ready, then weight by unit size
            c+=min(ReadHist[SRC][i],ReadReq[SRC])*SRCData[SRC][1]
        #once the total personnel is calculated,append to the list
        ReadCont.append(c)

    #calculate the total personnel required
    for SRC in ReadReq:
        Req+=ReadReq[SRC]*SRCData[SRC][1]

    d=min(ReadCont)
```

```
        #return ratio of ready to required
        return d/float(Req)

    def getSystemState(SRCData,SupplyList):
        s=getCurrentReadiness(SRCData,SupplyList)

        return s

    def UpdateStateTracker(StateStruct,alpha):
         l=len(StateTracker)
        tl=[]
        #Boolean to determine if state tracker has been updated with new data, only applies after initializing the DW
        update=False

        #make list of states in state tracker for searching
        st=[item['State'] for item in StateTracker]

        for item in StateStruct:
            #either the state is in the tracker, so retrieve its value, or populate a list of states to estimate
```

```
    if item[0] in st:


        #find where in the list the state resides
        dex=st.index(item[0])
        #retrieve the state's value from the state tracker
        lastValue=float(StateTracker[dex]['Value'])
        #use the Bellman update equation to determine the state's new value
        nextValue=float((1-alpha)*(lastValue)+alpha*item[1])
        #update the state tracker with the state's new value
        StateTracker[dex]['Value']=nextValue
        StateTracker[dex]['Visits']+=1
        update=True
    else: #make a list of states to estimate value
        tl=[]
        p={}
        p['State']=item[0]
        p['Value']=item[1] #little v hat
        p['Visits']=1
        tl.append(p)
if not(len(tl))==0:
    if Par.DWInitial=='Yes':
        Estimates=ADP.estStateValue(tl,Par.LenStateTrack)
```

```
                estStates=[item['State'] for item in Estimates]
                for item in tl:
                    p={}
                    p['State']=item['State']
                    dex=estStates.index(item['State'])
                    lastValue=float(Estimates[dex]['Value'])
                    p['Value']=float((1-alpha)*(lastValue)+alpha*item['Value'])
                    p['Visits']=1
                   #if the value is greater than the last entry in the StateTracker, add the
new state to the list
                    StateTracker.append(p)
                StateTracker.sort(key=lambda k: (k['Value']),reverse=True)
                StateTracker[Par.LenStateTrack:]=[]
                update=True
            else:
                for item in tl:
                    p={}
                    p['State']=item['State']
                    p['Value']=float(alpha*item['Value'])
                    p['Visits']=1
                    StateTracker.append(p)
```

```
        def applyDecision(decVector,Time,SupplyList,SRCData):


            for i in range(len(decVector)):
                if not decVector[i][1]==0:


SupplyList=ApplyStructChanges(decVector[i][0],decVector[i][1],Time,SupplyList)
                    SRCData=modSRCInventory(decVector[i][0],decVector[i][1],SRCData)
                else:
                    next
            return SupplyList,SRCData
        def convergenceCheck(stability):
            #determine number of iterations that are within bounds
            numiters=0
            YesList=[]
            YesList=[ErrorList[i] for i in range(len(ErrorList)-stability,len(ErrorList)) if
(ErrorList[i]>=mu-sigma and ErrorList[i]<=mu+sigma)]
            numiters=len(YesList)
            if numiters==stability:
                convergence='Yes'
            else:
                convergence='No'
```

```
    return convergence


def CalcReadReq(SRCData,ReadReq):
    Req=0
    for SRC in ReadReq:
        Req+=ReadReq[SRC]*SRCData[SRC][1]
    return Req


def TrackReadiness(SRCData,SupplyList,ReadReq):
    ready=0
    req=CalcReadReq()

    c=getCurrentReadiness(SRCData,SupplyList)

    for entry in c:
        ready+=SRCData[entry[0]][1]*min(entry[1],ReadReq[entry[0]])
    return ready/float(req)


def storeStateTracker(path,size):
    if not type(size)==str:
        name=path+'//StateSpace_'+str(size)+'.pkl'
    else:
```

```
        name=path+'//StateSpace_'+size+'.pkl'
    F=open(name,'wb')
    pickle.dump(StateTracker,F)
    F.close()


def estReadinessCont(Candidates,ReadReq,SRCData):
    ReadCont=[]
    Req=0

    #calculate the total personnel required
    for SRC in ReadReq:
        Req+=ReadReq[SRC]*SRCData[SRC][1]

    #calculate the total personnel ready at each sample
    for entry in Candidates:
        c=0
        r=0
        for i in range(len(entry['State'])):
            #take the minimum of required and ready, then weight by unit size
```

```
c+=min(entry['State'][i][1],ReadReq[entry['State'][i][0]])*SRCData[entry['State'][i][0]][1
]
                #once the total personnel is calculated,determine proportion for candidate
                r=(c/float(Req))*Par.MetricWeight['Ready']
                ReadCont.append([entry,r])

            return ReadCont


        def storeErrorTracker(path):
            name=path+'//ErrorTrack.pkl'
            F=open(name,'wb')
            pickle.dump(ErrorList,F)
            F.close()

        def UpdateVFA():
            TempState=[]
            Tracker=[]

            TempState.append(StateTracker[Par.LenStateTrack-1]['State'])
```

```
    #sort the state tracker and trim it down to size
    StateTracker.sort(key=lambda k: (k['Value']),reverse=True)
    for i in range(Par.LenStateTrack):
        Tracker.append(StateTracker[i])

    StateTracker[Par.LenStateTrack:]=[]

    #if the last entry in the newly trimmed tracker is different than before,
    #new items have been added to the list, so update the DW coefficients
    if not StateTracker[Par.LenStateTrack-1]['State']==TempState[0]:
        ADP.initDWdata(Tracker,.5)

def calcMSE(Square):
    Errors=[]
    l=len(ErrorList)

    for i in range(l):
        Errors.append(ErrorList[i]['Square'])

    #calculate MSE
    squaresum=sum(Errors)+Square
    MSE=squaresum/float(l+1)
```

```
    return MSE


def sortedTracker(keep,freq,number):
    Tracker=[]
    StateTracker.sort(key=lambda k: (k['Value']),reverse=True)
    for i in range(keep):
        Tracker.append(StateTracker[i])


    return Tracker


def checkVisitSaturation(keep,percent,number):
    #determine number of frequently visited states
    if Par.freqVisSaturation=='No':
        numHighVis=calcVisSet(number)
        #compare to percentFreqVis=.5
        if numHighVis>=keep*percent:
            Par.freqVisSaturation='Yes'
            StateTracker[keep:]=[]


def calcVisSet(number):
    templist=[i for i in StateTracker if i['Visits']>=number]
```

```
        lenlist=len(templist)
        return lenlist

    def getMaxDeployable(SRCData):
        MaxDep={}

        #theoretical max percentage of units depoyable
        percentage=(Par.policy['CycleMax']-
Par.policy['NotDep'])/float(Par.policy['CycleMax'])
        for item in SRCData:
            Inventory=item[0]
            Dep=int(int(Inventory)*percentage)
            MaxDep[item]=Dep

        return MaxDep

    def getCurrDep(SupplyList,SRCData):
        CurrDep={}
        SRClist=[obj for obj in SRCData.keys()]

        for item in SRClist:
```

```
            Dep=[entry for entry in SupplyList if entry.SRC==item and
entry.status=='Deployed']
            if Dep:
                num=len(Dep)
            else:
                num=0
            CurrDep[item]=num

        return CurrDep

    def TrimTracker():

        for item in StateTracker:
            if item['Visits']==1:
                StateTracker.remove(item)
        print 'StateTracker',len(StateTracker)

    def checkInitConditions(length,freqvis,density):
        #check to see if StateTracker has enough high valued, frequently visited states
        num=0
        StateTracker.sort(key=lambda k: (k['Value']),reverse=True)
        for i in range(length):
```

```
        if StateTracker[i]['Visits']>=freqvis:
            num+=1
    densitycheck.append(num/float(length))
    if num/float(length)>=density:
        print 'StateSpace conditions met!'
        return True
    else:
        return False



def calcBigV(states,DWInitial):
    bigV=[]
    appList=[]
    StateList=[item['State'] for item in StateTracker]
    for item in states:
        d={}
        d['State']=item
        d['Value']=0
        if item in StateList:
            #if the state is in the tracker, retrieve its value
            dex=StateList.index(item)
            d['Value']=StateTracker[dex]['Value']
```

```
    elif DWInitial=='Yes':
        #if the DW has been initialized, append to list for value approximation
        appList.append(d)
    else:
        #if estimating the value is not possible, set to zero
        d['Value']=0
    bigV.append(d)

#approximate value of states in appList, if it exists
if not len(appList)==0:
    Est=ADP.estStateValue(appList,Par.LenStateTrack)

    #create list of states in bigV tracker
    bigVStates=[p['State'] for p in bigV]

    for entry in Est:
        #find the location in bigV list of state, update value key
        dex=bigVStates.index(entry['State'])
        bigV[dex]['Value']=entry['Value']
bigV.sort(key=lambda k:k['Value'],reverse=True)
return bigV
```

```
def UpdateErrorList(bigV,littleV):



    for i in range(len(bigV)):


        diff=bigV[i]['Value']-littleV[i+1]
        ErrorList.append(diff**2)


def saveErrorList(Type,errorcount,bandcalc):
    global ErrorList
    #Save ErrorList to directory
    if not type(errorcount)==str:
        name='.//ErrorDumps//Error_'+str(Type)+str(errorcount)+'.pkl'
    else:
        name='.//ErrorDumps//Error_'+str(Type)+errorcount+'.pkl'
    F=open(name,'wb')
    pickle.dump(ErrorList,F)
    F.close()
    l=len(ErrorList)
    #calculate the mean and error for error band calculation
    mu=NP.mean(ErrorList[(l-bandcalc):])
```

```
    mulist.append(mu)
    sigma=NP.std(ErrorList[(l-bandcalc):])
    sigmalist.append(sigma)
    #clearErrorList
    ErrorList[:]=[]

def saveMuSigma(mu,sigma):
    muname='.//ErrorDumps//Mu.pkl'
    sigmaname='.//ErrorDumps//Sigma.pkl'
    F=open(muname,'wb')
    pickle.dump(mu,F)
    F.close()

    G=open(sigmaname,'wb')
    pickle.dump(sigma,G)
    G.close()

def writeMuSigma():
    muname='.//ErrorDumps//Mu.pkl'
    sigmaname='.//ErrorDumps//Sigma.pkl'

    F=open(muname,'rb')
```

```
    m=pickle.load(F)
    F.close()

    G=open(sigmaname,'rb')
    s=pickle.load(G)
    G.close()

    mufile=open('.//mu.txt','w')
    for item in m:
        mufile.write(str(item)+'\n')
    mufile.close()

    sigmafile=open('.//sigma.txt','w')
    for item in s:
        sigmafile.write(str(item)+'\n')
    sigmafile.close()

def calcRealReady(ReadyData,SRCData,ReadReq):
    Ready={}
    readsum=0
    reqsum=0
    SRCList=[obj for obj in SRCData.keys()]
```

```
        #calculate average readiness by SRC
        for item in ReadyData:
            Ready[item]=sum(ReadyData[item])/len(ReadyData[item])
        #calculate weighted Average readiness
        for entry in SRCList:
            readsum+=SRCData[entry][1]*Ready[entry]
        #calculate weighted Readiness required
        for entry in SRCList:
            reqsum+=SRCData[entry][1]*ReadReq[entry]
        #calculate weighted average readiness percent
        r=readsum/float(reqsum)


        return r
```

A.4 Demand Class

```
    import operator
    import SupplyClass as SC
    import Params
    class Demand:
        def __init__(self, scenario, SRC, name, start, duration, qty, priority,
status='Inactive'):
            self.scenario=scenario
            self.SRC=SRC
```

```
            self.start=int(start)
            self.finish=int(start)+int(duration)-1
            self.qty=int(qty)
            self.name=name
            self.assigned=0
            self.priority=int(priority)
            self.status=status
            self.TotReqd=int(qty)*int(duration)
            self.TotSat=0
        def UpdateAssigned(self,count):
            self.assigned=count
        def UpdateTotSat(self,count):
            self.TotSat+=count


    def FindUnfilledDemand(DemandList):
        DemandUnfilled=[]

        DemandUnfilled=[x for x in DemandList if x.status=='Active' and x.assigned
<= (x.qty-1)]
        DemandUnfilled=sorted(DemandUnfilled, key=lambda k: (k.SRC, k.priority))
```

```
    return DemandUnfilled


def getUnfilledSRC(List):


    UnfilledSRC=[]
    seen=set()
    for obj in List:
        if obj.SRC not in seen:
            UnfilledSRC.append(obj.SRC)
            seen.add(obj.SRC)


    return UnfilledSRC


def UpdateDemandStatus(Time,DemandList):
    #list of deactivating demands
    Deactivating=[]


    for x in DemandList:
        if x.start==Time:
            x.status='Active'
        elif x.finish==Time-1:
            x.status='Inactive'
```

```
            #add name of demand to both current deactivating and archive thereof
            Deactivating.append(x.name)
    return DemandList,Deactivating


def clearHistDeact():
    HistDeact[:]=[]


def getSatHistory(SRCData,decIntervalSat,Type):
    TotalSat=0
    TotalReqd=0

    if Type=='Full':
        #if type is full, calculate over full decision interval
        for item in decIntervalSat:
            for obj in item.keys():
                #TotalSat is weighted by SRC size
                TotalSat+=int(item[obj][0])*int(SRCData[obj][1])
                TotalReqd+=int(item[obj][1])*int(SRCData[obj][1])
    elif Type=='Est':
        #if type is estimate, calculate only for the latest entry into decIntervalSat
        l=len(decIntervalSat)
        item=decIntervalSat[l-1]
```

```
        for obj in item.keys():
            #TotalSat is weighted by SRC size
            TotalSat+=int(item[obj][0])*int(SRCData[obj][1])
            TotalReqd+=int(item[obj][1])*int(SRCData[obj][1])

    if not TotalReqd==0:
        z=TotalSat/float(TotalReqd)
    else:
        z=1
    #Return size weighted percentage of demand satisfaction

    return z

def TrackFill():
    Sat=0
    Req=0

    for obj in DemandList:
        Sat+=obj.TotSat*SC.SRCData[obj.SRC][1]
        if obj.finish>=(Params.SimLength+1) and obj.start<=Params.SimLength:
            Req+=(Params.SimLength-obj.start+1)*obj.qty*SC.SRCData[obj.SRC][1]
        else:
```

```
                Req+=obj.TotReqd*SC.SRCData[obj.SRC][1]
        return Sat/float(Req)


    def initRecDem(SRCData):
        recDem={}

        SRClist=[obj for obj in SRCData.keys()]

        for entry in SRClist:
            recDem[entry]=()

        return recDem

    def trackRecentDemand(SRCData,DemandList,decIntervalSat):
        recDem={}
        SRClist=[obj for obj in SRCData.keys()]

        for entry in SRClist:
            sat=[Demand.assigned for Demand in DemandList if Demand.SRC==entry
and Demand.status=='Active']
```

```
            req=[Demand.qty for Demand in DemandList if Demand.SRC==entry and
Demand.status=='Active']
            recDem[entry]=()
            if req:
                a=reduce(lambda x,y:x+y,sat)
                b=reduce(lambda x,y:x+y,req)
                recDem[entry]=(a,b)
            else:
                recDem[entry]=(0,0)
        #append by SRC demand satisfaction data to the decision interval history for
use in calculating contribution
        decIntervalSat.append(recDem)

        return decIntervalSat

    def updateFillHistory(SRCData,decIntervalSat,fillHistory):
        d={}
        SRClist=[obj for obj in SRCData.keys()]

        for entry in SRClist:
            d[entry]=(0,0)
```

```
            for item in decIntervalSat:
                for obj in item.keys():
                    d[obj]=tuple(map(lambda x,y:x+y,d[obj],item[obj]))

            fillHistory.append(d)

            return fillHistory

        def getDemandLookAhead(T,Window,DemandList,SRCData):
            DemForecast={}
            SRClist=[obj for obj in SRCData.keys()]

            for entry in SRClist:
                maxi=0
                for i in range(T,T+Window):
                    Dem=[Demand.qty for Demand in DemandList if Demand.SRC==entry
and Demand.start<=i and Demand.finish>=i]
                    if Dem:
                        a=reduce(lambda x,y:x+y,Dem)
                    else:
                        a=0
                    if a>=maxi:
```

```
                maxi=a
            DemForecast[entry]=maxi


        return DemForecast


    def calcSat(satDict,reqDict,SRCData):
        satWeight=0
        reqWeight=0

        for key in SRCData.keys():
            satWeight+=satDict[key]*SRCData[key][1]
            reqWeight+=reqDict[key]*SRCData[key][1]

        s=satWeight/float(reqWeight)

        return s
```

A.5 ADP

```
    import random
    import SupplyClass as SC
    import DemandClass as DC
    import numpy as np
    import DW
```

```
import Simulation as Sim
import operator
import Params as Par
import math
import time
import itertools as it
import MatchingFunctions as MF
from copy import deepcopy

def makeRandomDecision(SRCData):
    declist=[]
    slist=[SRC for SRC,data in SRCData.items() if data[0]>=1]
    #determine number of unit types to ad inventory
    num=random.randint(0,SC.maxNewStruct)
    #find random set of num unit types to add inventory
    add=random_combination(slist,num)
    for entry in add:
        declist.append((entry,1))
    #remove this list from total SRC list
    slist=list(set(slist)-set(add))
    #create random list of num unit types to reduce inventory
    red=random_combination(slist,num)
```

```
    for entry in red:
        declist.append((entry,-1))

    return declist

def RandDec(SRCData,length,state):
    #limit decision space to best and worst forecast SRCs
    t,b=MF.TopBottom(length,state)
    #make list of SRCs that can shrink
    slist=[SRC for SRC,data in SRCData.items() if data[0]>=1]
    #pare the list of reduction candidates to only those with non-zero inventory
    reduction=[item for item in b if item[0] in slist]
    #make the list of candidate solutions
    l=min(len(t),len(reduction))
    DecSpace=makeGoodDecSpace(t,reduction,3)
    #create random num to choose decision from DecSpace
    num=random.randint(0,len(DecSpace)-1)

    sol=DecSpace[num]

    return sol
```

```
def
makeGoodDecision(state,SRCData,SupplyList,ReadReq,decIntervalSat,discount):

    Candidates=[]
    estCont=[]

    #generate the list of candidate solutions
    Candidates=generateGoodDec(5,3,state)

    #Estimate the post decision state for each candidate solution
    CandStates=estPostDecState(Candidates,state)
    #estimate the value of the next state for each candidate decision
    est=estStateValue(CandStates,Par.LenStateTrack)
    estContList=SC.estReadinessCont(CandStates,ReadReq,SRCData)
    fill=Par.MetricWeight['Now']*DC.getSatHistory(SRCData,decIntervalSat,'Est')
    for item in estContList:
       item[1]=100*(Par.MetricWeight['Ready']*item[1]+fill)
       v=(item[0],item[1])
       estCont.append(v)

    #pick the candidate solution with the greatest total value
    y,maxval=findMaxEstimate(est,estCont,discount)
```

```
    #return this best candidate solution
    choice=Candidates[y]

    return choice,maxval

def MyopicDecision(state,SRCData,SupplyList,ReadReq,decIntervalSat):
    Candidates=[]
    estCont=[]

    #generate the list of candidate solutions
    Candidates=generateGoodDec(5,3,state)
    #Estimate the post decision state for each candidate solution
    CandStates=estPostDecState(Candidates,state)
    #Estimate contribution for candidate states
    estContList=SC.estReadinessCont(CandStates,ReadReq,SRCData)
    fill=Par.MetricWeight['Now']*DC.getSatHistory(SRCData,decIntervalSat,'Est')
    for item in estContList:
        item[1]=100*(Par.MetricWeight['Ready']*item[1]+fill)
        v=(item[0],item[1])
        estCont.append(v)
    #pick the candidate solution with the greatest total value
    y,maxval=findMyopicEstimate(estCont)
```

```
            #return this best candidate solution
            choice=Candidates[y]

            return choice

        def estPostDecState(DecList,CurrState):
            Cand=[]
            perc=(Par.policy['CycleMax']-
Par.policy['NotDep'])/float(Par.policy['CycleMax'])
            #Loop over entries in DecList
            for state in DecList:
                d={}
                a=[]
                #list of SRCs with changing inventory
                changelist=[entry[0] for entry in state]
                for j in range(len(CurrState)):
                    #if SRC in list of changing SRCs
                    if CurrState[j][0] in changelist:
                        #location in list of SRC
                        dex=changelist.index(CurrState[j][0])
                        b=(CurrState[j][0],round(CurrState[j][1]+perc*state[dex][1],0))
                    else:
```

```
                b=(CurrState[j][0],CurrState[j][1])
            a.append(b)
        #append the list to the list of candidate states
        d['State']=a
        d['Value']=0
        d['Visits']=1
        Cand.append(d)
    return Cand

def estStateValue(StateList,keep):

    #copy the state tracker to a temp variable, replace the last entries with the candidate states
    SC.StateTracker.sort(key=lambda k: (k['Value']),reverse=True)
    Temp=[]
    Temp=deepcopy(SC.StateTracker)


    #delete the last l entries from Temp and add the l entries from StateList
    l=len(StateList)
    Diff=len(Temp)-l
    Temp[Diff:]=[]
```

```
        for item in StateList:
            Temp.append(deepcopy(item))

        #run the DW procedure to estimate the state values of all states in the Temp
Tracker
        stateMat=StateToMat(Temp)
        phi,psi=DW.computeDW(stateMat,.5)
        values=DW.estimateValue(DW.cee,DW.dee,phi,psi)
        #associate the estimated values to their respective states
        for i in range(len(Temp)):
            #traverse the Temp list and replace the realization at the value key for each
state in the list
            Temp[i]['Value']=float(values[i])
        #trim the Temp Tracker to only contain candidate states
        del Temp[:Diff]
        #Temp.sort(key=operator.itemgetter('Value'))

        return Temp

    def StateToMat(StateStruct):
```

```
    Mat=[]

    #iterate over each entry in the state list
    for entry in StateStruct:
        #iterate over each element in the state for each entry
        for i in range(len(entry['State'])):
            #add the data from the state variable to the list
            Mat.append(entry['State'][i][1])

    #calculate the number or rows for later reshaping of the array
    rows=len(StateStruct)
    #calculate total length of the array of data
    TotalLength=len(Mat)

    Mat=np.array(Mat).reshape((rows,TotalLength/rows))

    #dimensions are rows,columns where columns=totallength/rows
    return Mat

def ValuetoArray(StateStruct):
    #converts the entries in the State dictinary at 'Value' key into an array
    Val=[]
```

```
    for entry in StateStruct:
        #append each 'Value' into the Val list
        Val.append(entry['Value'])

    rows=len(StateStruct)

    #convert the list to an array and reshape it to be rows by 1 in dimension
    Val=np.array(Val).reshape((rows,1))

    return Val

def initDWdata(StateData,delta):

    matrix=StateToMat(StateData)
    values=ValuetoArray(StateData)

    phi,psi=DW.computeDW(matrix,delta)

    DW.cee,DW.dee=DW.permDWcoeff(phi,psi,values)
```

```
def findMaxEstimate(StateStructure,ContEst,discount):
    maxim=-10000000
    index=-1

    for i in range(len(StateStructure)):
        val=discount*StateStructure[i]['Value']+ContEst[i][1]
        if val>maxim:
            index=i
            maxim=val

    return index,maxim

def findMyopicEstimate(ContEst):
    maxim=-10000000
    index=-1

    for i in range(len(ContEst)):

        if ContEst[i][1]>maxim:
            index=i
            maxim=ContEst[i][1]
```

```
    return index,maxim

def alphadecay(iteration):

    #attributed to DeGregory, alpha decay based on logistic curve
    decayparam=Par.alpha-(Par.alpha/(1+math.exp(5-10*iteration/Par.decayiters)))

    return decayparam

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def makeGoodDecSpace(top,bottom,maxlen):
    #list of decisions space items
    space=[]
    data=[]
    #loop over integers between 1 and maxlen
```

```
    for i in range(maxlen+1):
        t=it.combinations(top,i)
        b=it.combinations(bottom,i)
        sp=it.product(t,b)
        space.extend(list(sp))

    for i in space:
        temp=[]
        for j in i:
            for k in j:
                temp.append(k)
        data.append(temp)

    return data

def generateGoodDec(maxitems,maxlen,state):
    top,bottom=MF.TopBottom(maxitems,state)
    DecSpace=makeGoodDecSpace(top,bottom,maxlen)

    return DecSpace
```

A.6 DW

```
import numpy as np
```

```
import numpy.linalg as LA
import math
import scipy.linalg as spla
import time
import scipy
from scipy.spatial.distance import pdist, squareform


dee=np.array([])
DWInitial="



def computeWeight(matrix,delta):
    #computes a symmetric weight matrix
    W=np.zeros((np.shape(matrix)[0],np.shape(matrix)[0]))
    for i in range(np.shape(matrix)[0]):
        for j in range(i,np.shape(matrix)[0],1):
            q=matrix[i]-matrix[j]
            y=-LA.norm(q)/delta
            W[i][j]=math.exp(y)
            if j>i:
                W[j][i]=W[i][j]
```

```
    return W

def computeDmatrix(WMatrix):
    D=np.zeros((np.shape(WMatrix)[0],np.shape(WMatrix)[0]))

    d=np.sum(WMatrix,axis=1)
    for t in range(np.shape(d)[0]):
        D[t][t]=d[t]

    return D

def sqrtDiag(DMatrix):
    sD=np.zeros((np.shape(DMatrix)[0],np.shape(DMatrix)[0]))

    for t in range(np.shape(DMatrix)[0]):
        sD[t][t]=DMatrix[t][t]**.5

    return sD

def computePmatrix(D,W):
    P=np.dot(LA.inv(D),W)
```

```
    return P

def computeDW(matrix,delta):
    #initialize lists that will hold scaling (phi) and wavelet(psi) matrices
    phi=[]
    psi=[]

    #generate state-space graph matrices for input matrix(weight, diffusion, random
walk)
    W=Kernel(matrix,1)
    D=computeDmatrix(W)
    P=computePmatrix(D,W)

    #create indentity matrix for phi0 and add to phi list
    phimat=np.eye(np.shape(P)[0])

    #compute T matrix
    a=LA.inv(D)
    b=sqrtDiag(a)
    T=np.dot(np.dot(b,W),b)
    Q,R=np.linalg.qr(T)
    #columns of Q are DW scaling functions
```

```
        c=phimat-np.dot(Q,np.matrix.conjugate(Q))
        Qp,Rp=np.linalg.qr(c)
         return Q, Qp

    def computeDWcoeff(phi,psi,values):
        #initialize list for coefficients. Each list entry will correspond to one level in the
DW decomposition
        c=[]
        d=[]
        c=np.dot(values.T,phi)
        d=np.dot(values.T,psi)
        return c,d

    def estimateValue(c,d,phi,psi):

        ValVect=(np.dot(phi,c.T)+np.dot(psi,d.T))/2

        return ValVect

    def permDWcoeff(phi,psi,values):
        cee,dee=computeDWcoeff(phi,psi,values)
```

```
    return cee,dee


def Gaussian(x,z,sigma,axis):
    return scipy.exp((-LA.norm(x-z,axis=axis))**2/2*sigma**2)


def Kernel(X,s):
    pairwise_sq_dists = squareform(pdist(X, 'sqeuclidean'))
    K = scipy.exp(-pairwise_sq_dists / s**2)
    return K
```

A.7 StochDemand

```
import numpy as np
import DemandClass as DC
import os
import Params
import math
import pickle


def GenerateEvents(expparam, maxT):


    LastSample = 0
    interval = 0
    EventList=[]
```

```
    while LastSample <=maxT:
        #generate exponential random variable for interval between events
        interval=(np.random.exponential(scale=expparam, size=1))
        LastSample=LastSample+interval
        EventList.append(LastSample)

    return EventList

def ReadDemData(filename):

    DemData=[]
    DemData = [line.split() for line in open(filename)]

    FP=float("".join(DemData[0]))
    DemData = DemData[1:]

    return DemData,FP
    #First line contains frequency parameter data

def processDemandDirectory(path):
    DemandList=[]
```

```
        dirs=os.listdir(path) #create a list of all files in the path

        p=str(path)+'//'
        for file in dirs: #loop over each file in the directory
            DemData,FreqParam=ReadDemData(p+file) #read in the demand records
            EventList=GenerateEvents(FreqParam,Params.SimLength)

            for t in range(len(EventList)):#loop over each event instance
                for i in range(len(DemData)): #loop over each demand record from the file
                    #create the Demand objects

DemandList.append(DC.Demand(DemData[i][0],DemData[i][1],DemData[i][1]+'_'+De
mData[i][0]+str(EventList[t]),EventList[t],DemData[i][2],DemData[i][3],DemData[i][4],
))

        return DemandList

    def CreateTestDemands(path,SimLength,count):
        DemandList=[]
        dirs=os.listdir(path) #create a list of all files in the path

        p=str(path)+'//'
```

```
    for file in dirs: #loop over each file in the directory
        d={}
        d['filename']=file

        DemData,FreqParam=ReadDemData(p+file) #read in the demand records

        d['Events']=GenerateEvents(FreqParam,SimLength)
        DemandList.append(d)

    name=p+'..//Validate//dem_'+str(count)+'.pkl'
    F=open(name,'wb')
    pickle.dump(DemandList,F)
    F.close()

def InitValDemand(path,filename):
    DemandList=[]
    F=open(filename,'rb')
    E=pickle.load(F)
    F.close()
    DemandList=GenerateValDemand(path,E)

    return DemandList
```

```
def GenerateValDemand(path,structure):
    DemandList=[]
    p=path+'DemFiles//'

    for item in structure:
        #determine which file to read
        name=p+item['filename']
        #determine which demands to create
        OccurList=item['Events']

        data=[line.split() for line in open(name)]
        data=data[1:]

        for t in range(len(OccurList)):#loop over each event instance
            for i in range(len(data)): #loop over each demand record from the file
                #create the Demand objects

DemandList.append(DC.Demand(data[i][0],data[i][1],data[i][1]+'_'+data[i][0]+str(Occur
List[t]),OccurList[t],data[i][2],data[i][3],data[i][4],))

        return DemandList
```

A.8 Initialize

```
import SupplyClass as SC

import StochDemand as sd

import ADP

import pickle

import Params


def Initialize():

   SC.InitSupply('//SupplyRecords.txt')

   sd.processDemandDirectory(' //DemFiles')

   SC.InitReadReqt('//Readiness.txt')


def InitializeValidation(path, filename,supplyname):


SupplyList,SRCData=SC.InitSupply(path+'SupplyRecords_'+supplyname+'.txt')

   DemandList=sd.InitValDemand(path,filename)

   ReadReqt,ReadHist=SC.InitReadReqt(path+'Readiness.txt')


   return SupplyList,SRCData,DemandList,ReadHist,ReadReqt


def InitializeApprox(path,filename,numstates):

   F=open(path+filename,'rb')

   SC.StateTracker=pickle.load(F)
```

```
F.close()

#sort the StateTracker by values

SC.StateTracker.sort(key=lambda k: (k['Value']),reverse=True)

#Trim the state tracker to its apprporiate size

SC.StateTracker[numstates:]=[]

#initialize the DW approximation

ADP.initDWdata(SC.StateTracker,.5)

Params.DWInitial='Yes'
```

A.9 Params

```
policy = {'NotDep': 6, 'CycleMax': 24, 'Deploy': 9}

SimLength=288

#number of data point to put into state tracker for each explore iteration

Observations=20

#number of months between decisions (set to one year)

DecInterval=12

#number of months used to evaluate readiness

DepWindow=6

MetricWeight={'Now':.7,'Ready':.3}

#length of the state tracker for DW approximation purposes

LenStateTrack=625

#length to run simulation without making decisions to ensure some supply

#is assigned to demands at start of decision making functionality
```

```
BurnInPd=24
#initial value of the stepsize parameter for value function updates
alpha=.95
#number of iterations over which to gradually reduce the alpha stepsize parameter
decayiters=50000
#number of iterations for exploration phase
exploreiters=250000
#discount parameter for Bellman's equation
gamma=.97
#param to set how far back to calculate MSE
lookbackiters=100
#param to set number of consecutive error measurements within bounds to
achieve convergence
stable=100
#Number of validation runs to complete
validations=100
#maximum end-strength violation for decision optimization
maxvio=100
#parameter to control periodic reduction of state tracker by size
maxtracker=1500
DWInitial='No'
#number of parallel simulations to run
```

numthreads=1

freqVis=10

percentFreqVis=1

freqVisSaturation='No'

lookAhead=24

REFERENCES

Abbas, H.A., et al. “Computational Scenario-Based Capability Planning. “ *Proceedings GECCO*. 2008.

Alvarez, J. Fernando, et al. “Robust Fleet Sizing and Deployment for Industrial and Independent Bulk Ocean Shipping Companies.” *Information Systems and Operational Research,* 49.2 (2011): 93-107. Print.

Balakrishna, Poornima. *Scalable Approximate Dynamic Programming Models with Applications in Air* T*ransportation*. Diss. George Mason University, 2009.

Barlow, Michael, et al. “A Temporal Risk Assessment Framework for Planning a Future Force Structure.” *Proceedings of the IEEE Symposium on Computational Intelligence in Security and Defense Applications. 2007.*

Bellman, Richard. *Dynamic Programming*. Mineola, NY: Dover, 2003.

Bertsimas, Dimitris, and Melvyn Sim. “Robust Discrete Optimization and Network Flows.” *Mathematical Programming, Series* B, 98 (2003): 49-71. Print.

Betts, Richard. *Military Readiness: Concepts, Choice, Consequences*. Washington, DC: Brookings Institution, 1995. Print.

Bookbinder, James, and Kathleen Reece. “Vehicle Routing Considerations in Distribution System Design.” *European Journal of Operational Researc*, 37 (1988): 204-213.

Brandao, Jose. “A Deterministic TABU Search Algorithm for the Fleet Size and Mix Vehicle Routing Problem.” *European Journal of Operational Research*, 195 (2009): 716-728.

Braysy, Olli, et al. “A Well-Scalable Metaheuristic for the Fleet Size and Mix Vehicle Routing Problem with Time Windows.” *Expert Systems with Application*, 36 (2009): 8460-8475.

Brown, Gerald., et al. “An Optimization Model for Modernizing the Army’s Helicopter Fleet.” *Interfaces*, 21.4 (1991): 39-52.

Busoniu, Lucian, et al. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. Boca Raton, FL: CRC Press, 2010.

Cambini, Riccardo, and Rossana Riccardi, “Theoretical and Algorithmic Results for a Class of Hierarchical Fleet Mix Problems.” *European Journal of Operational Research*, 198 (2009): 741-747. Print.

Carter, Clarence, et al. “Dynamic Commitment: Wargaming Projected Forces Against the QDR Defense Strategy.” *National Defense University Strategic Forum*. 131 (1997). Print

Checco, John. *Developing Discrete Empirical Distributions for Tractable Stochastic Programming Problems with Application for U.S. Army Force Sizing*. Diss. George Mason University, 2015.

Cheon, Myun-Seok, et al. “A Modeling Framework for Railcar Fleet Sizing in the Chemical Industry.” *Industrial and Engineering Chemistry Research*, 51 (2012): 9825-9834.

Choi, Eunjeong, and Dong-Wan Tcha. “A Column Generation Approach to the Heterogeneous Fleet Vehicle Routing Problem.” *Computers and Operations Research*, 34 (2007): 2080-2095.

Clarke, G, and J. Wright. “Scheduling of Vehicles Form a Central Depot to a Number of Delivery Points.” *Operations Research*, 12.4 (1964): 568-581.

Cortes, CE, et al. “A Simulation-Based Approach for Fleet Design in a Technician Dispatch Problem with Stochastic Demand.” *Journal of the Operational Research Society*, 62 (2011): 1510-1523.

Crain, William. “Theater Campaign Analysis.” *Methods for Conducting Military Operational Analysis*. Ed. Loerch, Andrew, Larry Rainey. Alexandria, VA: Military Operations Research Society, 2007. 13-50. Print.

Davis, Paul. *Analytic Architecture for Capabilities-Based Planning, Mission-System Analysis, and Transformation*. Santa Monica, CA: RAND, 2002. Print.

Defense Science Board. “Enhancing Adaptability of U.S. Military Forces.” 2011. Web. 19 Jan. 2012.

<
http://www.acq.osd.mil/dsb/reports/EnhancingAdaptabilityOfUSMilitaryForcesB.pdf>.

DeGregory, Keith. *An Approximate Dynamic Program for Allocating Federal Air Marshals in Near Real-Time Under Uncertainty*. Diss. George Mason University, 2014.

Department of Defense. *Quadrennial Defense Review Report*. Washington, DC: Department of Defense, 2010. Print

Denardo, Eric. *Dynamic Programming: Models and Applications*. Mineola, NY:Dover, 2003.

Deshmukh, Abhi, et al. "Valuing Flexibility." 2010. Web. 2 Feb. 2012. <**http://www.dtic.mil/cgi-bin/GetTRDoc?AD=ada546882.pdf&Location=U2&doc=GetTRDoc.pdf**>

Downey, Kris, et al. "An Introduction to Smart Assemblies for Robust Design," *Research in Engineering Design*, 14. 4 (2003): 236-246. Print.

DuBois, Patrick, and Thomas Kastner. "Stochastic Analysis for Deployments and Excursions." *Military Operations Research,* 5.4 (2000): 19-35. Print.

DuBois, Patrick. *Stochastic Analysis of Resources for Deployments and Excursions. Memorandum Report CAA-MR-99-14*. Fort Belvoir, VA: Center for Army Analysis, 1999. Print.

Francis, Peter, and Karen Smilowitz. "Modeling Techniques for Periodic Vehicle Routing Problems." *Transportation Research, Part B,* 40 (2006): 872-884.

George, James. "Is Readiness Overrated?: Implications for a Tiered Readiness Force Structure." Washington, DC: CATO Institute, 1999. Web. 6 Aug. 2012. <www.cato.org/pubs/pas/pa342.pdf>

George, Abraham, and Warren Powell. "Adaptive Stepsizes for Recursive Estimation with Application in Approximate Dynamic Programming." *Machine Learning* 65 (2006): 167-198. Print.

Gerwin, Donald. "Manufacturing Flexibility: A Strategic Perspective." *Management Science*, 39.4 (1993): 395-410.

Ghanmi, Ahmed, et al. “Tactical Vehicle Fleet Mix Optimization.” *Proceedings of the Summer Computer Simulation Multiconference.*

Golden, B., et al. “Approximate Traveling Salesman Algorithms.” *Operations* Research, 28.3 Part 2 (1980): 694-711.

Golden, Bruce, et al. “The Fleet Size and Mix Vehicle Routing Problem.” *Computers and Operations* Research, 11.1 (1984): 49-66

Gosavi, Abhijit. *Simulation-Based Optimization*. Kluwer Academic Publisher: Boston, 2003.

Helms, Josh. *Randomly Generated Requirements Informed by Past Operational Deployments,* Fort Belvoir, VA: Center for Army Analysis, 2012. Print.

Jabali, Ola, et al. “A Continuous Approximation Model for the Fleet Composition Problem.” *Transportation Research, Part B,* 46 (2012): 1591-1606.

The Joint Staff. *Capabilities to Forces Integration Tool Information Paper*. Washington, DC: The Joint Staff, 2012a. Print

The Joint Staff. *CFIT Force Management Tool Information Paper*. Washington, DC: The Joint Staff, 2012b. Print.

The Joint Staff. *Joint Publication 1-02: Department of Defense Dictionary of Military and Associated Terms*. Washington, DC: The Joint Staff, 2012c. Print.

The Joint Staff. *Mitigation Options Selection Tool Information Paper*. Washington, DC: The Joint Staff, 2012d. Print.

Kok, A.L., et al. “A Dynamic Programming Heuristic for the Vehicle Routing Problem with Time Windows and European Community Social Legislation.” *Transportation Science*, 44.4 (2010): 442-454.

Lempert, Robert, et al. *Shaping the Next One Hundred Years: New Methods for Quantitative, Long-Term Policy Analysis*. Santa Monica, CA: RAND, 2003. Print.

List, George, et al. “Logistics Planning Under Uncertainty for Disposition of Radioactive Wastes.” *Computers and Operations Research*, 33 (2006): 701-723.

Liu, F-H, and S-Y Shen. “The Fleet Size and Mix Vehicle Routing Problem with Time Windows.” *Journal of the Operational Research Society*, 50 (1999): 721-732.

Liu, Shuguang, et al. “An Effective Genetic Algorithm for the Fleet Size and Mix Vehicle Routing Problem.” *Transportation Research Part E*, 45 (2009): 434-445.

Loerch, Andrew, and Linda Coblentz. “Incorporating Operations Other than War in Analysis of Force Structure Alternatives for U.S. Ground Forces.” *Proceedings of the Tenth National Symposium on Defense Management (Taiwan)*. 2002.

Loerch, Andrew. “Analysis Support for Force Structure Decisions.” *Methods for Conducting Military Operational Analysis*. Ed. Loerch, Andrew, and Larry Rainey. Alexandria, VA: Military Operations Research Society, 2007. 241-280. Print.

Mazurek, Michael, Slawomir Wesolkowski. “Minimizing Risk on a Fleet Mix Problem with a Multiobjective Evolutionary Algorithm.” *Proceedings of the 2009 IEEE Symposium on Computational Intelligence*, 2009.

McCain, John. “Ready Tomorrow: Defending American Interests in the 21st Century.” 1996. Web. 23 Jul. 2013.

<http://www.mccain.senate.gov/public/index.cfm?FuseAction=PressOffice.PressReleases&ContentRecord_id=813cc18b-7382-4c89-a55b-ed9b8f532fbd&Region_id=&Issue_id=1172f761-a830-4020-ae1b-7ec3db088fc9>.

McIlvaine, Rob. "Soldiers to Begin 2012 with Nine Month Deployments." 2011. Web. 28 Jul 2013. < http://www.army.mil/article/63073/>

Osman, Ibrahim, and Said Salhi. "Local Search Strategies for the Vehicle Fleet Mix Problem." *Modern Heuristic Search Methods*. Ed. Rayward-Smith, V.J., et al. :John Wiley and Sons, 1996.

Pessoa, 2007

Powell, Warren. *Approximate Dynamic Programming: Solving the Curses of Dimensionality*. Hoboken, NJ: Wiey, 2011.

Repoussis, P., and C. Tarantilis. "Solving the Fleet Size and Mix Vehicle Routing Problem with Time Windows via Adaptive Memory Programming." *Transportation Research, Part C*, 18 (2010): 695-712.

Salhi, Said, and Graham Rand. "Incorporating Vehicle Routing into the Vehicle Fleet Composition Problem." *European Journal of Operational Research*, 66 (1993): 313-330.

Shaffi, Kamran, et al. "Fleet Estimation for Defence Logistics Using a Multi-Objective Learning Classifier System." *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, 2011.

Simao, Hugo, et al. “An Approximate Dynamic Programming Algorithm for Large-Scale Fleet Management: A Case Application.” *Transportation Science* 43. 2 (2009): 178-197. Print.

Southerland, Jason, and Andrew Loerch. “Using Simulation and Optimization to Inform Army Force Structure Reduction Decisions.” *Proceedings of the 2014 Winter Simulation Conference*, 2014.

Spoon, Thomas. *Marathon History from 1.x to 3.x*. (Working Draft). Fort Belvoir, VA: Center for Army Analysis, 2011.

Spoon, Thomas. *Marathon 3.1415926535897932384 Design Documentation* (Working Draft). Fort Belvoir, VA: Center for Army Analysis, 2012.

Stoddard, Steven, et al. “An Analytic Approach to Army Force Structure.” *79th Military Operations Research Society Symposium.* 2011.

Stuive, Leanne, et al. “Tactical Fleet Mix Computation Using Multiobjective Evolutionary Optimization. “ *Proceedings IEEE CEC, 2010.*

Taillard, E.D. “Parallel Iterative Search Methods for Vehicle Routing Problems.” *Networks,* 23 (1993): 661-676.

Taillard, E.D. “A Heuristic Column Generation Method for the Heterogeneous Fleet VRP.” *RAIRO Operations Research*, 33.1 (1999): 1-14.

Ulusoy, Gunduz. “The Fleet Size and Mix Vehicle Routing Problem for Capacitated Arc Routing.” *European Journal of Operational Research,* 22 (1985): 329-337.

United States Army. *Army Regulation 1-1: Planning, Programming, Budgeting and Execution System*. Washington, DC: Headquarters, Department of the Army, 1994. Print.

United States Army. *Army Regulation 71-11: Total Army Analysis (TAA)*. Washington, DC: Headquarters, Department of the Army, 1995. Print.

United States Army. *Army Regulation 525-29: Army Force Generation*. Washington, DC: Headquarters, Department of the Army, 2011. Print.

United States Army. *Field Manual 3-0: Operations*. Change 1. Washington, DC: Headquarters, Department of the Army, 2011. Print.

United States Army Force Management School. "Department of Defense Planning, Programming, Budgeting, and Execution (PPBE) Process/Army Defense Planning, Programming, Budgeting, and Execution (PPBE) Process: An Executive Primer." Web. 1 Jun. 2012.

< http://www.afms1.belvoir.army.mil/primers.php>.

United States Government. *Title 10, United States Code, The Armed Forces*. Web.23 July 2013

< http://uscode.house.gov/download/title_10.shtml>

Walmsley, NS, and P Hearn. "Balance of Investment in Armoured Combat Support Vehicles: An Application of Mixed Integer Programming." *Journal of the Operational Research Society*, 55.4 (2004): 403-412.

Wesolkowski, Slawomir, Andrew Billyard. "The Stochastic Fleet Estimation Model." *Proceedings of the Spring Simulation Multiconference*, 2008.

Wesolkowski, Slawomir, et al. “Robustness and Adaptability Analysis of Future Military Air Transportation Fleets.” *Proceedings SimTecT, 2009.*

Wesolkowski, Slawomir, et al. “Multi-Objective Optimization of the Fleet Mix Problem Using the SaFER Model.” *IEEE World Congress on Computational Intelligence,* 2012a.

Wesolkowski,, Daniel Wojtaszek.”SaFESST: Stochastic Fleet Estimation Under Steady State Tasking via Evolutionary Fleet Scheduling.” *IEEE World Congress on Computational Intelligence,* 2012b.

Whitacre, James, et al. “Network Topology and Time Criticality Effects in the Modularised Fleet Mix Problem. “ *Proceedings SimTecT, 2008.*

Wojtaszek, Daniel, Slawomir Wesolkowski. “Multi-Objective Evolutionary Optimization of a Military Air Transportation Fleet Mix with the Flexibility Objective. “ *Proceedings IEEE CISDA,* 2011.

Wojtaszek, Daniel, Slawomir Wesolkowski. “Military Fleet Computation and Analysis.” *IEEE Computational Intelligence Magazine*, (2012): 53-61.

Yaman, Hande. “Formulations and Valid Inequalities for the Heterogeneous Vehicle Routing Problem.” *Mathematical Programming, Series A*, 106 (2006): 365-390.

BIOGRAPHY

Jason Alan Southerland graduated from Judson High School, Converse, Texas, in 2000. He received his Bachelor of Arts in Mathematics from the University of Texas at Austin in 2004. He has been employed as an analyst at the Center for Army Analysis since 2007 and received his Master of Science in Operations Research from George Mason University in 2008.