DECISION GUIDANCE QUERY LANGUAGE (DGQL), ALGORITHMS AND SYSTEM

by

Nathan E. Egge

A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Computer Science

Committee:

_____	Dr. Alexander Brodsky, Dissertation Director
_____	Dr. Amihai Motro, Committee Member
_____	Dr. Larry Kerschberg, Committee Member
_____	Dr. Igor Griva, Committee Member
_____	Dr. Sanjeev Setia, Department Chair
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering
Date: _____	Summer Semester 2014 George Mason University Fairfax, VA

Decision Guidance Query Language (DGQL), Algorithms and System

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

by

Nathan E. Egge
Master of Science
Virginia Tech, 2001
Bachelor of Science
Virginia Tech, 1999

Director: Alexander Brodsky, Associate Professor
Department of Computer Science

Summer Semester 2014
George Mason University
Fairfax, VA

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Alexander Brodsky for his ongoing support throughout this process, my PhD committee, Dr. Amihai Motro, Dr. Larry Kerschberg and Dr. Igor Griva for pushing me to extend my research in new directions, and finally Dr. X. Sean Wang and Mr. Hadon Nash for inspiration on early DGQL prototypes.

I would also like to recognize the many people who have been instrumental in my academic career over the past 35 years. The teachers, parents, friends, coworkers, colleagues and mentors who have encouraged and nurtured my curiosity. I am standing in the shade today because long ago someone planted a tree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF MATHEMATICAL PROGRAMMING PROBLEMS

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

A Mathematical Programming Language ..... AMPL
Adaptive Pre-processing Algorithm ..... APA
Branch & Bound ..... B&B
Constraint Programming ..... CP
Constraint Satisfaction Problem ..... CSP
Constrained Variable Name ..... CVN
Database Management System ..... DBMS
Decision Guidance Query Language ..... DGQL
Decision Support and Guidance Systems ..... DSGS
Distributed Manufacturing Network ..... DMN
Enterprise Resource Planning ..... ERP
General Algebraic Modeling System ..... GAMS
Integer Linear Programming ..... ILP
Information Technology ..... IT
Irreducible Infeasible Set ..... IIS
Linear Programming ..... LP
Mathematical Programming ..... MP
Mixed-Integer Linear Programming ..... MILP
Mixed-Integer Non-Linear Programming ..... MINLP
Multi-stage Production Network ..... MPN
Non-Linear Programming ..... NLP
Online-Decomposition Algorithm ..... ODA
Operations Research ..... OR
Optimization Programming Language ..... OPL
Piece-Wise Linear ..... PWL
Quadratic Programming ..... QP
Specially Ordered Set ..... SOS
Structured Query Language ..... SQL
User Interface ..... UI
User eXperience ..... UX

ABSTRACT

DECISION GUIDANCE QUERY LANGUAGE (DGQL), ALGORITHMS AND SYSTEM

Nathan E. Egge, Ph.D.

George Mason University, 2014

Dissertation Director: Dr. Alexander Brodsky

Decision optimization is widely used in many decision support and guidance systems (DSGS) to support business decisions such as procurement, scheduling and planning. In spite of rapid changes in customer requirements, the implementation of DSGS is typically rigid, expensive and not easily extensible, in stark contrast to the agile implementation of information systems based on the DBMS and SQL technologies. This dissertation introduces the Decision Guidance Query Language (DGQL) designed to (re-)use SQL programs for decision optimization with the goals of making DSGS implementation agile and intuitive, and leveraging existing investment in SQL-implemented systems. This dissertation addresses several related technical issues with DGQL: (1) how to annotate existing queries to precisely express the optimization semantics, (2) how to translate the annotated queries into equivalent mathematical programming (MP) formulations that can be solved efficiently using existing industrial

solvers, and (3) how to develop specialized optimization algorithms for a class of multi-stage production problems modeled in DGQL.

The algorithms for the multi-stage production network utilize the fact that only part of the problem is dynamic, e.g., the demand for output products in a manufacturing process, whereas the rest of the problem is static, e.g., the connectivity graph of the assembly processes and the cost functions of machine assemblies. An online decomposition algorithm (ODA) is developed based on offline preprocessing of static assembly components to create an approximated cost function, which is used to decompose the original problem into smaller problems and significantly improve solution quality and time complexity. The preprocessing of each static assembly component involves discretizing assembly output, finding the corresponding optimal machine configuration, and constructing a piecewise linear approximation of the assembly cost function. An adaptive preprocessing algorithm (APA) is introduced that considers only a small percentage of the discretized points by classifying outputs based on their predicted machine configuration. An initial experimental evaluation suggests that (1) machine generated MP models introduce little or no degradation in performance as compared with expertly crafted models, (2) ODA, using offline preprocessing, leads to an order of magnitude improvement in quality of solutions and optimization time as compared to MILP, and (3) APA shows significant improvement in preprocessing time with no reduction in the quality of the online solution.

1. INTRODUCTION

An increasing number of decision-guidance applications are needed to provide human decision makers with actionable recommendations on how to steer a complex process toward desirable outcomes. Examples include finding the best course of action in emergencies, deciding on business transactions within supply chains, and deciding on public policies aimed at the most positive outcomes. In these applications, decision guidance is provided in the presence of large amounts of dynamically collected data. Also, decision guidance needs to take into account the diverse constraints that model the complex engineering or business processes, composite services, resource limitations, and supply and demand. Moreover, actionable recommendations need to be produced iteratively in response to the diverse requests from human decision makers.

1.1 Motivation and Background

The state-of-the-art implementation of decision guidance applications involves two disparate technologies: (1) Database Management Systems (DBMS), used to collect, process and query large amounts of dynamically collected data, and (2) Mathematical and Constraint Programming (MP and CP), used for decision optimization, i.e., finding values for control variables that maximize or minimize an objective within given constraints. The challenge for application developers is to bridge the "impedance mismatch" between these two technologies [1]. While database programming is mostly intuitive for software

developers, they typically do not have the mathematical expertise necessary for MP or CP. In addition, much investment has already been spent on database applications. Clearly, it is desirable to leverage this investment when building decision optimization applications.

The object of this dissertation is to combine the effectiveness of MP and CP algorithms with the ease of development of database applications. This is the goal of Decision Guidance Query Language (DGQL) and the efficient evaluation algorithms proposed in this dissertation for rapid development of decision-guidance applications.

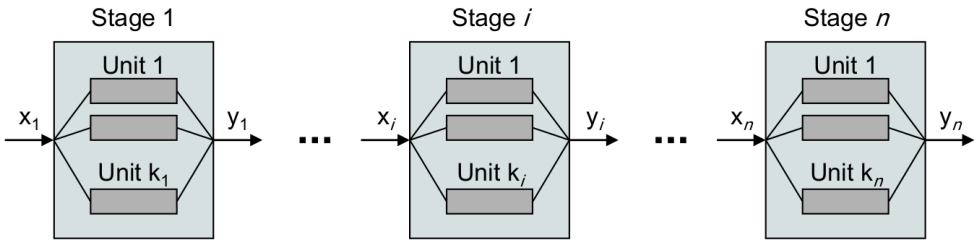

Figure 1: Multi-stage Production Network

To make the discussion concrete, consider a decision-guidance application to make recommendations to an operations officer on how to optimally run a multi-stage production process (e.g., discrete or process manufacturing) depicted in Figure 1. Assume that every stage of the process consumes a number of resources and produce a number of products. These products serve as resources for the next production stage. The products produced by the last production stage must satisfy the given demand. The

work at every production stage is distributed among a number of machines (or units), which collectively produce the products and consume the resources of their production stage. Each unit has a different cost function, as well as possibly different requirements for resources necessary to produce the products. Assume that the underlying DBMS contains this information, as well as the demand for products.

The operations officer needs to decide which machines at each stage must be operational, and the exact quantities of products that need to be produced at each stage by each operational unit. Clearly, the number of available options is very large (or even infinite for continuous quantities) and the operations officer needs to consider the current situation reflected in the DBMS data. Furthermore, the decision needs to take into account critical metrics such as time, cost and reliability of the production process. In such a scenario, it is crucial to have a system that can quickly respond to various analysis tasks, including best options under various considerations or constraints. For example, she may want to minimize the overall operational cost, while satisfying demand and operational limitations of the units, or minimize the production time given a budget and resource limitations. Obtaining the best options under constraints would involve decision optimization over a DBMS.

1.2 Research Gap

In this section, we only overview how Decision Support and Guidance Systems fit within OR and DB research and will discuss more specific related work under research directions described in Chapter 5. To build a decision guidance application, like the one in the multi-stage production example, requires both DBMS and operations research

(OR) solutions. The OR solutions include mathematical programming (MP) and constraint programming (CP) algorithms and tools. Specific classes of MP (discussed further in Chapter 4), such as linear programming (LP), mixed integer linear programming (MILP) and non-linear programming (NLP) have been hugely successful in solving large-scale, real-world optimization problems. In fact most optimization solutions used in practice are MP-based. The CP solutions have been very successful in handling combinatorial optimization problems, such as those prevalent in scheduling and planning applications.

However, using OR tools such as MP and CP poses significant challenges for software or database developers, including the need to build formal mathematical models, using a modeling language such as AMPL [2] or GAMS [3]. These formal models are typically difficult to modify or extend, as compared with well-structured application code written using programming and/or query languages, which are designed to be easily maintained.

On the other hand, database (DB) query languages and tools are more intuitive and have been adopted by many application domains. However, database query languages are not designed for decision optimization as they cannot express general decision optimization problems, notably with variables that range over infinite and/or continuous domains. There has been work in the DB area on optimizing an objective over a finite relation. This includes the "onion technique" in [4] which deals with an unconstrained linear objective function over a finite relation; the "PREFER" technique [5] used for unconstrained and linear top-*k* queries; and "Boolean+Ranking" approach [6]

for optimizing Boolean constrained ranking functions. Interesting and more general "model-based" optimization queries are considered in [7], in which I/O efficient algorithms are studied for the case when the user-defined objective and constraints are convex (for minimization problems). More generally, top-*k* queries (e.g., see survey [8]) can be considered optimization queries. However, none of these optimization query types support infinite (e.g., integers) and/or continuous (e.g., reals) domains, which is the focus of MP. In this dissertation we are interested in general purpose optimization problems, including continuous and integer domains, e.g., those considered in LP (linear programming), MILP (mixed integer linear programming), and NLP (non-linear programming), and finite domain and mixed problems (considered in CP). Furthermore, even for the finite domain, the database optimization query techniques above do not take advantage of the mathematical structure of numeric constraints and MP/CP strategies for more efficient evaluation.

Specialized optimization tools (e.g., for optimizing price-revenue, transportation, sourcing, production planning, etc.) have been developed to be ready for end users or integration with ERP/ERM systems, so that operations research expertise would not be necessary [9]. However, this approach is not extensible and does not support general decision-guidance application development. The high-level comparison between the OR approach and the DB approach is captured in Table 1.

Table 1: High-level comparison between the OR approach and the DB approach

	OR Approach Use equational languages, e.g. AMPL, GAMS, OPL	DB Approach Use DBMS and SQL
Quality and efficiency of optimization solution	• Mathematical and Constraint Programming (MP & CP) • Robust and scalable for many optimization applications	• Support *only* restricted optimization over finite but not continuous domain, e.g., top-*k* solutions • Cannot compete with MP and CP techniques when solution space is very large
Ease of use for problem modeling and system development	• Requires significant mathematical skills that software and database developers lack • Solutions difficult to modify and extend	• Intuitive and easy to use query language, e.g., SQL, with large developer community • Easy to integrate, extend and modify • Significant number of existing DB applications (models)

It is highly desirable to allow database application developers to design models akin to developing database applications, yet be able to use OR/MP algorithms by translation.

Another research challenge is related to the fact that typical MP and CP approaches assume that the entire model instance is given as input at optimization time. However, in many decision applications, parts of a model are relatively static, while some other parts are dynamic. This distinction is common in databases. For example, a database schema is typically static. In some applications the database itself is relatively static. Often the DB query structure in an application is static, while their invocation

parameters are dynamic. This distinction is extensively used in indexing and query evaluation and optimization. In contrast, typical MP and CP approaches do not utilize the separation of static and dynamic parts of a problem, investing off-line time in order to speed up on-line computation.

Given the state of the art, the development of decision-guidance systems to date amounts to a large modeling and software development effort, perhaps using database technologies such as stored procedures and application servers. Much of the application domain knowledge as well as the techniques found in various areas are encoded. In order to apply the solutions from several areas, the application must include hand crafted code that reformats, translates and expresses domain knowledge and data in the appropriate form each time a different technique is applied. As a result, the same domain knowledge and data may be represented multiple times using different mathematical abstractions. All of these challenges amount to costly software development and maintenance. Furthermore, additional development will periodically be needed when different types of new queries arise. We elaborate on related work further when discussing specific research directions in Chapter 2.

1.3 Thesis and Summary of Contributions

The thesis of this dissertation is that it is possible to design a Decision Guidance Query Language (DGQL) and efficient algorithms that:

- use standard SQL syntax with light annotation to automatically generate and solve formal optimization problems,

- introduce little or no overhead in machine generated MP/CP model formulation when compared with the equivalent expertly crafted solution,
- speed up run-time optimization by separating the static and dynamic parts of the problem and utilizing off-line preprocessing, indexing, heuristic prioritization and continuous approximation of combinatorial components.

Specifically, the key contributions of this dissertation are as follows:

1. *DGQL Syntax and Formal Semantics:* Introduction of the Decision Guidance Query Language (DGQL) and its formal syntax and semantics. DGQL allows SQL-like modeling of optimization problems and their solutions using mathematical and constraint programming. DGQL queries are essentially a sequence of regular SQL views plus the annotation of tables and views indicating columns for decision variables, check constraints and the designation of one view to compute the objective. The semantics of DGQL instantiate missing columns of tables and views so that all check constraints are satisfied, and the objective computed by the designated view is minimized or maximized.
2. *Algorithms for MP & CP Reducible Queries:* A reduction procedure is developed that generates a formal optimization problem from a DGQL query and database, solves it using the appropriate mathematical programming (MP) or constraint programming (CP) algorithms, and uses this solution to produce the DGQL output tables. The reduction procedure is proven to be correct, e.g., sound and complete as formally defined in this dissertation. Intuitively,

soundness means that every optimal solution generated through the reduction is indeed an optimal solution according to DGQL semantics and completeness means that every optimal solution according to DGQL semantics could be generated through the reduction procedure.

3. *Online Optimization Algorithms Based on Offline Pre-processing:* Considered is a class of multi-stage production problems in which only part of the problem is dynamic, e.g., the demand for the output product in a manufacturing process, whereas the rest of the problem is static, e.g., the connectivity graph of the assembly process and the cost function of machines. Proposed is the online-decomposition algorithm (ODA) based on offline preprocessing that optimizes each static problem component for discretized values of shared constraint variables and approximates the optimal aggregated utility functions. ODA uses the pre-processed approximated aggregated cost functions to decompose the original problem into smaller problems and utilizes search heuristics for the combinatorial part of the problem based on the pre-computed look-up tables.
4. *Adaptive Pre-processing Algorithm:* For the same class of multi-stage production problems, the Adaptive Pre-processing Algorithm (APA) is developed. It produces a set of preprocessed static components by considering only a small part of the discretized range of assembly output values. A parameterizable subdivision heuristic is developed to guide the selection of sample points.

5. *Proof of Concept Prototype:* A research prototype implementation of DGQL was built into the PostgreSQL relational DBMS to support solving DGQL queries which reduce to mixed integer linear programming and constraint programming problems. The prototype implementation is based on the reduction procedure and integrates with several industrial solver technologies including ILOG CPLEX and ILOG CP Optimizer. The research prototype has been effectively used to solve problems across application domains by several other students and researchers.
6. *Experimental Study:* Experimental studies were conducted to prove the efficiency of the algorithms developed in this dissertation. For the DGQL prototype based on the reduction procedure, an experimental evaluation was conducted using mixed integer linear programming supply chain problems to show that the overhead introduced by DGQL-generated optimization problems is marginal. For the online-decomposition algorithm an experimental evaluation was conducted on the class of multi-stage production network problems that showed ODA, as compared with MILP, provides an order of magnitude improvement in terms of both computational time and quality of solutions found. For the adaptive preprocessing algorithm (APA) an experimental study was conducted for the same class of problems that showed a significant improvement in preprocessing time with no reduction in the quality of the online solution.

1.4 Thesis Structure

This dissertation is organized as follows: In Chapter 2, we survey the state-of-art in decision optimization and review related works. In Chapter 3, we describe the formal syntax and semantics of DGQL through a running example. Chapter 4 presents an algorithm for DGQL reduction to mathematical programming and its proof of correctness. We give the theoretical basis for problem decomposition and introduce the Online-Decomposition Algorithm (ODA) in Chapter 5. This concept is further extended in Chapter 6 with the Adaptive Pre-processing Algorithm (APA) and an experimental evaluation of a distributed manufacturing network. In Chapter 7, we give implementation details on the DGQL research prototype. We then conclude in Chapter 8 with the current status of my research and potential future directions.

2. RELATED WORK

To build decision guidance applications like the one described in the previous chapter requires both DBMS and operations research (OR) solutions. Using OR tools such as MP and CP poses significant challenges. Among them is the requirement of full knowledge of the search space and the objective for the task at hand for efficient system operation. The OR modeling abstraction (e.g., AMPL [2] and GAMS [10]) typically requires OR expertise, which makes the development by information technology (IT) professionals challenging. In contrast, DBMS tools are more intuitive and have been adopted by many application domains. However, DBMS query languages (e.g. SQL [11] and Hibernate [12]) are not designed for decision optimization as they cannot express decision optimization problems, most notably over continuous variables. Indeed, in the continuous variable case, there are infinite possibilities to choose from that cannot be expressed as a regular database query problem. For the discrete case, when potential choices come from a large space, populating tables with all possible choices and then performing a rank query can be quite inefficient. Although query languages can handle some limited discrete optimization computations, e.g., find a tuple that has a minimal value over a finite set of discrete choices [8], even in the cases of expressible rank queries, evaluation algorithms have not typically taken advantage of MP and CP search strategies to achieve potential efficiency and flexible optimization goals. More

important, the optimization query will look very different from the reporting query, increasing the complexity of system management.

Specialized optimization tools (e.g., for optimizing price-revenue, transportation, sourcing, production planning, etc.) have been developed to be ready for end users or integration with other systems (e.g., ERP/ERM) so that operations research expertise would not be necessary [9]. However, this approach is not extensible and does not support general decision guidance application development. As more users embrace technologies to aid day-to-day tasks, there is an expectation that application developers include decision optimization in their software [13].

Within the operations research community, several high-level abstractions have been created to ease the development and management of large optimization problems. The most popular of these include AMPL [14], AIMMS [15], GAMS [16], LINGO [17], and OPL [18] all of which provide a modeling language and a set of tools that allow the user to seamlessly move between different solver technologies based on the type of mathematical or constraint programming problem provided. In the case of LINGO and OPL, an integrated development environment (IDE) allows an expert user to manipulate the model in place. All of these, however, require specialized training and have steep learning curves to master. In addition, because the intended audience is not the application developer community these tools do not typically integrate well with production data sources.

There has been extensive work on integrating data manipulation and mathematical programming modeling languages, e.g. [19] [20] [21] [22] [23]. The work in [22]

strongly advocates the need to unify models such as optimization, econometric, simulation and data models. It gives an excellent introduction and overview to model integration and integrated modeling environments, including the integration of schema, process, models with data, models with solvers, modeling paradigms, environments, and finally modeling system interfaces. The work [20] presents a methodology and provides systematic means for integrating data and mathematical modeling languages. The work [23] extends the syntax of Mathematical Programming Language (MPL) so that the index sets used in the constraint formulation are extracted from a relational database in a flexible fashion. The work [21] makes a tight integration of SQL and a mathematical programming language by creating an SQL-like syntax to define constraints and extract data from the database. The work [19] takes a similar approach and proposes using SQL to specify "blocks" which represent matrices for linear programming models. The above work on integration is significant because it makes implementation of systems easier due to tighter integration of MP modeling with data extraction and manipulation.

However, in all of these works, whether data manipulation and MP functionalities are integrated in a looser or tighter fashion, users must still work with two conceptually different paradigms: (1) data manipulation and (2) definition of MP problems. Thus, the user of those systems must still possess the expertise of mathematical programming, which most database developers do not have. In contrast, with DGQL there is no conceptual separation between data manipulation and optimization. As explained in detail in Chapter 3 the user needs only to understand the computational process of SQL, plus the concept of "inverse" computation which is the instantiation of table columns in

such a way that all of the database integrity constraints are satisfied, and a particular value computed by SQL be minimized or maximized.

The languages most closely related to DGQL are those that translate procedural algorithms into declarative constraints. These languages unify procedural and constraint semantics, such that the same program statements determine both interpretations. The language Modelica [24] supports unified models, which can define both simulation and optimization problems. Modelica models are translated into equations, which may in turn be solved by an optimizer or sorted and compiled into an efficient sequential procedure. Within pure functions (functions without side effects), Modelica can also translate procedural algorithms into constraint equations. Within these functions, Modelica gains the advantage of specifying constraints using familiar procedural operations and flow of control. However, Modelica is fundamentally an equational language, and it supports procedural algorithms only in this limited context.

The language extensions CoJava [25] and OptimJ [26] have a thoroughly procedural object oriented-syntax and semantics of Java and add the optimization semantics of an optimal non-deterministic execution path. These present the developer with no visible boundary between procedures and constraints. Familiar procedural operations and flow of control can be used uniformly throughout an entire model, or even throughout an entire software system.

A general Decision Guidance Management System (DGMS) was proposed in [1] that introduced the DG-SQL language which followed the CoJava syntax and semantics to define the search space implicitly through a set of non-deterministic query evaluations.

In addition, the DG-SQL language had support for expectation (`EXPECT`) and probability (`PROB`) operators so the same non-deterministic query definitions could be used for stochastic simulation, prediction and parameter learning.

DGQL is in the spirit of Modelica and especially CoJava to minimize the learning curve for database application developers and to take advantage of existing database queries by conforming to a well-understood syntax and semantics. While SQL is chosen as the base language to be annotated, the same idea of annotating a database program and translating it to a formal optimization problem can be applied to other database languages including Query-By-Example (QBE) [27] and object relational query languages such as Hibernate Query Language (HQL) [12]. Using this approach, DGQL gives database application developers the flexibility to move model components freely back and forth between query evaluation and declarative optimization models.

Despite the challenges mentioned earlier, decision guidance systems are being built and deployed today. However, their development to-date amounts to a large modeling and software development effort. Furthermore, as business requirements evolve, additional development will periodically be needed to incorporate these requirements into the optimization model. DGQL, on the other hand, continues the tested database tradition in building a level of abstraction that is closer to the user intuition. By doing so, many applications can take advantage of decision optimization without large upfront investment and heavy long-term maintenance expenses.

3. DGQL SYNTAX AND SEMANTICS

3.1 A Running Example

To make our discussion concrete, consider a simple example of decision guidance to support a procurement officer who manages the purchase and delivery of various items to delivery locations. Let us assume that a database is set up with the following tables:

`demand(loc, item, requested)`:
: a tuple indicates that for a delivery location (`loc`), the quantity `requested` of `item` is needed.

`supply(vendor, item, price_per_unit, available)`:
: a tuple indicates that a `vendor` can provide at most `available` quantity of `item` at the `price_per_unit`.

`orders(vendor, item, loc, quantity)`:
: a tuple indicates the order of `item` in quantity `quantity` from `vendor` for location `loc`.

With the above tables, a reporting system may have the following query to provide (total) cost information:

```
CREATE VIEW total_cost AS
      SELECT SUM(s.price_per_unit * o.quantity) AS total
      FROM orders o INNER JOIN supplier s
      ON o.vendor = s.vendor AND o.item = s.item;
```

The above assumes that the item orders are decided elsewhere, and the database is simply to record these orders. Now consider the situation that the procurement officer needs to decide these orders, with the objective of minimizing the cost. This is a typical "inverse" problem of the above reporting function. More specifically, we need to assume

that the `orders` table needs to be filled such that all the demands are satisfied while the total cost is minimized. This is a non-trivial problem especially when there are constraints like limiting the number of vendors to supply a particular location.

State-of-art procurment solutions will need to model the situation in a separate decision optimization system to make the order decisions. Our approach is to reuse the above SQL query, plus some auxiliary SQL statements and SQL-like annotations (e.g., for adding the constraints), to automatically generate mathematical programming models that will generate the optimal orders. The resulting annotated SQL query is called a DGQL query. We deliberately made the above query rather simple for easy presentation. One may imagine that more tables may be involved and some investment has already been made to debug and test the queries so that they correct reflect business rules. This ability to leverage existing investment in database design through annotation is a distinct advantage over modeling languages like AMPL, GAMS, OPL and others because it allows a clear separation between modeling the business objects and processes, and indicating *which* parameters are to be used for decision making. One can imagine a multistage decision optimization system where the output of one decision problem is used as the input to another. By allowing for heavy query reuse, new optimization problems can be created through a few annotations with little advanced planning.

Furthermore, additional information may be added to the system at different times, such as shipping options, packaging options, etc. Consider the addition of shipping options. As multiple shipping companies may be available with different pricing structures, it is useful to optimize the total shipping cost with the best shipping

options. We may have two additional tables: `shipping_options` and `shipping_orders`. An additional query can be added to provide total shipping cost. Care needs to be taken to make the item `orders` and `shipping_orders` consistent, perhaps by using integrity constraints in the database. An interesting point here is that relational database systems allow modularized development, which increases the productivity of the programmers and makes the system development more intuitive.

With the additional needed information, the procurement officer may need to also make shipping (and other) decisions so that the total cost of items and shipping is minimized. Other requirements may come in as well, such as the time of delivery requirements. Again we can "invert" the SQL function to make decision optimizations, by reusing the SQL code already developed. If shipping option is added to the item orders, two tables, namely `item_orders` and `shipping_orders`, need to be filled to make the total cost optimal. The DGQL query will be modular in exactly the same way that SQL queries are modular.

3.2 DGQL by Example

In this section, we use the existing queries in our logistics example shown in Section 3.1 to illustrate the DGQL framework. We will discuss the DGQL formal syntax and semantics in the next section. Throughout this dissertation all database examples are purely SQL with the exception of DGQL annotation that we indicate by ***ITALICS*** in the code listings.

Continuing with the example in Section 3.1 we assume that the `orders` table is the one we need to fill, with the objective of minimizing the total cost as described by the

total_cost query (view). Instead of the given database orders table, we replace it with a view created by the following SQL statement plus an augment annotation:

```
CREATE VIEW orders AS AUGMENT
      (SELECT d.loc, s.vendor, s.item
       FROM demand d
       LEFT OUTER JOIN supplier s
       ON d.item = s.item)
WITH quantity INTEGER >=0;
```

Note here the nested query is a standard query, giving all possible parings of vendor with item for each location, while the view creation here is a DGQL annotation that indicates the resulting table of the nested query to be *augmented* with an extra column called quantity with the condition that the quantity has to be a non-negative integer. The augmented column is the one to be filled by DGQL execution automatically with optimal objective (to be annotated below).

Intuitively, with any assignment of nonnegative values to the augmented quantity column to the generated orders table, there is a total cost value given by the total_cost view. The task now is to automatically generate an assignment that gives the minimum total cost for our logistics example. However, if we examine the total_cost view, we note that table demand is not used. When the reporting query is written, we have assumed that the orders table is given externally and that all the demands are satisfied by the orders. This latter assumption can be checked by the database with the following constraint:

```
CREATE VIEW requested_vs_delivered AS
      SELECT o.loc, o.item, SUM(o.quantity) AS delivered,
             d.requested
      FROM orders o
      INNER JOIN demand d
      ON o.loc = d.loc AND o.item = d.item
      GROUP BY o.loc, o.item, d.requested
      CHECK delivered >= requested;
```

This constraint checks if an item request is satisfied at a location with enough orders by comparing the total quantities of an item ordered for the location with the demand quantity at that location. This constraint needs to be explicitly given in order to generate `orders` table correctly, and DGQL supports the above constraint specification (while implementation of such a constraint in standard DBMS is not uniformly present).

Finally, we need to specify the objective explicitly. In DGQL, we use the following statement:

```
MINIMIZE total_cost;
```

Note that `total_cost` is the view defined in Section 3.1. The semantics of the ***MINIMIZE*** is to create a table for `orders` (in general, all augmented tables are instantiated), in which the last column is filled with integer values, in such a way that the total cost is minimum among all possible orders to satisfy all the item demands.

The above optimization problem of minimizing the cost can be solved rather easily since the optimal strategy is to always use the least-costly supplier for each item. This strategy can be coded directly in SQL and provide an optimal solution without using DGQL or MP (although the code will look different from the reporting query, increasing

the management complexity). However, in general the capability of SQL in providing decision optimization is limited. For example, consider the situation that (1) each vendor has a limited supply of each item, and (2) in practice, we may want to limit the number of (say up to 3) vendors to supply all the items for a particular location in order to facilitate easier tracking and management. In this case, it is not clear how to use SQL to directly code an optimal solution. In DGQL framework, this can be done with the following two additional constraints:

```
CREATE VIEW purchased_vs_available AS
      SELECT o.vendor, o.item, SUM(o.quantity) AS purchased,
             s.available
      FROM orders o
      INNER JOIN supplier s
      ON o.vendor = s.vendor AND o.item = s.item
      GROUP BY o.vendor, o.item, s.available
      CHECK purchased <= available;


CREATE VIEW vendor_restriction AS
      SELECT loc, COUNT(*) AS vendors
      FROM (SELECT DISTINCT loc, vendor
            FROM orders o
            WHERE quantity > 0)
      GROUP BY loc
      CHECK vendors <= 3;
```

The first constraint above states that the ordered item quantity from a vendor does not exceed its available quantity, and second enforces that the total number of distinct vendors for each location does not exceed 3. With the above ***CHECK*** statements, DGQL will find the optimal solution satisfying these constraints.

Note that a constraint is quite different from a selection (`WHERE`) condition. Indeed, in our example, constraints will limit the possible `orders` table instantiations to

be considered when obtaining the optimal solution by rejecting those that do not satisfy the condition.

In addition, DGQL allows modular development. If shipping and other options are to be considered, we only need to annotate the relevant queries and add appropriate constraints in order to extend the decision optimization problem. DGQL will then automatically include the relevant SQL queries, without any modifications, when performing decision optimization, thus reusing existing investment.

3.3 Formal Syntax and Semantics

We now give a precise exposition of the DGQL syntax and semantics. The philosophy of DGQL is to add annotation, in the form of operators, to standard SQL in order to express decision optimization problems. Like standard SQL, the precise syntax and semantics is best explained using relational algebra. Hence, in this section, we use the approach of adding a number of operators to the standard relational algebra. The informal, SQL-like expressions directly correspond to these operators.

As with the standard relational algebra, we assume a given set of base relations in a database each having a unique name and schema. We also assume there is a supply of globally unique relation names to be used for assignments, each with a schema. We assume the conventional notions and notation (e.g., [28]), including those of schemas, domains, tuples, relations, databases and all the traditional algebraic operations, as well as the assignment statement (or view definition). We also use a simple form of aggregation (as the “aggregation formulation” in [29]).

The annotation of DGQL consists of three operations: augmentation, constraint, and optimization. All three operators are used in the example in the previous section. Honoring the tradition, except for the optimization operator, which is the last statement in any DGQL program, we define augmentation and constraint as relational operators that can appear at any step of a relational expression.

For the sake of completeness, we list all the operators that can be used in DGQL. More precisely, we define DGQL query expressions recursively as follows:

1. (Base Case) Each relation name (base relation or not) is a DGQL query expression. The schema of the expression is exactly the relation schema of the base relation;

2. (Traditional Operators) Given DGQL query expressions e_1 and e_2, then $\pi_L(e_1)$, $\sigma_C(e_1)$, $e_1 \times e_2$, $e_1 \cup e_2$, $e_1 \cap e_2$, and $e_1 - e_2$ are all DGQL query expressions. The operations are all the traditional relational operators, and we assume the schema requirement of the operators are satisfied by e_1 and e_2. The schemas of the expressions are as defined traditionally;

3. (Arithmetic) Given DGQL query expression e, then $\gamma[B=arith(L)](e)$, where L is a subset of numeric attributes in e, B is a new attribute, and $arith(L)$ is an arithmetic expression using attributes in L;

4. (Aggregation) Given DGQL query expression e and assuming $A_1, \ldots, A_n$ and B are attributes of the schema of e, then $\delta[A_1, \ldots, A_n, f(B)](e)$ is a DGQL query expression, where f is an aggregation function (mapping bags to values); The schema of the expression is $A_1, \ldots, A_n, f(B)$; A special case is when $n = 0$, in which only $f(B)$ attribute exists for the schema of the expression.

5. (Augmentation from Relation) Given DGQL query expressions e_1 and e_2 and assuming $A_1, \ldots, A_n$ are attributes in e_1 and not attributes in e_2, $\alpha[A_1, \ldots, A_n \in e_1](e_2)$ is also a DGQL query expression; The schema of the expression is the schema of e_2 extended with $A_1, \ldots, A_n$;

6. (Augmentation from Domain) Given DGQL query expression *e*, $\alpha[A \in domain](e)$ is also a DGQL expression, where *domain* is a set of values of a specific type, e.g., $\mathbb{R}$ $[0,\infty)$ for non-negative reals. This form of augmentation is equivalent to the above one if *domain* is taken as a (possibly infinite) relation with one attribute *A* and each value is taken as a tuple;

7. (Constraint) Given DGQL query expression *e*, then $\xi[C](e)$ is a DGQL query expressions, where *C* a condition as found in selection. The schema of the expression is that of *e*;

8. (Optimization) Given a DGQL query expression *e* that computes a 1-tuple relation over a schema with a single attribute, $\omega[\texttt{min/max}](e)$ is also a DGQL query expression. The schema of the expression is that of *e*.

Definition 1 A *DGQL query* is an assignment sequence of the form

$$R_1 = e_1; \ldots; R_n = e_n; \omega$$

where each e_i is a DGQL expression and ω is either $\texttt{min}(R_k)$ or $\texttt{max}(R_k)$, where $1 \leq k \leq n$ and e_k must compute a single numeric value, or `sat`. That is, a DGQL query is a sequence of assignments followed by an optimization operator or a satisfaction operator.

A *valid* DGQL query is one that always uses a relation name that is either a base relation name or one that has been defined (i.e., on the left-hand-side of an assignment) earlier in the sequence. We henceforth assume we only deal with valid DGQL expressions if validity is not mentioned.

The example in Section 3.2 can be easily rewritten in the DGQL algebraic query form, with the each view being an assignment followed by ω=`min(total_cost)`.

We now turn to define the semantics of DGQL queries. For simplicity, we can assume that in a DGQL query, each e_i is either a single relation name (base relation name

or one of R_1, …, R_{i-1}) or a single operator applied to operands in the form of single relation names. In other words, each assignment is a "simple step" of copying a relation or applying one operation over the previously defined relations. We also assume the optimization operator only applies to single relation name, e.g., $\omega(R_k)$ where $1 \leq k \leq n$.

Definition 2 A DGQL query is a *basic assignment sequence* if each assignment is a simple step as described above.

Since a DGQL query can be rewritten into a basic assignment sequence by introducing more assignments, without loss of generality, we henceforth assume all DGQL queries are basic assignment sequences if not mentioned otherwise.

Intuitively, (1) the assignments of a DGQL query is to give all "feasible" instantiations of all the relations for the left-hand-side of each assignment such that the "input" and "output" of each assignment is "correct", i.e., given the input relations, the left-hand-side relation is actually the result of the operator applied to the given operands; (2) the operator ω as the last step of a DGQL query is to find one such feasible instantiation, when ω is `sat`, with the additional restriction that the e_k value is the minimum or maximum among all the feasible instantiations, when ω is either `min`(R_k) or `max`(R_k) respectively. For example, the DGQL query in Section 4 obtains an assignment to all the views, including the view `orders` such that the total cost is minimized. In this way, the optimization done by DGQL automatically fills the `orders` table for the logistics optimization application.

Different from the standard SQL, where the operands and the operator determine the output relation, the DGQL operator augmentation introduces non-determinism. That is, for the augmentation operator, infinitely many "output" relations may correspond to an "input" relation. Therefore, a DGQL query can have many feasible instantiations for the assignment sequence. The optimization operator at the end is to choose one feasible instantiation.

Formally, consider a DGQL query $R_1 = e_1; \ldots; R_n = e_n; \omega$. A (non-deterministic) *execution path* is a sequence $(r_1, \ldots, r_n)$ of relations over the schemas of $e_1, \ldots, e_n$, respectively. A *partial execution path* is a prefix of an execution path, i.e., a sequence of relations $(r_1, \ldots, r_i)$, $1 \le i \le n$, of relations over the schemas of $e_1, \ldots, e_i$ respectively, or the empty prefix ε.

Definition 3 The *feasibility* of a partial execution path is defined recursively as follows: The empty execution path ε is feasible. Recursively, $(r_1, \ldots, r_i)$, $1 \le i \le n$, is feasible if (1) $(r_1, \ldots, r_{i-1})$ is feasible, and (2) one of the following are satisfied:

- If e_i is a *traditional* or *aggregation* operation, then r_i is the result of e_i computation when operands from $R_1, \ldots, R_{i-1}$ are replaced with relations $r_1, \ldots, r_{i-1}$ respectively.

- If e_i is an augmentation operation $\alpha[A_1, \ldots, A_m \in R_j](R_k)$, where $1 \le k, j < i$, r_i has the schema of R_k extended with $A_1, \ldots, A_m$, and is constructed by joining a tuple from r_k with a single tuple (non-deterministically chosen) from $\pi_{A_1,\ldots,A_m}(r_j)$. Similarly, if e_i is $\alpha[A \in domain](R_k)$, where $1 \le j < i$, r_i has the schemas of R_k extended with A, and is constructed by joining a tuple from r_k with a single value (non-deterministically chosen) from the *domain* of A.

- If e_i is a constraint operation $\xi[C](R_j)$, $1 \le j < i$, the constraint C is satisfied by r_j and $r_i = r_j$.

We say $(r_1, \ldots, r_n)$ is a *feasible execution path* if it is a feasible *partial execution path* and denote by $F(d)$ the set of all feasible execution paths for $R_1 = e_1; \ldots; R_n = e_n$ and input database d.

Definition 4 The *semantics* of a DGQL query

$$R_1 = e_1; \ldots; R_n = e_n; \omega$$

is defined as the function

$$\Phi : D_1 \rightarrow 2^{D_2}$$

where D_1 is the set of all relational databases over the base database schemas, and D_2 is the set of all databases over the schema $S_1, \ldots, S_n$ of the expressions $e_1, \ldots, e_n$, respectively, as follows: For a given database d, $\Phi(d)$ is the set of all databases $d' = (r_1, \ldots, r_n)$ that satisfy the ω operator, i.e.,

Case 1: When ω=`sat`, $\Phi(d)$ is the set of all feasible execution paths, i.e., $\Phi(d) \stackrel{\text{def}}{=} F(d)$.

Case 2: When ω=`min`(R_k), $\Phi(d)$ is the set of all feasible execution paths that give the minimum value computed by r_k among all feasible execution paths, i.e., $\Phi(d) \stackrel{\text{def}}{=} \{(r_1, \ldots, r_n) \in F(d) \mid \forall\ (r'_1, \ldots, r'_n) \in F(d), r_k \leq r'_k\}$.

Case 3: When ω=`max`(R_k), $\Phi(d)$ is the set of all feasible execution paths that give the maximum value computed by r_k among all feasible execution paths, i.e., $\Phi(d) \stackrel{\text{def}}{=} \{(r_1, \ldots, r_n) \in F(d) \mid \forall\ (r'_1, \ldots, r'_n) \in F(d), r_k \geq r'_k\}$.

Note that the optimal value may not lead to a unique feasible execution path, i.e., more than one assignment to relations $r_1, \ldots, r_n$ may lead to the same optimal value for r_k. We say that a database $d' = (r_1, \ldots, r_n) \in \Phi(d)$ is a non-deterministic answer to the DGQL query. If $\Phi(d) = \varnothing$, we say that $\perp$ (to denote infeasibility) is returned and assigned to $R_1, \ldots, R_n$.

4. ALGORITHMS FOR DGQL REDUCTION TO MP AND CP

In this chapter we present a reduction procedure to translate DGQL queries to mathematical programming (MP) formulations. We first introduce MP with some discussion, and then give a sound and complete reduction procedure.

4.1 MP and its Computational Complexity

A mathematical programming (MP) problem has the form *min* $f(\hat{x})$ s.t. $C(\hat{x})$ where $\hat{x}$ is a vector of variables that range over domain $\hat{D}$, f is an *objective* function from $\hat{D}$ to the set of reals $\mathbb{R}$, and $C(\hat{x})$ is a constraint. The constraint $C(\hat{x})$ is associated with a Boolean function that, given an instantiation of vector $\hat{a} \in \hat{D}$ to $\hat{x}$, evaluates to True or False. A solution to the optimization problem is a vector $\hat{a} \in \hat{D}$ that satisfies the constraint *C* so that $f(\hat{a}) \leq f(\hat{b})$ for all vectors $\hat{b}$ that satisfy *C*. Note that *min* can be replaced with *max* in this formulation trivially by considering $-f(\hat{x})$.

Formal optimization models are solved using MP algorithms. MP algorithms heavily depend on the type of the problem, determined by the domains of the variables (e.g., reals, integers, binary, finite domain), the structure of constraints *C* (e.g., linear, quadratic, polynomial, etc.), and the form of the objective function *f*. As an example, Linear Programming (LP) is a class of MP where variables range over reals, constraints

are inequalities between linear arithmetic expressions, and the objective function is linear.

The LP problem was proven to be in *P* in [30]. However, the class of SIMPLEX algorithms [31] is still the most widely used in practice, although its worst case complexity is exponential in the dimension (i.e., the number of variables). On the other hand, even the {0,1} Integer Linear Programming (ILP), is known to be *NP*-complete [32]. The Branch and Bound algorithm [33] is a popular algorithm for ILP and MILP (mixed integer LP). While exponential in the worst case, it has the nice property of reaching intermediate (but not necessarily optimal) solutions.

Although not immediately obvious, the class of problems that can be modeled as MILP optimization is quite broad. Logical conditions (e.g. AND, OR, and NOT), Boolean implications (e.g., $p \rightarrow q$, $q \leftrightarrow r$, etc.), if-then-else statements, and other statements are all expressible using integer modeling "tricks". In addition, many solvers expose special constructs or APIs to correctly and effectively combine these techniques. These include indicator variables, Type I and Type II Special Ordered Sets (SOS) and piecewise linear functions. Thus, it is possible to model smooth non-convex functions in a MILP using piecewise linear approximations up to some epsilon [34]. This property is exploited in Chapter 5 when we develop the Online-Decomposition Algorithm by decomposing problem components via piecewise linear approximation.

In the research prototype described in Chapter 7, a commercial MILP solver is used to encode DGQL queries and derive solutions.

4.2 Reduction Procedure

The key idea of the DGQL reduction procedure is to use a symbolic table formulation to represent each step of the DGQL query. Without loss of generality, we assume that every tuple in a relation has a globally unique tuple identifier *tid* and that when new tuples are introduced (e.g., by materializing a view definition), new *tid*s will be generated accordingly.

Definition 5 A *symbolic table* is a standard relational table with the following additions: (1) an attribute can be of a constraint variable name type (or CVN type), and the domain of a CVN attribute is variable names, (2) there must be a special attribute of CVN type named `TAF`, for tuple-active-flag, with domain of binary variable names.

As an example, below is a symbolic orders table:

TAF	sup	item	loc	qty
taf[1]	S1	I1	L1	qty[1]
taf[2]	S2	I2	L2	qty[2]

where `TAF` and `qty` are CVN type attributes. The purpose of the `TAF` attribute is to indicate that a particular tuple exists in the table in a feasible execution path, and the CVN attributes allow possible instantiations.

Symbolic tables can be instantiated as regular relational tables by instantiating all of the columns with CVN attributes. More specifically, given an instantiation of variables appearing in a symbolic table, the symbolic table becomes a regular table,

minus the TAF column as follows: For each tuple *t* in the symbolic table, if taf[id] is instantiated to True, then the resulting regular table contains the tuple *t* with other CVN attributes instantiated and TAF attribute dropped. For example, in the above table, if taf[1]=True, taf[2]=False, qty[1]=20, and qty[2]=4, then the resulting table is:

sup	item	loc	qty
S1	I1	L1	20

Note that the TAF attribute is dropped.

The reduction to MP is as follows. Given a DGQL query $R_1 = e_1; \ldots; R_n = e_n; \omega$ we generate a symbolic table for each R_i. In order to represent all the feasible execution paths, constraints are to be added so all and only feasible execution paths are represented. More specifically, the *reduction result* will be the triple:

$$(\{s_1, \ldots, s_n\}, V, C)$$

where $\{s_1, \ldots, s_n\}$ are symbolic relational tables, V is the set of all constraint variables, including those that appear in $\{s_1, \ldots, s_n\}$, and C is a set of constraints over V. The schema of each s_i, $1 \le i \le n$, will be the schema of e_i extended with the TAF attribute.

Definition 6 Given the reduction result ($\{s_1, \ldots, s_n\}$, V, C) and an instantiation of variables in V to values in their corresponding domains, we say that ($r_1, \ldots, r_n$) is a *reduction instantiation* of ($\{s_1, \ldots, s_n\}$, V, C) if $\forall\ i \in \mathbb{N}$ such that $1 \leq i \leq n$, r_i is constructed by (1) replacing variables in s_i with their corresponding instantiated values, (2) eliminating all tuples in s_i where `TAF` attribute is False, and (3) removing the `TAF` attribute.

Before we discuss the procedure to obtain a reduction from a DGQL query, we formalize the correctness of the reduction result. As discussed above, the symbolic tables together with the constraints should represent all and only feasible execution paths. To formalize this idea, we have:

Definition 7 The reduction result ($\{s_1, \ldots, s_n\}$, V, C) of a DGQL query $R_1 = e_1; \ldots; R_n = e_n$; ω is said to be *correct* if:

Soundness

Every reduction instantiation ($r_1, \ldots, r_n$) of ($s_1, \ldots, s_n$) that satisfies constraint C is indeed a feasible execution path of the DGQL query; and

Completeness

Every feasible execution path is a reduction instantiation of ($s_1, \ldots, s_n$) that satisfies C.

The reduction steps follow the traditional relational algebra approach in the sense that at each step we simply generate a new symbolic table and add appropriate constraints. These steps are summarized in Algorithm 1.

Algorithm 1: Constraint Reduction

Input: A DGQL query $R_1 = e_1; \ldots; R_n = e_n; \omega$
Output: A correct reduction $(\{s_1, \ldots, s_n\}, V, C)$
Method:

1: Let $V = \varnothing$ and C = True
2: **for** each base table B **do**
3: generate a symbolic table by adding a `TAF` attribute
4: for each tuple in B with tuple id *tid*, use CVN `taf[tid]` as the value under TAF, and let C = C $\wedge$ `taf[tid]`. Add `taf[tid]` to V.
5: **end for**
6: **for** $i = 1$ to $n - 1$ **do**
7: Generate symbolic table s_i from the operator of e_i and the input symbolic tables used in e_i (these input symbolic tables must be from $s_1, \ldots, s_{i-1}$ and those corresponding to the base tables).
8: Generate the corresponding C_i
9: Add the new variables used in s_i and C_i to V
10: Let $C = C \wedge C_i$
11: **end for**
12: **return** $(\{s_1, \ldots, s_n\}, V, C)$

Algorithm 1 produces symbolic tables s_i recursively using the symbolic tables generated earlier. The base case (steps 2-5) is to make the base database tables symbolic by adding a `TAF` attribute and the corresponding variables and constraints, indicating that all the tuples are "active" in the base tables. At each step of the reduction (steps 7-10), a new symbolic table for e_i is generated and constraints and variables added to C and V, respectively. We now describe the key idea behind these steps. In Section 4.3, we give a complete description of these steps for each operator, along with the complete reduction for our running example.

Step 7 of Algorithm 1 is to generate a symbolic table for s_i given one or two symbolic tables as the input. The schema of the output table is easily found by using the

formal definition of the DGQL and traditional relational algebra operators. Setting up all the tuples in s_i is easy for most operators, but a bit challenging for others. Intuitively, we want to set up all the possible output tuples in s_i and use constraints involving `taf[tid]` to indicate if the tuple should actually exist in a feasible execution path. For example, if the operator is the selection operation, then the output table can simply be a copy of the input table, but in the constraint we code that an output tuple exists only if the selection condition is true. The reason we need to use constraint instead of simply dropping the tuple is that the selection condition may be applied to a CVN entry whose value is unknown at the reduction time.

The number of all possible tuples in s_i can be exponential in some cases. For example, when "group by" is done on a CVN attribute, we need to set up s_i to prepare for all possible groupings since we do not yet know the values of these CVN attributes. In practice, we may want to restrict "group by" only on non-CVN attributes so that the grouping will be known at the reduction time and each group will correspond to one possible tuple in s_i (although the actual values in the tuple may remain CVN and unknown at the reduction time). Details can be found in Section 4.3.

Once the symbolic output table is setup, the remaining task is to generate the constraint (Step 8). This step is basically to encode the correspondence between the input tuples and the possible output tuples. Since we know which symbolic tuple in the output is generated by which input tuples, this task is relatively easy as this is in reality a tuple level task. Again, we refer the reader to Section 4.3 for details and for a complete example.

We are now ready to address the correctness of the reduction.

Theorem 1 The reduction produced by Algorithm 1 is correct, i.e., both sound and complete.

Using the correct reduction, the result of a DGQL query can be derived as indicated in Algorithm 2. Basically, we solve the corresponding MP problem given by the correct reduction result (Step 1). If the constraint is infeasible, then we return $\perp$ (Steps 2-3). Otherwise, we use the instantiation of the variables found in the Constraint Satisfaction Problem (CSP) solution (Step 6) or MP solution (Step 8) to instantiate all the symbolic tables (Step 10).

Algorithm 2: DGQL Computation

Input: A DGQL query $R_1 = e_1; \ldots; R_n = e_n; \omega$ and a database d
Output: Query result
Method:

1. Perform Constraint Reduction (Algorithm 1) to produce $(\{s_1, \ldots, s_n\}, V, C)$
2. **if** constraint C is infeasible **then**
3. **return** $\perp$
4. **else**
5. **if** $\omega =$ `sat` **then**
6. Solve the CSP for C and V
7. **else**
8. Solve the MP problem *min/max* V_{R_k} such that C is satisfied
9. **end if**
10. Given the variable instantiation produced in Step 6 or Step 8, return the Reduction Instantiation $(r_1, \ldots, r_n)$ as described in Definition 6.
11. **end if**

Theorem 2 Algorithm 2 correctly, by Definition 4, computes the DGQL query result.

4.3 Detailed Reduction Example

The key idea behind constraint encoding of the step $R_i = e_i$ of a DGQL sequence $R_1 = e_1$; …; $R_n = e_n$; ω depends on the algebraic operator in e_i. For an *augment* operator, the idea is to replace an augmented attribute in every tuple with a CVN attribute that will hold a constraint variable name (unique per tuple id). For the *constraint* operator, the idea is to extend the constraints *C* being built in the reduction with the corresponding constraints over variables or constants in every tuple. For all the *deterministic* operators, the idea is to build a symbolic representation of the result and add the constraint that connects the variables in the output to the variables in the input. For the *optimization* operator, the idea is to represent an optimization problem that minimizes or maximizes the variable value computed by the operand of the optimization operator. In the next two subsections, we first explain these ideas of the reduction procedure via a detailed example, and then give a formal reduction procedure.

To make the reduction example more compact we use a simplified version of the example introduced in Section 3.1, by considering just a single location, ignoring `available` quantities by suppliers, and combining the `demand` and `supply` tables in a single view `supplier_selection_input` as follows:

vendor	item	ppu	requested
S1	I1	2.5	100
S1	I2	10	20
S2	I1	3.1	100
S2	I2	9	20

As before, the attribute `ppu` is the price per unit charged by the `vendor`, and `requested` is the total quantity of the `item` that is requested from all vendors. We would like to find, for each vendor and item, the quantity of the item to be purchased from that vendor so that the total purchase cost is minimal while the demand is satisfied. Below is the DGQL program that computes this:

```
CREATE VIEW orders AS AUGMENT
      supplier_selection_input
WITH qty INTEGER >= 0;

CREATE VIEW total_cost AS
      SELECT SUM(o.ppu * o.qty) AS total
      FROM orders o;

CREATE VIEW requested_vs_delivered AS
      SELECT o.item, o.requested, SUM(o.qty) AS delivered
      FROM orders o
      CHECK delivered >= requested;

MINIMIZE total_cost;
```

We first give an algebraic form of this program. The first view definition (with ***AUGMENT*** and inequality) is expressed as:

$$R_1 = \texttt{supplier\_selection\_input}$$
$$R_2 = \alpha[\texttt{qty} \in \texttt{INTEGERS}](R_1)$$
$$R_3 = \xi[\texttt{qty} \geq 0](R_2)$$

The second view `total_cost` is expressed as:

$$R_4 = \gamma[\texttt{cost = ppu * qty}](R_2)$$
$$R_5 = \delta[\texttt{total = sum(cost)}](R_4)$$

The third view `requested_vs_delivered` and the ***CHECK*** constraint are expressed as:

$R_6 = \delta[\texttt{item, requested, delivered = sum(qty)}](R_2)$
$R_7 = \xi[\texttt{delivered} \geq \texttt{requested}](R_6)$

Finally, the ***MINIMIZE*** statement is expressed as:

$\omega = \texttt{min}(R_5)$

We now show how a reduction to mathematical programming is done step by step. Initially $S_0 = \varnothing$, i.e., no tables for $R_1, \ldots, R_n$ have been constructed, $V_0 = \varnothing$, and C_0 = True. After the first assignment into R_1, $S_1 = \{s_1\}$, where s_1 is the input table extended with the attribute `TAF` with values being binary constraint variables:

tid	TAF	vendor	item	ppu	requested
tid1	taf[tid1]	S1	I1	2.5	100
tid2	taf[tid2]	S1	I2	10	20
tid3	taf[tid3]	S2	I1	3.1	100
tid4	taf[tid4]	S2	I2	9	20

Note that `tid1`, `tid2`, `tid3`, `tid4` are unique tuple identifiers. We need to add new variables for the `TAF` attribute and the constraints to indicate that they must be True, i.e., V_1 = `taf[tid1], ..., taf[tid4]` and C_1 = `taf[tid1]` $\wedge \ldots \wedge$ `taf[tid4]`.

After the second assignment into R_2, the table s_2 will be:

tid	TAF	vendor	item	ppu	requested	qty
tid5	taf[tid5]	S1	I1	2.5	100	qty[tid5]
tid6	taf[tid6]	S1	I2	10	20	qty[tid6]
tid7	taf[tid7]	S2	I1	3.1	100	qty[tid7]
tid8	taf[tid8]	S2	I2	9	20	qty[tid8]

Note that values for attribute `qty` are constraint variables `qty[tid5], ..., qty[tid8]` which will need to be instantiated by the system. The variable set V_2 will be

$V_1 \cup \{$taf[tid5], …, taf[tid8], qty[tid5], …, qty[tid8]$\}$. The constraints C_2 will be $C_1 \wedge$ taf[tid5] = taf[tid1] $\wedge$ … $\wedge$ taf[tid8] = taf[tid4] to indicate that all the tuples (with instantiated values to qty variables) must appear in the result.

After the third assignment into R_3, the table and set of variables do not change, i.e., $s_3 = s_2$ and $V_3 = V_2$, and the set of constraints is extended to indicate that the qty variables are greater than or equal to zero. That is $C_3 = C_2 \wedge$ qty[tid5] $\geq 0 \wedge$ … $\wedge$ qty[tid8] ≥ 0.

After assignment to R_4, the table for s_4 will be as follows:

tid	TAF	vendor	item	ppu	requested	qty	cost
tid9	taf[tid9]	S1	I1	2.5	100	qty[tid5]	cost[tid9]
tid10	taf[tid10]	S1	I2	10	20	qty[tid6]	cost[tid10]
tid11	taf[tid11]	S2	I1	3.1	100	qty[tid7]	cost[tid11]
tid12	taf[tid12]	S2	I2	9	20	qty[tid8]	cost[tid12]

The set of variables will be extended with the new cost variables, and the constraints will be extended to indicate how the cost is computed, i.e., $V_4 = V_3 \cup$ {taf[tid9], …, taf[tid12], cost[tid9], …, cost[tid12]} and $C_4 = C_3 \wedge$ cost[tid9] = 2.5 * qty[tid5] $\wedge$ … $\wedge$ cost[tid12] = 9 * qty[tid8].

After the fifth assignment into R_5 the table s_5 will be as follows:

tid	TAF	total
tid13	taf[tid13]	total[tid13]

The only new variable is total[tid13] and the new constraint states that the total cost is the summation of costs. Note that the constraint

total[tid13] = cost[tid9] + …+ cost[tid12]

is not sufficient since tuples in s_4 may or may not appear in the final table instantiation. Instead, the idea is to introduce and use new variables `cost'[tid9]`, …, `cost'[tid13]` and to express that for each tuple the new `cost'` variable equals `cost` if its `taf` variable is True, and equals 0 otherwise. That is, $V_5 = V_4 \cup$ {`total[tid13]`, `cost'[tid9]`, …, `cost'[tid13]`} and

$$
\begin{aligned}
C_5 = C_4 \; & \wedge \texttt{total[tid13] = cost'[tid9]} + \ldots + \texttt{cost'[tid13]} \\
& \wedge \texttt{taf[tid9]} \rightarrow \texttt{cost'[tid9] = cost[tid9]} \\
& \wedge \neg \texttt{taf[tid9]} \rightarrow \texttt{cost'[tid9] = 0} \wedge \ldots \\
& \wedge \texttt{taf[tid13]} \rightarrow \texttt{cost'[tid9] = cost[tid13]} \\
& \wedge \neg \texttt{taf[tid13]} \rightarrow \texttt{cost'[tid13] = 0}
\end{aligned}
$$

After the sixth assignment into R_6 the symbolic table s_6 will be:

tid	TAF	item	requested	delivered
tid14	taf[tid14]	I1	100	delivered[tid14]
tid15	taf[tid15]	I2	20	delivered[tid15]

The set of variables is extended with the new `taf` and `delivered` (i.e., total items purchased) variables, and the new constraint will reflect how they are computed. Again, note that we cannot just express the computation of `delivered` for `I1` as:

```
delivered[tid14] = qty[tid5] + qty[tid7]
```

because tuples `tid5` and `tid7` may or may not appear in the result instantiation. Instead, the idea is to introduce and use new variables, e.g., `qty'[tid5]` and `qty'[tid7]`, and express that each of them is equal to the corresponding `qty` variable if the corresponding `taf` variable is True (which indicates that the tuple will be in the answer), and it is equal to 0 if the corresponding `taf` variable is False. Thus, $V_6 = V_5 \cup$ {`taf[tid14]`,

taf[tid15], qty'[tid5], ..., qty'[tid8], delivered[tid14], delivered[tid15]}

and

$$
\begin{aligned}
C_6 = C_5 \wedge\ & \texttt{delivered[tid14]} = \texttt{qty'[tid5]} + \ldots + \texttt{qty'[tid7]} \\
\wedge\ & \texttt{delivered[tid15]} = \texttt{qty'[tid6]} + \ldots + \texttt{qty'[tid8]} \\
\wedge\ & \texttt{taf[tid5]} \rightarrow \texttt{qty'[tid5]} = \texttt{qty[tid5]} \\
\wedge\ & \neg\texttt{taf[tid5]} \rightarrow \texttt{qty'[tid5]} = 0 \wedge \ldots \\
\wedge\ & \texttt{taf[tid8]} \rightarrow \texttt{qty'[tid8]} = \texttt{qty[tid8]} \\
\wedge\ & \neg\texttt{taf[tid8]} \rightarrow \texttt{qty'[tid8]} = 0 \\
\wedge\ & \texttt{taf[tid14]} \leftrightarrow (\texttt{taf[tid5]} \wedge \texttt{taf[tid7]}) \\
\wedge\ & \texttt{taf[tid15]} \leftrightarrow (\texttt{taf[tid6]} \wedge \texttt{taf[tid8]})
\end{aligned}
$$

After the seventh assignment into R_7 the symbolic table and the set of variables do not change, i.e., $s_7 = s_6$ and $V_7 = V_6$, but new constraints are added to each tuple, i.e., C_7 is a conjunction of C_6 and the new constraints:

$$\texttt{delivered[tid14]} \geq 100 \wedge \texttt{delivered[tid15]} \geq 20$$

Finally, after the eighth assignment into R_8, the system extracts the objective `total[tid13]`, and solves the optimization problem:

$$\textit{min}\ \texttt{total[tid13]}\ \text{s.t.}\ C_7$$

If the problem is feasible, the solution of the optimization problem gives an instantiation of a value into each variable in V_7. The output of the DGQL program is a database $r_1, \ldots, r_8$, where each r_i is constructed from the symbolic table s_i as follows. First, all the variables are replaced with the instantiated values. Second, all tuples in which the `TAF` variable is False are discarded. Finally, the special attribute `TAF` is removed.

4.4 Formal Reduction

In this section, we present our reduction formally. We will build the reduction iteratively, by constructing a triple $Q_i = (S_i = \{s_1, \ldots, s_i\}, V_i, C_i)$, for every i, $1 \le i < n$, which will be a reduction from the partial assignment sequence expression ($R_1 = e_1$; …; $R_i = e_i$), as follows: Initially, $S_0 = \varnothing$, $V_0 = \varnothing$, and C_0 = True. Iteratively, given Q_{i-1}, $1 \le i < n$-1, Q_i is constructed based on the query expression e_i as follows:

- If e_i is a base relational name R, and r is the corresponding base relation, s_i will be constructed from r by adding the attribute `TAF` (whose value for a tuple will be the variable `TAF[tid]` where `tid` is that tuple's id). V_i will be the union of V_{i-1} and the set of new `TAF[tid]` variables. C_i will be the conjunction of C_{i-1} and the equality constraint `TAF[tid]` = True for every tuple in r.

- If e_i is a previously assigned relational name R_j ($1 \le j < i$), then Q_i is constructed by taking $V_i = V_{i-1}$, $C_i = C_{i-1}$ and $s_i = s_j$.

- If e_i is an augmentation-from-domain $\alpha[A \in D](R_j)$, where $1 \le j < i$, then Q_i is constructed as follows. The relation s_i is constructed from s_j by adding a CVN(D) attribute, and using as its value the variable name A[`tid`] for every tuple in s_j. V_i is the union of V_{i-1} with the set of all new variables in the CVN(D) attribute. Finally, $C_i = C_{i-1}$.

- If e_i is an augmentation-from-relation $\alpha[A_1, \ldots, A_m \in R_j](R_k)$, where $1 \le j,k \le n$, then Q_i is constructed as follows. The relation s_i is constructed from s_k, by adding attributes $A_1, \ldots, A_m$ of type CVN (over the corresponding domains in s_j). V_i is the union of V_{i-1} and the set of all new variables. Finally, C_i is the conjunction of C_{i-1} and the constraint

$$\bigwedge_{tid \in Tid} (TAF[tid] = True \rightarrow (A_1[tid], \ldots, A_m[tid]) \in s_j)$$

  where *Tid* is the set of all tuple ids in s_k.

- If e_i is a projection operation $\pi_L(R_j)$, where $1 \le j < i$, then Q_i is constructed as follows. $V_i = V_{i-1}$, $C_i = C_{i-1}$, and $s_i = \pi_{L'}(s_j)$, where L'=L $\cup$ {`TAF`}.

- If e_i is a (bag) union operation $R_j \cup R_k$, where $1 \le j,k < i$, then Q_i is constructed by taking $s_i = s_j \cup s_k$, $V_i = V_{i-1}$ and $C_i = C_{i-1}$.

- If e_i is an intersection operation $R_j \cap R_k$, where $1 \le j,k < i$, then Q_i is constructed as follows. Assume $A_1, \ldots, A_n, B_1, \ldots, B_m$ is the schema of both R_j and R_k, where $A_1, \ldots, A_n$ are all attributes that are non-CVN in both s_j and s_k. For every pair of tuples $tid_1 \in s_j$ and $tid_2 \in s_k$ that agree on $A_1, \ldots, A_n$, construct a tuple tid (new id), with values for $A_1, \ldots, A_n$ taken from tid_1 (which are the same as in tid_2), and the values for CVN attributes $B_1, \ldots, B_m$ being variable names $B_1[tid], \ldots, B_m[tid]$, as well as CVN({0,1}) attribute TAF. s_i is constructed as the bag of all such tuples. V_i will be the union of V_{i-1}, the set of all new (globally unique) variables above, and the set of an auxiliary CVN({0,1}) variables $E[tid]$ for every tid in s_i. To construct C_i, create the following new constraint for every tuple tid in s_i:

$$E[tid] = \text{True} \rightarrow (tid_1.B_1 = tid_2.B_1 \wedge \ldots \wedge tid_1.B_m = tid_2.B_m)$$
$$\wedge \texttt{TAF}[tid] = \texttt{TAF}[tid_1] \wedge \texttt{TAF}[tid_2] \wedge E[tid]$$

  where tid_1, tid_2 are two tuples that gave rise to tuple tid in s_i. Note that values of the B attributes (such as $tid_1.B_1$) may either be drawn from the original domain of B (e.g., if B_1 is a non-CVN attribute in s_j), or a variable name (e.g., $B_1[tid_1]$, if B_1 is a CVN attribute in s_j.) Finally, C_i is constructed as the union of C_{i-1} and the set of all new constraints above.

- If e_i is a set difference operation $R_j - R_k$, where $1 \le j,k < n$, Q_i is constructed by taking $s_i = s_j - s_k$, $V_i = V_{i-1}$, and $C_i = C_{i-1}$. Note we assume that the set difference cannot involve augmented attributes.

- If e_i is a Cartesian product $R_j \times R_k$, where $1 \le j,k < i$, then Q_i is constructed as follows. s_i is constructed by (1) computing $s_j \times s_k$, and (2) replacing two attributes R_j.TAF and R_k.TAF with a single attribute TAF of type CVN({0,1}). Recall that each tuple in the Cartesian product will have a new global tid, and then the value of TAF for that tuple will be the variable TAF[tid]. V_i is the union of V_{i-1} and the set of all new TAF variables. C_i is the conjunction of C_{i-1} and the constraints TAF[tid] = TAF[tid_1] $\wedge$ TAF[tid_2], for all tid in the Cartesian product, where tid_1 and tid_2 are the ids of tuples from s_j and s_k, respectively, that formed the tuple tid.

- If e_i is selection operation $\sigma_P(R_j)$, where $1 \le j < i$ and P is the selection predicate then Q_i is constructed as follows. Let $P = P_{reg} \wedge P_{cvn}$, where P_{reg} is the selection predicate that does not involve CVN attributes, and P_{cvn} is the selection predicate that may involve CVN attributes. s_i is constructed as $\sigma_{P_{reg}}(s_j)$. Note that s_i has globally new tuple ids, and thus the variables TAF[tid] in tuples of s_i are new unique variables. For every new variable TAF[tid] we also create an auxiliary

binary variable $PB[tid]$. V_i is constructed as the union of V_{i-1} and the set of all new variables TAF[*tid*] and $PB[tid]$ for every *tid* in s'_j. Finally, C_i is the conjunction of C_{i-1} and

$$\bigwedge_{tid \in Tid} (PB[tid] = True \leftrightarrow PP_{cvn}[tid] \wedge TAF[tid] = PB[tid] \wedge TAF[tid'])$$

where *Tid* is the set of all tuple ids in s_i, *tid'* is the id of the tuple in s_j from which the tuple *tid* in s_i originated in selection, and $PP_{cvn}[tid]$ is the constraint constructed from P_{cvn} by replacing every attribute name *A* with CVN variable *A*[*tid*].

- If e_i is an aggregation $\delta[A_1, \ldots, A_n, f(B)](R_j)$, $1 \le j < i$, then Q_i is constructed as follows. s_i is constructed from $\pi[A_1, \ldots, A_n](s_j)$ (with all new tuple ids), by first eliminating duplicates, and then adding the CVN(*D*) attribute *f*(*B*), where *D* is the domain of values returned by aggregation *f*, and CVN({0,1}) attribute TAF. V_i is the union of V_{i-1} and all new variables *f*(*B*)[*tid*], TAF[*tid*], for each *tid* in s_i, and also auxiliary variables *B*'[*tid*] for every *tid* in s_j. C_i will be the conjunction of C_{i-1} with

$$\bigwedge_{tid \in Tid} (TAF[tid] = True \rightarrow B'[tid] = B[tid] \wedge TAF[tid]=False \rightarrow B'[tid]=0)$$

where *Tid* is the set of all tuples ids in s_j, and

$$\bigwedge_{tid \in Tid} \left(f(B)[tid] = \sum_{i \in Tid(tid)} B'[i] \right)$$

where *Tid*(*tid*) is the set of tuple ids in s_j that have the same values for $A_1, \ldots, A_n$ as the tuple *tid* in s_i.

- If e_i is a constraint operation $\xi[C](R_j)$, where $1 \le j < i$, then Q_i is constructed by taking $s_i=s_j$, $V_i=V_{i-1}$, and C_i is the conjunction of C_{i-1} and

$$\bigwedge_{tid \in Tid} (C'[tid])$$

where *Tid* is the set of all tuple ids in s_j, and *C*'[*tid*] is the constraint constructed from *C* by replacing every attribute *A* with the CVN variable *A*[*tid*].

Finally, for an optimization operator[1] ω[`min`](R_j), where $1 \leq j < n$ and e_j is an expression that returns a table with a single value (i.e., the value over the single attribute of the single tuple), it is encoded as the following MP optimization problem: Let *v* be a CVN variable (or a constant) corresponding to the single value in s_j. Then, the following minimization problem is being constructed as follows:

$$\texttt{min } v \texttt{ subject to } C_{n-1}$$

For the case of ω[`max`](R_j), the reduction is the same, except that `min` is replaced with `max`. Finally, if the operator is ω[`sat`](R_j), where `sat` is a keyword that indicates that we are only interested in satisfiability (not optimization), the problem constructed is one that finds a feasible instantiation to variables in V_{n-1}.[2]

4.5 Proof of Correctness

In this section we present a formal proof of correctness for both Theorem 1 and Theorem 2.

Proof 1 First, we extend the definition of *correct reduction* to *partial reduction* as follows. A partial reduction ($\{s_1, \ldots, s_i\}$, V_i, C_i) for $1 \leq i \leq n$ of a DGQL query $R_1 = e_1$; …; $R_n = e_n$; ω is correct if:

[1] Recall it must be the last operator in the sequence.
[2] Note this can always be done by minimizing, or maximizing, a constant value.

Soundness

Every reduction instantiation (r_1, …, r_i) of (s_1, …, s_i) that satisfies constraint C_i is indeed a partial feasible execution path of the DGQL query; and

Completeness

Every partial feasible execution path (r_1, …, r_i) is a reduction instantiation of (s_1, …, s_i) that satisfies C_i.

Note, that when i=0, the partial reduction is ($\varnothing$, $V_0=\varnothing$, C_0=True) and is defined to be correct. Also note that the correct partial reduction for $i = n$ is exactly the correct reduction.

We prove the correctness of partial constraint reduction for every $1 \le i \le n$ by induction on i. For $i = 0$ the reduction is correct by definition. Inductively, assuming correctness of reduction ($\{s_1, \ldots, s_{i-1}\}$, V_{i-1}, C_{i-1}) we need to prove correctness for reduction ($\{s_1, \ldots, s_i\}$, V_i, C_i).

To prove soundness, let (r_1, …, r_i) be a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, $C_i\}$). We need to show that (r_1, …, r_i) is a partial feasible execution path. Clearly, (r_1, …, r_{i-1}) constitutes a reduction instantiation of ($\{s_1, \ldots, s_{i-1}\}$, V_{i-1}, C_{i-1}). Therefore, by induction hypothesis, (r_1, …, r_{i-1}) is a partial feasible execution path.

To prove that (r_1, …, r_i) is a partial feasible execution path, we consider cases below corresponding to the cases of the formal reduction of $R_i = e_i$:

Case 1: e_i is a base relational table r. Since the constraints added by s_i are of the form `TAF`[*tid*] = True they are all satisfied and r_i = r. Therefore (r_1, …, r_i) is a partial feasible execution path.

Case 2: e_i is a previously assigned relational name R_j, $1 \le j < i$. Since $C_i = C_{i-1}$, $r_i = r_j$ and therefore, (r_1, …, r_i) is a partial feasible execution path.

Case 3: e_i is an *augmentation-from-domain* $\alpha[A \in D](R_j)$, $1 \le j < i$. Since $C_i = C_{i-1}$, r_i is an augmentation of r_j and therefore $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 4: e_i is an *augmentation-from-relation* $\alpha[A_1, \ldots, A_m \in R_j](R_k)$, where $1 \le j, k < i$. Since the constraints added to C_{i-1} to form C_i must be satisfied, the table r_i constructed as the reduction instantiation, is also an augmentation of r_k. Therefore, $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 5: e_i is a projection operation $\pi_L(R_j)$, where $1 \le j < i$. Since $C_i = C_{i-1}$, clearly reduction instantiation r_i is a projection of r_j, and therefore $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 6: e_i is an intersection operation $R_j \cap R_k$, where $1 \le j, k < i$. The constraints added to C_{i-1} to form C_i make sure that the pairs of tuples that make `TAF` variable True are exactly those that are equal to each other and, therefore, must be in the intersection. Therefore, the reduction instantiation r_i is indeed the intersection of r_j and r_k, and thus $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 7: e_i is a set difference operation $R_j - R_k$, where $1 \le j, k < i$. The constraints added to C_{i-1} to form C_i guarantee that in the instantiation of s_i, the tuples with `TAF` = True are exactly the tuples in r_j but not in r_k. Therefore, the reduction instantiation r_i is exactly $r_j - r_k$, and thus $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 8: e_i is a Cartesian product $R_j \times R_k$, where $1 \le j, k < i$. The constraints added to C_{i-1} to form C_i guarantee that in the instantiation of s_i, the tuples with `TAF` = True are exactly those tuples composed from a pair of tuples with the corresponding `TAF` being True. Therefore, the reduction instantiation r_i is exactly the Cartesian product $r_j \times r_k$, and thus $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 9: e_i is selection operation $\sigma_P(R_j)$, where $1 \le j < i$ and P is the selection predicate. The constraints added to C_{i-1} to form C_i guarantee that the tuples in the instantiation of s_i with `TAF` = True are exactly those that satisfy the selection condition. Therefore, the reduction instantiation $r_i = \sigma_P(r_j)$, and thus $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 10: e_i is an aggregation $\delta[A_1, \ldots, A_n, f(B)](R_j)$, $1 \le j < i$. The constraints added to C_{i-1} to form C_i guarantee that the reduction instantiation r_i is

exactly the aggregation of $\delta[A_1, \ldots, A_n, f(B)](r_j)$. Therefore, $(r_1, \ldots, r_i)$ is a partial feasible execution path.

Case 11: e_i is a constraint operation $\xi[C](R_j)$, where $1 \leq j < i$. The constraints added to C_{i-1} to form C_i are exactly those that express C. Therefore, since C_i is satisfied, the reduction instantiation r_i satisfies C, and thus $(r_1, \ldots, r_i)$ is a partial feasible execution path.

We have shown that in all cases of the formal reduction $(r_1, \ldots, r_i)$ is a partial feasible execution path. Therefore the soundness portion of Theorem 1 has been proved.

To prove completeness, let $(r_1, \ldots, r_i)$ be a partial feasible execution path. We need to show that $(r_1, \ldots, r_i)$ is also a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$. Clearly, $(r_1, \ldots, r_{i-1})$ is a partial feasible execution path. Therefore, by the induction hypothesis, $(r_1, \ldots, r_{i-1})$ is also a reduction instantiation of $(\{s_1, \ldots, s_{i-1}\}, V_{i-1}, C_{i-1})$.

To prove that $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$, we consider cases below corresponding to the cases of the formal reduction $R_i = e_i$.

Case 1: e_i is a base relational table r. Since $r_i = r$, r_i is a reduction instantiation of s_i when all TAF variables are True, and thus $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$.

Case 2: e_i is a previously assigned relational name R_j, $1 \leq j < i$. Since $C_i = C_{i-1}$, $r_i = r_j$, and r_j is a reduction instantiation of s_j, r_i is an reduction instantiation of s_j. Thus, $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$.

Case 3: e_i is an *augmentation-from-domain* $\alpha[A \in D](R_j)$, $1 \leq j < i$. Since $C_i = C_{i-1}$, and r_i is an augmentation from domain of r_j, r_i is a reduction instantiation in which new variables in V_i are instantiated with values from the augmentation. Thus, $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$.

Case 4: e_i is an *augmentation-from-relation* $\alpha[A_1, \ldots, A_m \in R_j](R_k)$, where $1 \leq j, k < i$. Since r_i is an augmentation from relation of r_k, the corresponding instantiation of values into new variables in V_i must

satisfy the constraints added to C_{i-1} to form C_i. Therefore, r_i is a reduction instantiation of s_i. Thus, (r_1, …, r_i) is a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, C_i).

Case 5: e_i is a projection operation $\pi_L(R_j)$, where $1 \leq j < i$. Since $C_i = C_{i-1}$, the instantiation of values that corresponds to r_j satisfies C_i. Therefore, r_i is a reduction instantiation of s_i, and (r_1, …, r_i) is a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, C_i).

Case 6: e_i is an intersection operation $R_j \cap R_k$, where $1 \leq j, k < i$. The constraints added to C_{i-1} to form C_i make sure that the pairs of tuples that make `TAF` variable True are exactly those that are equal to each other and, therefore, must be in the intersection. Since $r_i = \pi_L(r_j)$, the variable instantiation corresponding to r_i must satisfy the constraints added to C_{i-1} to form C_i, and so r_i is reduction instantiation of s_i. Thus, (r_1, …, r_i) is a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, C_i).

Case 7: e_i is a set difference operation R_j - R_k, where $1 \leq j, k < i$. The constraints added to C_{i-1} to form C_i guarantee that in the instantiation of s_i, the tuples with `TAF` = True are exactly the tuples in r_j but not in r_k. Since $r_i = r_j - r_k$, variable instantiation corresponding to r_i must satisfy the constraints added to C_{i-1} to form C_i, and so r_i is reduction instantiation of s_i. Thus, (r_1, …, r_i) is a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, C_i).

Case 8: e_i is a Cartesian product $R_j \times R_k$, where $1 \leq j, k < i$. The constraints added to C_{i-1} to form C_i guarantee that in the instantiation of s_i, the tuples with `TAF` = True are exactly those tuples composed from a pair of tuples with the corresponding `TAF`s being True. Since $r_i = r_j \times r_k$, the variable instantiation corresponding to r_i must satisfy the constraints added to C_{i-1} to form C_i, and so r_i is a reduction instantiation of s_i. Thus, (r_1, …, r_i) is a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, C_i).

Case 9: e_i is selection operation $\sigma_P(R_j)$, where $1 \leq j < i$ and P is the selection predicate. The constraints added to C_{i-1} to form C_i guarantee that the tuples in the instantiation of s_i with `TAF` = True are exactly those that satisfy the selection condition. Since $r_i = \sigma_P(R_j)$, the variable instantiation corresponding to r_i must satisfy the constraints added to C_{i-1} to form C_i, and so r_i is reduction instantiation of s_i. Thus, (r_1, …, r_i) is a reduction instantiation of ($\{s_1, \ldots, s_i\}$, V_i, C_i).

Case 10: e_i is an aggregation $\delta[A_1, \ldots, A_n, f(B)](R_j)$, $1 \leq j < i$. The constraints added to C_{i-1} to form C_i guarantee that the reduction instantiation r_i is

exactly the aggregation of $\delta[A_1, \ldots, A_n, f(B)](r_j)$. Since $r_i = \delta[A_1, \ldots, A_n, f(B)](r_j)$, the variable instantiation corresponding to r_i must satisfy the constraints added to C_{i-1} to form C_i, and so r_i is reduction instantiation of s_i. Thus, $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$.

Case 11: e_i is a constraint operation $\xi[C](R_j)$, where $1 \leq j < i$. The constraints added to C_{i-1} to form C_i are exactly those that express C. Since r_i must satisfy C to form a partial feasible execution path, the variable instantiation corresponding to r_i must satisfy C_i, and so r_i is a reduction instantiation of s_i. Thus, $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$.

We have shown that in all cases of the formal reduction, $(r_1, \ldots, r_i)$ is a reduction instantiation of $(\{s_1, \ldots, s_i\}, V_i, C_i)$. Therefore the completeness portion of Theorem 1 has been proven and the reduction produced by Algorithm 1 is correct.

Proof 2 Let $(\{s_1, \ldots, s_n\}, V, C)$ be the reduction produced by applying Algorithm 1 to the DGQL query $R_1=e_1; \ldots; R_n=e_n; \omega$. Assume that the constraints C are feasible (otherwise Algorithm 2 returns $\perp$ which is correct). We now consider three cases:

Case 1: ω=`sat`. The constraints C and variables V make up a Constraint Satisfaction Problem. Because C is feasible, applying a CSP solver will produce an assignment of variables in V that satisfies C. We then apply the steps in Definition 6 to get the reduction instantiation $(r_1, ..., r_n)$. By Theorem 1 every reduction instantiation is a feasible execution path, thus $(r_1, ..., r_n)$ is correct.

Case 2: ω=`min`(R_k). The expression R_k returns a single numeric value that is held in the variable V_{R_k}. This is the objective in the Mathematical Programming problem: *min* V_{R_k} s.t. C. Because C is feasible, applying an MP solver will produce an assignment of variables in V that satisfies C such that V_{R_k} is minimum. We then apply the steps in Definition 6 to get the reduction instantiation $(r_1, ..., r_n)$. By Theorem 1 every reduction instantiation is a feasible execution path. Assume that r_j is not the minimal value over all feasible execution paths. Then the reduction instantiation is not minimal which contradicts the fact the MP solver found the minimal. Thus $(r_1, ..., r_n)$ is correct.

Case 3: $\omega = \texttt{max}(R_k)$. This follows from the proof of Case 2 and the observation that *max* V_{R_k} s.t. *C* is equivalent to *min* $-V_{R_k}$ s.t. *C*.

Therefore Algorithm 2: DGQL Computation correctly computes the DGQL result.

4.6 Experimental Evaluation for Reduction Overhead

The running time of mathematical programs is often sensitive to the way the problem is represented. For many classes of problems, a poor choice of mathematical representation will cause the solver to take significantly longer to find an optimal solution. One concern with DGQL (and the reduction process as described in Section 4.2) is that the extra views and tables used in modeling add substantial overhead by introducing redundant or unnecessary variables and constraints. It is possible a simpler or more concise modeling may lead to a significantly faster solution.

The hypothesis is that problems described using DGQL are solved as efficiently as when described using algebraic modeling languages such as AMPL or GAMS. As an initial step to test this hypothesis, we took the logistics optimization problem from Section 3.2 and manually created a concise AMPL formulation of the same problem. We then generated several database instances with varying problem size and compared the running time when solved using this AMPL model and through DGQL reduction. The AMPL implementation of Problem 1 can be found in below.

```
set Vendors;
set Items;
set Locations;
```

```
param PricePerUnit{Vendors,Items}>=0;
param Available{Vendors,Items}>=0;
param Requested{Locations,Items}>=0;

var Orders{Vendors,Items,Locations}>=0;
var OrderPlaced{Vendors,Items,Locations} binary;
var ItemsPurchased{Vendors,Locations}>=0;
var VendorShipped{Vendors,Locations} binary;

minimize total_cost:
    sum{v in Vendors} sum{i in Items} sum{l in Locations}
        PricePerUnit[v,i] * Orders[v,i,l];

subject to
    available_vs_purchased {v in Vendors, i in Items}:
        Available[v,i] >= sum{l in Locations} Orders[v,i,l];
    requested_vs_delivered {l in Locations, i in Items}:
        Requested[l,i] <= sum{v in Vendors} Orders[v,i,l];
    order_placed {v in Vendors, i in Items, l in Locations}:
        OrderPlaced[v,i,l] = 0 ==>
            Orders[v,i,l] = 0 else Orders[v,i,l] >= 1;
    items_purchased {v in Vendors, l in Locations}:
        ItemsPurchased[v,l] = sum{i in Items} OrderPlaced[v,i,l];
    vendor_shipped {v in Vendors, l in Locations}:
        VendorShipped[v,l] = 0 ==>
            ItemsPurchased[v,l] = 0 else ItemsPurchased[v,l]>= 1;
    vendor_restriction {l in Locations}:
        sum{v in Vendors} VendorShipped[v,l] <= 3;
```

Problem 1: AMPL implementation of Procurement Example

The DGQL query and the AMPL program were run using ILOG CPLEX 12.1 as the solver. The test was on Windows XP running on a 2.0GHz Core 2 Duo with 4GB of RAM. In both cases, the tuning parameters passed to ILOG CPLEX were the same and were typical for solving MILP problems.

The size of the logistics optimization problem here is parameterized by the number of vendors, items, and locations. Given these parameters, a problem is generated by randomly instantiating the database tables `supply` and `demand`. Note that problem

size corresponds to the number of decision variables in `orders` as well as the indicator variables (tuple-active-flags) implicit in `vendor_restrictions`. Thus, we compute this directly from the problem parameters, i.e., *decision variables = vendors * items * locs* and *indicator variables = vendors * items * locs + vendors * locs*.

Each random problem generated was solved using both the DGQL reduction and the AMPL program, and the solution time was measured. This is the CPU time spent searching for an optimal solution and does not include the time to load the program off disk, or in the case of DGQL from the database. Only random databases that contained a feasible solution were included in the results. The solution time for both the DGQL and AMPL implementations was less than 10ms for every randomly generated infeasible problem. The results of this experiment can be seen in Figure 2.

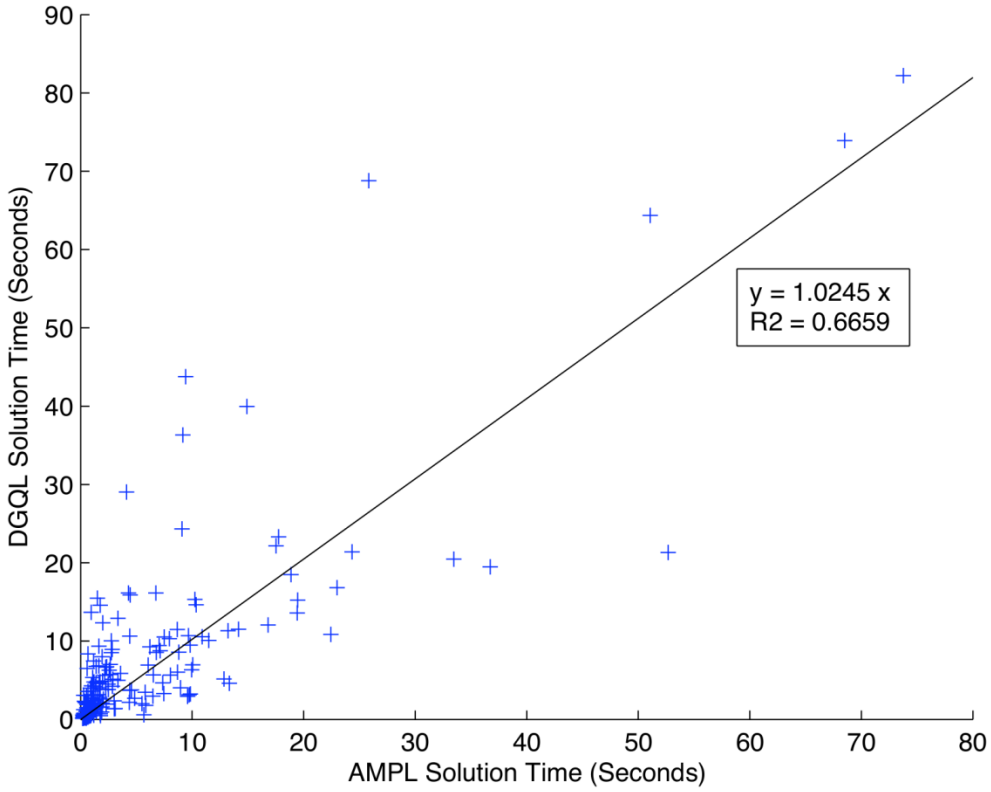

Figure 2: Scatter plot of solution time (DGQL v. AMPL)

Note in the graph, a linear regression through the origin has slope 1.0245. This implies the AMPL model is on average ≈ 2.45% faster than DGQL. This difference is small and not significant when considering the large variation in solution time between the two systems for the same problem.

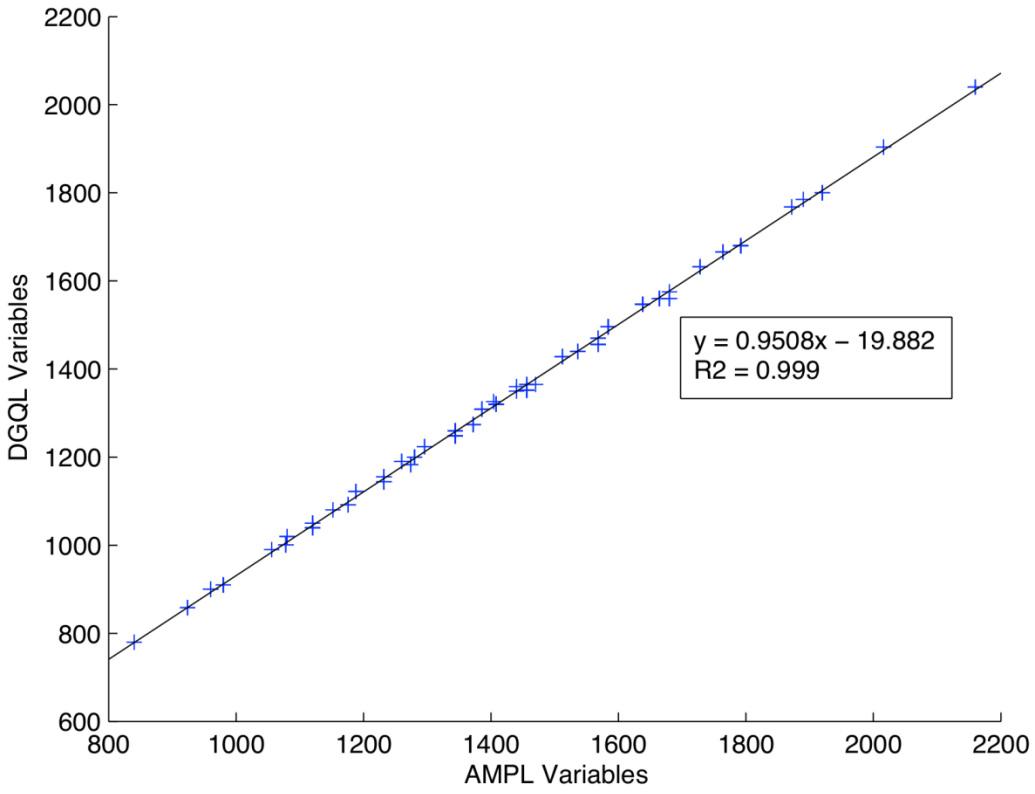

Figure 3: Scatter plot of total number of variables (DGQL v. AMPL)

On closer inspection, it appears the advantage AMPL has is limited only to those problem instances where a tight solution is found by CPLEX quickly. Our DGQL implementation uses Java and JNI to communicate with CPLEX and this may add overhead to small problems. Moreover AMPL performs limited presolving when converting problems into a special format CPLEX understands which may reduce CPLEX solution time.

Additionally, DGQL appears to do better on problem instances where a larger parameter space must be searched. This is due in part to the fact that the AMPL

formulation has an extra *vendors * locs* number of variables in its solution space (see Figure 3). As the logistics optimization problems become larger, this extra number of variables will have a measurable impact on solution time. It will be interesting to investigate further if these extra variables are a limitation of the AMPL modeling language or just our formulation of this problem.

5. ONLINE OPTIMIZATION BASED ON OFFLINE PREPROCESSING

5.1 A Motivating Example

As a motivating example, we consider the problem of optimizing a distributed manufacturing network (DMN), in which a number of products need to be manufactured. Each product can be produced by one or more assembly node processes, where each assembly is composed of several machines working in parallel, as shown in Figure 4.

For the required demand of the output products, a decision must be made on how much should be produced by each DMN assembly node, which machines should be on and their output level. Operation of each machine has associated cost, which is a function of the machine output level. Typically, the cost function has an S shape, indicating less efficient operation at the edges of its operational range, and more efficient operation in its "sweet spot". In turn, each DMN assembly node requires sub-products as its input, and those in turn are produced by other assemblies. Thus the overall DMN may involve multiple layers of DMN assembly nodes. An important question is how to make production decisions that minimize the total production cost.

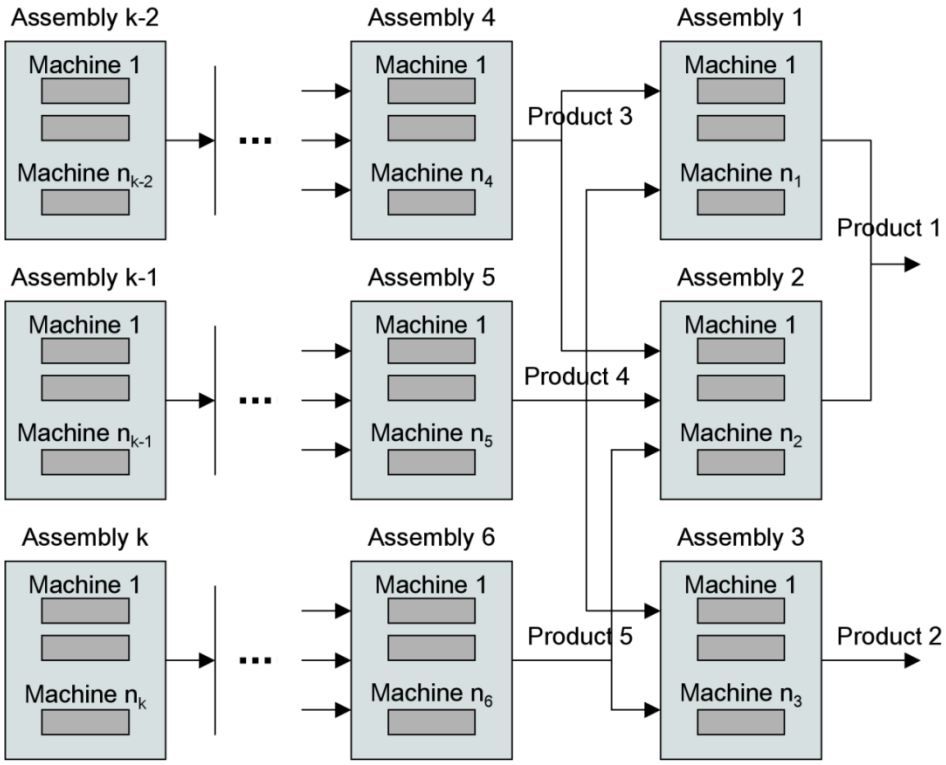

Figure 4: Distributed Manufacturing Network

However, for optimization problems such as the DMN problem in Figure 4, even when only linear constraints are used, the DGQL compiler generates a Mixed Integer Linear Programming (MILP) formulation, which involves a large number of binary decision variables. Thus, for this family of optimization problems, standard algorithms for MILP such as the *branch and bound* method are highly inefficient.

In this chapter we focus on the DMN-like problems, which are expressible in MILP, and assume that the machine costs are represented as piecewise linear functions. In addition, the operational state of each machine within a DMN assembly, i.e., ON or OFF, would be represented by a binary variable. Therefore, the problem search space is

exponential in the total number of machines, as we need to decide for each assembly the operational subset of machines and their output level (and thus associated cost).

Over the last decade many analytical and numerical approaches that study the multi-stage production model have been developed. A comprehensive review of analytical methods that estimate throughput of production systems can be found in [35]. In some cases the production system is analyzed using queuing models (see e.g. [36]). A comprehensive review of the simulation and multi-scale optimization techniques in manufacturing can be found in [37] [38]. Some of the numerical approaches used to solve similar problems rely on constraint programming (see e.g. [39] [40]). Other approaches deal with analyzing production under uncertainty [41] [42].

One of the important and promising approaches to analyze large scale production models is based on decomposition of larger problems into smaller and simpler subproblems [43] [44] [45] [46]. The way our approach uses this decomposition principle, however, is different from those described in the references above.

The key idea in this dissertation is to leverage the fact that in a class of problems like the one described, only a part of the problem is dynamic, e.g., the demand for the output product in the example, whereas the rest of the problem is static, e.g., the connectivity graph of the assemblies and the cost functions of machines in the example. A technical question arises whether we can perform (even with significant effort) preprocessing of the static parts, so that the online optimization, given the specific demand, can be done considerably faster.

In this chapter, we answer this question positively for the general multi-stage production problem and propose the *online-decomposition algorithm* (ODA) based on preprocessing. The key idea, explained in detail in Section 5.4, is based on offline preprocessing to optimize each assembly for discretized values of possible output and approximate the aggregated cost functions for optimal selection of machines. Then the online optimization uses the approximated cost functions to decompose the original *NP*-hard problem to smaller problems, and then utilize search heuristics based on pre-processed look-up tables for selection of operational machines. We also conduct an initial experimental evaluation which shows that ODA, as compared with MILP, provides an order of magnitude improvement in terms of both computational time and the quality of found solutions.

5.2 Distributed Manufacturing Network as DGQL Query

In this section, we construct a DGQL query for the manufacturing example in Section 5.1 by first constructing a pure SQL database reporting application for the problem. As usual, we forget for a moment about the optimization question at hand and consider first a database that contains the following tables:

`product(id,demand)`:
a tuple indicates that the quantity `demand` of product `id` is requested.

`assembly(id,output)`:
a tuple indicates that machines in assembly `id` produce the product `output`.

`resource(assembly,input,amount)`:
a tuple indicates that machines in `assembly` require quantity `amount` of product `input` to produce one unit of `output`.

machine(id,assembly,min,max,active,qty,c0,c1,c2,c3):
a tuple indicates machine `id` can produce between `min` and `max` of product `output` with `cost` of `c3*qty^3 + c2*qty^2 + c1*qty + c0` if `active` is *True* or will produce nothing (i.e., `qty=0`) if `active` is *False*.

Note that the `resource` table may contain more than one row with the same `assembly`. This corresponds to the machines in that assembly requiring more than one `input` product to produce the `output` product. Raw materials used in the earliest stage that do not come from a manufacturing process can be encoded by adding an assembly that has no entries in the `resource` table and contains a single machine with a potentially zero cost function.

For the moment, assume that the operations manager has decided upon a production plan, and the columns `active` and `qty` of table `machine` contain the correct values. It is then possible to develop a simple database reporting application that computes the cost of this production plan with the following SQL:

```
CREATE VIEW machine_production AS
    SELECT id, assembly,
        CASE WHEN active
            THEN qty
            ELSE 0
        END AS produced,
        CASE WHEN active
            THEN c3 * qty^3 + c2 * qty^2 + c1 * qty + c0
            ELSE 0
        END AS cost
    FROM machine;

CREATE VIEW total_cost AS
    SELECT SUM(cost) FROM machine_production;
```

Note that the cost computation above is completely independent of the product demand and is based solely on the configuration decisions of the operations manager. To extend this database application to include a report on how much product was produced versus the demand requested, we can add the following additional SQL:

```
CREATE VIEW assembly_production AS
    SELECT a.id, a.product,
        SUM(mp.produced) AS produced
    FROM assembly a
    LEFT JOIN machine_production mp
        ON mp.assembly = a.id
    GROUP BY a.id, a.product;

CREATE VIEW product_production AS
    SELECT p.id, p.demand,
        SUM(ap.produced) AS produced
    FROM product p
    LEFT JOIN assembly_production ap
        ON p.id = ap.product
    GROUP BY p.id, p.demand;

CREATE VIEW demand_vs_produced AS
    SELECT pp.id, pp.produced,
        pp.demand + SUM(ap.produced * r.amount) AS demand
    FROM product_production pp
    LEFT JOIN assembly_production ap
        ON ap.product = pp.id
    LEFT JOIN resource r
        ON r.assembly = ap.id AND r.product = pp.id
    GROUP BY pp.id, pp.produced, pp.demand;
```

The operations manager can then use the report from view `demand_vs_produced` to check that the amount produced for each product does indeed exceed the requested demand.

Let us now assume that rather than use trial and error, the operations manager would like the system to recommend the best production plan, i.e., to suggest values for

the columns `active` and `qty` of table `machine`. This needs to be done so that the total cost is minimized while the amount of products produced satisfies the requested demand. The Decision Guidance Query Language (DGQL) was designed with this goal in mind, as an extension to SQL for precisely describing optimization semantics on top of an existing database reporting application.

To turn the above pure SQL implementation into an optimization query, the following DGQL annotations are applied:

```
VARIABLE active ON machine;
VARIABLE qty >= 0 ON machine;

CONSTRAINT NOT active OR
   min <= qty AND qty <= max ON machine;
CONSTRAINT produced >= demand ON demand_vs_produced;
```

The ***VARIABLE*** annotations indicate that the columns `active` and `qty` on table `machine` are not provided, but rather are open variables that DGQL should provide values for. The ***CONSTRAINT*** annotations indicate that for each row of machine and `demand_vs_produced` the given SQL fragment should evaluate to *True*. This is identical to the **CHECK** constraint semantics for tables in the SQL language specification. While it is possible to reuse the SQL **CHECK** constraint on the table `machine`, implementation of integrity constraints on database views is not uniformly present in DBMS systems, and is therefore explicitly added in DGQL.

Finally, we need to specify the optimization objective. In DGQL this is done using the command:

```
MINIMIZE total_cost;
```

The semantics of ***MINIMIZE*** is to instantiate values for `active` and `qty` of every row in table `orders` (in general, all tables with variables are instantiated), in a way such that the objective computed in `total_cost` is minimum among all possible configurations of operational machines and produced quantities.

5.3 Problem Formulation and Complexity

To solve the optimization problem in Section 5.2, DGQL automatically translates the program into a formal mathematical programming problem. This is done using the DGQL reduction algorithm, the details of which can be found in Section 4.2. Below is the MP formulation of the DGQL program from Section 5.2 using the equational language AMPL [47]. For the parameters `Output` and `Production` note that each assembly can produce only one product and each machine can belong to only one assembly.

```
set Products;
set Assemblies;
set Machines;

param Demand{Products} >= 0;
param Resources{Assemblies,Products} >= 0;
param Production{Assemblies,Machines} binary;
param Output{Products,Assemblies} binary;
```

```
param MinQty{Machines} >= 0;
param MaxQty{Machines} >= 0;
param C0{Machine};
param C1{Machine};
param C2{Machine};
param C3{Machine};

var Active{Machines} binary;
var Qty{Machines} >= 0;
var Cost{Machines} >= 0;
var MachQty{Machines} >= 0;
var AsmQty{Assemblies} >= 0;
var Produced{Products} >= 0;

minimize total_cost:
    sum{m in Machines} Cost[m];

subject to
    machine_operation {m in Machines}:
        MinQty[m] <= Qty[m] <= MaxQty[m];
    machine_cost {m in Machines}:
        Active[m] = 0 ==> Cost[m] = 0
            else Cost[m] = C3[m]*Qty[m]^3 + C2[m]*Qty[m]^2 +
                           C1[m]*Qty[m] + C0[m];
    machine_production {m in Machines}:
        Active[m] = 0 ==> MachQty[m] = 0
            else MachQty[m] = Qty[m];
    assembly_products {a in Assemblies}:
        AsmQty[a] =
            sum{m in Machines} Production[a,m]*MachQty[m];
    product_production {p in Products}:
        Produced[p] =
            sum{a in Assemblies} Output[p,a]*AsmQty[a];
    demand_vs_produced {p in Products}:
        Produced[p] >= Demand[p] +
            sum{a in Assemblies} Resources[a,p]*AsmQty[a];
```

Problem 2: Distributed Manufacturing Network implemented in AMPL

In this formulation, we can see that the problem is made up of parameters, variables, and constraints. Different optimization problems can be created with the same problem structure through different input parameters. In practice though, some of these parameters are dynamic and some are more static. In this example, the product demand is market driven and may change quite frequently due to market conditions, etc. Thus the

`Demand` parameter can be considered dynamic. In contrast, the factory machines represent a fixed and costly investment and are unlikely to change. The assembly connectivity graph described in `Production`, `Output`, and `Resources` as well as the machine operating range and cost described in `MinQty`, `MaxQty`, `C0`, `C1`, `C2` and `C3` can be considered static parameters of the problem.

The key idea behind the online-decomposition algorithm (ODA) is to reformulate the MP problem in such a way as to take advantage of this distinction between static and dynamic input parameters. Consider the following equivalent form of the objective:

```
var AsmCost{Assemblies} >= 0;

minimize total_cost:
    sum{a in Assemblies} AsmCost[a];

subject to
    assembly_cost {a in Assemblies}:
        AsmCost[a] =
            sum{m in Machines} Production[a,m]*Cost[m];
```

Problem 3: Equivalent objective implementation in AMPL

The objective is now a function of the assembly cost alone. Intuitively, the overall optimization problem can be broken down into components corresponding to each assembly. Each component contains variables that are used to describe the assembly configuration as well as variables that are used to capture global constraints and the objective function. We call the configuration variables *internal variables* and the variables that are used across components the *interface variables*. Given the reformulated objective above, `Active`, `Qty`, `Cost` and `MachQty` are the internal variables and `AsmQty`

and `AsmCost` are the interface variables. Note that other than computing the values for the interface variables, there are no other references to the internal variables anywhere in the formulation.

In addition, all of the problem parameters referenced internally by the assembly components are the ones identified as static. The dynamic `Demand` parameters are only referenced in global constraints which are in terms of the interface variables `AsmQty`. These conditions suggest that it is possible to preprocess each component and reformulate the original problem in terms of just the interface variables.

Suppose we had an oracle function $CompCost_a(qty)$ which returned the cost of manufacturing *qty* amount of `output` for the given assembly *a*. We could then reformulate the original MP problem as follows:

```
set Products;
set Assemblies;

param Demand{Products} >= 0;
param Resources{Assemblies,Products} >= 0;
param Output{Products,Assemblies} binary;
param AsmMinQty{Assemblies} >= 0;
param AsmMaxQty{Assemblies} >= 0;

var Active{Assemblies} binary;
var Qty{Assemblies} >= 0;
var AsmCost{Assemblies} >= 0;
var AsmQty{Assemblies} >= 0;
var Produced{Products} >= 0;

minimize total_cost:
    sum{a in Assemblies} AsmCost[a];

subject to
    assembly_operation {a in Assemblies}:
        AsmMinQty{a} <= Qty{a} <= AsmMaxQty{a};
    assembly_products {a in Assemblies}:
        Active[a] = 0 ==> AsmQty[a] = 0
```

```
        else AsmQty[a] = Qty[a];
assembly_cost {a in Assemblies}:
    AsmCost[a] = CompCost(a,AsmQty[a]);
product_production {p in Products}:
    Produced[p] =
        sum{a in Assemblies} Output[p,a]*AsmQty[a];
demand_vs_produced {p in Products}:
    Produced[p] >= Demand[p] +
        sum{a in Assemblies} Resources[a,p]*AsmQty[a];
```

Problem 4: Reformulation of Distributed Manufacturing Network

Note that `Active` and `Qty` in this formulation are variables of each assembly and the parameters `AsmMinQty` and `AsmMaxQty` can be computed from `MinQty` and `MaxQty` as follows:

$$AsmMinQty_a = \min_{m \in a}(MinQty_m)$$

$$AsmMaxQty_a = \sum_{m \in a} MaxQty_m$$

Consider the improvement in solution time of the reformulated problem compared to the original. Assuming both the machine cost function `c3*qty^3 + c2*qty^2 + c1*qty + c0` and the assembly cost function $CompCost_a(qty)$ can be reasonably approximated with piecewise linear functions, the reformulated problem and the original contain only linear equations in binary and real valued variables. These problems fall into the class of Mixed Integer Linear Programs (MILP) and are solvable using standard searching algorithms such as branch and bound [32]. In both cases, the search space is exponential in the number of binary variables. However, in the reformulated problem, the *n* binary variables associated with each machine of an assembly are replaced by a single binary variable for the entire assembly. This represents a significant reduction in the

search space and has the potential of finding good working solutions for otherwise intractable problems.

5.4 Principles of Preprocessing and Decomposition

As we have shown, being able to define the assembly cost function $CompCost_a(qty)$ is critical to problem reformulation. Indeed, the claimed reduction in problem complexity relies heavily on the fact that this function can be tightly approximated with a piecewise linear function. More generally, the same decomposition technique can be applied to any optimization problem so long as the following pre-computability conditions hold:

1. The optimization problem:

   $$min \ f(\hat{x}) \ s.t. \ C(\hat{x})$$

   is of the form:

   $$\begin{aligned} &min \ f_1(\hat{x}_1, \hat{y}) + \ldots + f_n(\hat{x}_n, \hat{y}) \\ &s.t. \ C_1(\hat{x}_1, \hat{y}) \wedge \ldots \wedge C_n(\hat{x}_n, \hat{y}) \wedge C_0(\hat{y}) \end{aligned} \quad (1)$$

   where variables in $\hat{x}_i$ only appear in f_i and C_i.

2. All dynamic parameters are contained in $C_0(\hat{y})$.

Here the vector $\hat{y}$ represents the interface variables and the $\hat{x}_i$ vectors are the internal variables. As in the running example, we would like to find optimal values for the interface variables $\hat{y}$ and use the pre-computed values for the internal variables to fix the combinatorial choices for each component. In the general case, we would like to

reformulate (1) in terms of only $\hat{y}$. This can be done by defining new component objective functions as:

$$F_1(\hat{y}) = min_{\hat{x}_i} f_1(\hat{x}_1, \hat{y})$$
$$\vdots \quad \vdots$$
$$F_n(\hat{y}) = min_{\hat{x}_n} f_n(\hat{x}_n, \hat{y})$$

Similarly the component constraints can be redefined as:

$$K_1(\hat{y}) = (\exists x_1) C_1(\hat{x}_1, \hat{y})$$
$$\vdots \quad \vdots$$
$$K_n(\hat{y}) = (\exists x_n) C_n(\hat{x}_n, \hat{y})$$

Note that while both of these definitions are mathematically well defined, in practice $F_1(\hat{y}), \ldots, F_n(\hat{y})$ and $K_1(\hat{y}), \ldots, K_n(\hat{y})$ may not have a simple analytical form. We can then reformulate the original problem in terms of F_i and K_i as follows:

$$min\ F_1(\hat{y}) + \ldots + F_n(\hat{y}) \tag{2}$$
$$s.t.\ K_1(\hat{y}) \wedge \ldots \wedge K_n(\hat{y}) \wedge C_0(\hat{y})$$

This reformulated problem is now only a function of the interface variables $\hat{y}$. We claim without proof that a solution to (2) is a partial solution to the original problem (1). More formally, if $\hat{y}^*$ is a solution to (2) then there exists a solution $(\hat{x}_1^*, \ldots, \hat{x}_n^*, \hat{y}^*)$ to (2). Similarly the converse is true. A solution to the original problem (1) gives a solution to (2). If $(\hat{x}_1^*, \ldots, \hat{x}_n^*, \hat{y}^*)$ is a solution to (1) then $\hat{y}^*$ is a solution to (2).

Given a solution $\hat{y}^*$ to the reformulated problem (2), it is possible to "decompose" the large combinatorial problem in (1) into *n* smaller combinatorial problems which may have a considerably smaller search space and can be solved independently of each other.

$$\begin{array}{l} min_{x_1}\ f_1(\hat{x}_1, \hat{y}^*)\ s.t.\ C_1(\hat{x}_1, \hat{y}^*) \\ \vdots \\ min_{x_n}\ f_n(\hat{x}_n, \hat{y}^*)\ s.t.\ C_n(\hat{x}_n, \hat{y}^*) \end{array} \quad (3)$$

Each of the *n* optimization problems in (3) is a function of only $\hat{x}_i$ and $\hat{y}$. We further claim that a solution to (2) along with the solutions to (3) is a solution to the original problem (1). That is, if $\hat{y}^*$ is a solution to (2) and $\hat{x}_1^*, \ldots, \hat{x}_n^*$ are solutions to the *n* optimization subproblems in (3) then $(\hat{x}_1^*, \ldots, \hat{x}_n^*, \hat{y}^*)$ is a solution to (1).

The general reformulation above shows how any optimization problem that meets the precomputability conditions can be "decomposed" into *n* optimization subproblems. Note that the claims made in this reformulation are only valid when the solutions to (2) and (3) are exact. As stated above, the main challenge is that the definitions for $F_1, \ldots, F_n$ and $K_1, \ldots, K_n$ may not have a simple analytical form that can be used with existing solver technologies (e.g., LP, MILP, QP, NLP, etc.). Thus while the decomposition above does in fact reduce the size of the combinatorial search space, it may produce problem formulations that cannot be solved in practice.

The approach this dissertation takes with the online-decomposition algorithm is to use approximations for $F_1, \ldots, F_n$ and $K_1, \ldots, K_n$ that have an analytical form which can be solved in practice. The resulting solution $\hat{y}^*$ is thus an approximate partial solution to (1). This approximate partial solution can still be used to reduce the combinatorial search space through decomposition into (3), however the resulting solution $(\hat{x}_1^*, \ldots, \hat{x}_n^*, \hat{y}^*)$ is not only an approximate solution, but it may not even be a feasible solution to (1). Thus as a

final step, the combinatorial variables in $\hat{x}_1, \ldots, \hat{x}_n$ are fixed based on the solution found in $\hat{x}_1^*, \ldots, \hat{x}_n^*$ so that the original problem (1) now contains no combinatorial search space. This problem can then be solved producing a reasonably good, feasible solution to (1). This idea forms the basis for the online-decomposition algorithm presented in the next section for the multi-stage production problem.

5.5 Online-Decomposition Algorithm

We now present the online-decomposition algorithm (ODA) for the multi-stage manufacturing problem in Section 5.2. A key observation in this algorithm is that for each assembly the cost function is univariate in a bounded variable, i.e., a function of only $qty \in [AsmMinQty_a, AsmMaxQty_a]$. Thus it is possible to create a piecewise linear approximation of $CompCost_a(qty)$ through offline preprocessing by discretizing *qty* and solving the minimization subproblem for the assembly a below:

```
set Machines;

param Quantity >= 0;
param MinQty{Machine} >= 0;
param MaxQty{Machine} >= 0;
param C0{Machine};
param C1{Machine};
param C2{Machine};
param C3{Machine};

var Active{Machines} binary;
var Qty{Machines} >= 0;
var Cost{Machines} >= 0;
var MachQty{Machines} >= 0;

minimize total_cost:
    sum{m in Machines} Cost[m];
```

```
subject to
    machine_operation {m in Machines}:
        MinQty[m] <= Qty[m] <= MaxQty[m];
    machine_cost {m in Machines}:
        Active[m] = 0 ==> Cost[m] = 0
            else Cost[m] = C3[m]*Qty[m]^3 + C2[m]*Qty[m]^2 +
            C1[m]*Qty[m] + C0[m];
    machine_production {m in Machines}:
        Active[m] = 0 ==> MachQty[m] = 0
            else MachQty[m] = Qty[m];
    demand_vs_produced:
        Quantity >= sum{m in Machines} MachQty[m];
```

Problem 5: AMPL implementation of assembly subproblem

Note that because *qty* is real-valued, $CompCost_a(qty)$ is only accurate for those points in our discretization. Along with the optimal cost value `total_cost`, after solving each subproblem, the optimal machine configuration in `Active` is captured for later use in ODA. Algorithm 3 gives the preprocessing step.

Algorithm 3: Subproblem Preprocessing

Input: Assembly *a*, sampling resolution *r*
Output: List of samples (*qty*, *cost*, *Active*)
Method:
1: Let $S \leftarrow \emptyset$
2: Let $qty \leftarrow AsmMinQty_a$
3: **while** $qty \leq AsmMaxQty_a$ **do**
4: Solve MP subproblem in Problem 5 for a with `Quantity` = *qty*
5: Add (*qty*, `total_cost`, `Active`) to *S*
6: $qty \leftarrow qty + r$
7: **end while**
8: Return *S*

In the online portion of the algorithm, the MP problem in Problem 4 is solved with the dynamic parameters `Demand` using the piecewise linear approximations of $CompCost_a(qty)$ found in preprocessing. As noted earlier these approximations are only accurate for those points in our discretization. The solution values for `AsmQty` do not necessarily lie on those points. Thus the solution found in Problem 4 is only an approximate solution to the original problem, and in fact the `total_cost` found may be infeasible.

To account for this, the original MP problem in Problem 2 needs to be solved using the optimal machine configuration found in the reformulated problem. This can be achieved by simply adding to the problem in Problem 2 additional constraints that fix the binary variables in `Active`. The resulting MP formulation then has only real-valued variables and becomes a pure Linear Programming (LP) problem, which can be solved efficiently using an exterior point algorithm such as SIMPLEX [31].

The optimal machine configuration was recorded for each sample in preprocessing, but for any value of `AsmQty[a]` it is unclear which of the adjacent samples should be used. We propose using a local search heuristic weighted on the distance to the nearest sample point to select candidate machine configurations. Because the resulting LP formulation can be solved efficiently, this can be done multiple times for a given solution to Problem 4, as shown in Algorithm 4.

Algorithm 4: Heuristic Search

Input: `Active`, `AsmQty` output of Problem 5, *maxRuns*
Output: Solution to Problem 2 (`total_cost`, `Active`, `Qty`)
Method:

Let *bestCost* ← ∞ , *bestActive* ← ∅ , *bestQty* ← ∅
Let *i* ← 1
while *i* ≤ *maxRuns* **do**
    for each Assembly *a* **do**
        if `Active[a]` **then**
            Select sample *s* from s_a using exponential distribution based on sample distance from `AsmQty[a]`
            Add constraints to Problem 2 fixing `Active[m]` for each machine in *a* based on *s*
        else
            Add constraints to Problem 2 fixing `Active[m]` = 0 for each machine in *a*
        end if
    end for
    Solve LP problem in modified Problem 2
    if `total_cost` < *bestCost* **then**
        bestCost ← `total_cost`
        bestActive ← `Active`
        bestQty ← `MachQty`
    end if
    i ← *i* + 1
end while
Return (*bestCost*, *bestActive*, *bestQty*)

Finally, when solving the reformulated problem in Problem 4 it is not necessary to wait until the optimal solution is found. In some instances the reformulated problem will have a significant number of binary variables and solving the MILP to optimality will still take a considerable amount of time. The branch and bound algorithm has the convenient property that it reports the best known feasible solution as it traverses the search space. These best known feasible solutions can be fed into Algorithm 4 as they

are found, producing good feasible solutions to Problem 2. Algorithm 5 shows the full Online-Decomposition Algorithm.

Algorithm 5: Online-Decomposition Algorithm

Input: `Demand`, *maxSubRuns*
Output: Solution to Problem 2 (`total_cost`, `Active`, `Qty`)
Method:

Let *bestCost* $\leftarrow \infty$, *bestActive* $\leftarrow \emptyset$, *bestQty* $\leftarrow \emptyset$
Let *bestRelCost* $\leftarrow \infty$
Let *relCost* $\leftarrow \infty$, *relActive* $\leftarrow \emptyset$, *relQty* $\leftarrow \emptyset$
Start branch and bound algorithm on MILP problem in Problem 4 with `Demand`
while branch and bound not terminated **do**
  Let (*relCost*, *relActive*, *relAsmQty*) $\leftarrow$ Best known solution from branch and bound
  if *relCost* < *bestRelCost* **then**
    bestRelCost $\leftarrow$ *relCost*
    Let (*cost*, *active*, *qty*) $\leftarrow$ HeuristicSearch(*relActive*, *relAsmQty*, *maxSubRuns*)
    if *cost* < *bestCost* **then**
      bestCost $\leftarrow$ *cost*
      bestActive $\leftarrow$ *active*
      bestQty $\leftarrow$ *qty*
    end if
  end if
end while
Let (*cost*, *active*, *qty*) $\leftarrow$ HeuristicSearch(*relActive*, *relAsmQty*, ∞)
if *cost* < *bestCost* **then**
  bestCost $\leftarrow$ *cost*
  bestActive $\leftarrow$ *active*
  bestQty $\leftarrow$ *qty*
end if
Return (*bestCost*, *bestActive*, *bestQty*)

5.6 Experimental Evaluation for Online Speed-Up

In this chapter we showed a technique for reformulating the distributed manufacturing example into an approximate problem with fewer binary variables. In the previous section we present an algorithm that uses this reformulation to solve the original

problem through heuristic search and make the claim that the reduced dimensionality of the reformulated problem offers a significant speed up when compared to solving the original problem with MILP. To test this claim, we generated a family of problems using the DGQL template from Section 5.2 and compared the convergence rate of the best feasible solution when applying both the MILP and the ODA algorithms to the same problem.

These problems were generated based on the following parameters:

- `S` : number of stages
- `P` : number of products per stage
- `A` : number of assemblies per product
- `I` : number of inputs per assembly
- `M` : number of machines per assembly

When solving a problem using pure MILP branch and bound, the size of the search space is dictated entirely by the number of binary variables in the encoding. As we showed in Section 5.3, the number of binary variables is the same as the number of machines. This can be found by:

$$|B| = S * P * A * M$$

When solving the problem using the online-decomposition algorithm (ODA) in Section 5.5, the complexity of the search space is based on two factors. When solving the reformulated MILP problem in Algorithm 5, the output function of each assembly is replaced by a piecewise linear approximation. This includes a binary variable to encode

the case when no machines are operational and the assembly output is zero. Thus the complexity of the reduced MILP problem is measured by the number of assemblies:

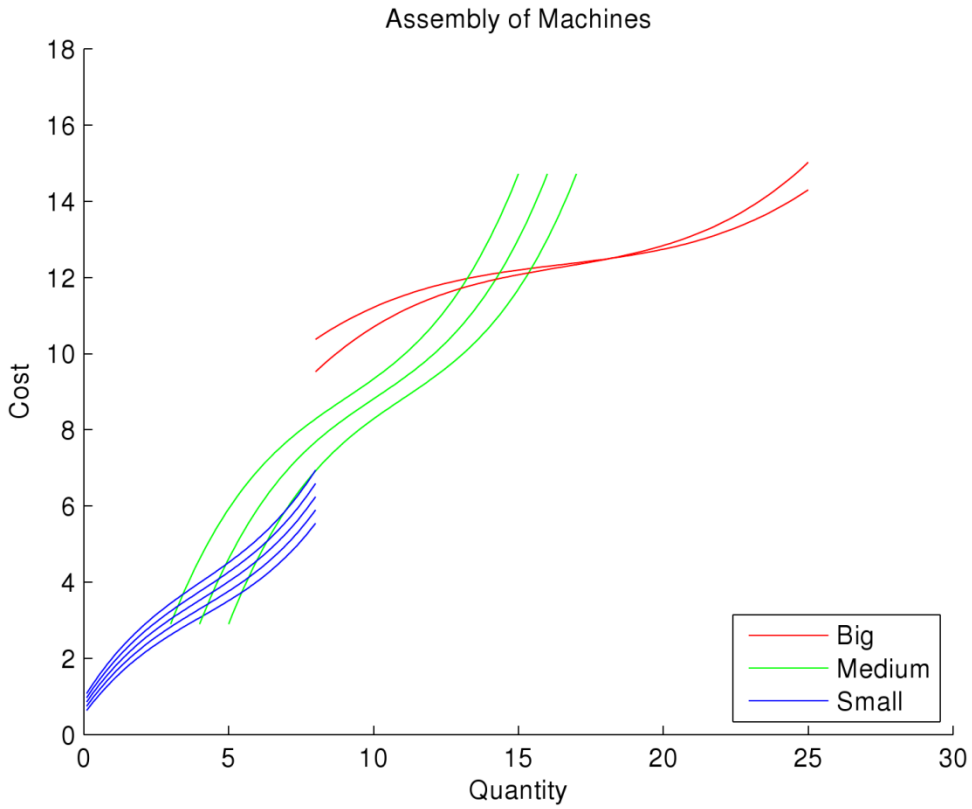

Figure 5: Cost functions of different size machines in one assembly

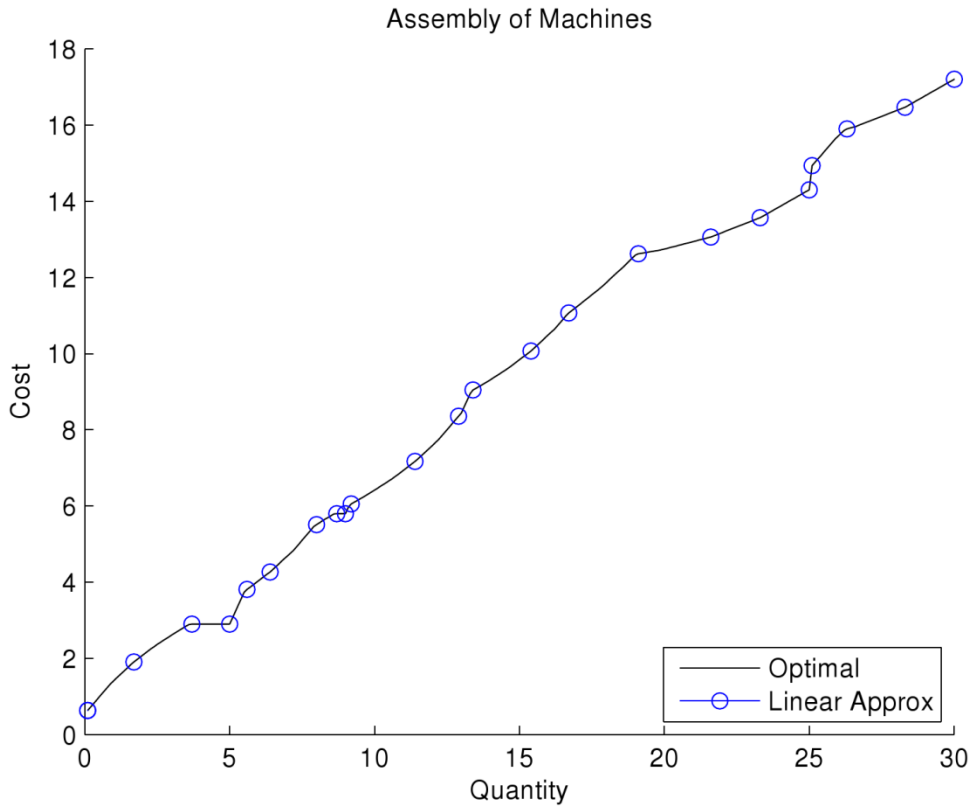

Figure 6: Preprocessed cost function with linear approximation

Once the problem is generated, each assembly of machines is preprocessed to produce a high resolution sampling of the optimal cost function over the assembly output range. For the set of machines shown in Figure 5, this produced a piecewise linear approximation of the assembly cost function that contained 1380 segments, the black line in Figure 6.

When first running the ODA algorithm, we noticed two things. First, considerably more time was spent in the first phase of ODA on problems where the possible quantity range was larger than those where it was smaller, even for the same

number of machines. Even though the branch and bound solver we used had an efficient representation of piecewise linear, the number of segments still impacted performance. Second, when comparing the best solution found by ODA to the actual optimal solution to the problem, there were some instances in which the heuristic search failed to find the optimal configuration of machines, even though it was contained in the pre-computed lookup table. High-resolution sampling of the assembly output range meant that in most neighborhoods around a relaxed solution, all of the configurations were the same. The exponential distribution used in the Heuristic Search makes it extremely unlikely that the correct solution would be found in a reasonable amount of time.

To address both of these issues, a best fit linear approximation of the optimal assembly cost function was created by starting with all sample points where the machine configuration changed, and adding more sample points until a certain error tolerance was reached. These points are shown in blue in Figure 6. In our example, this reduced the number of samples from 1380 to 76 and improved ODA performance both in solution time and accuracy.

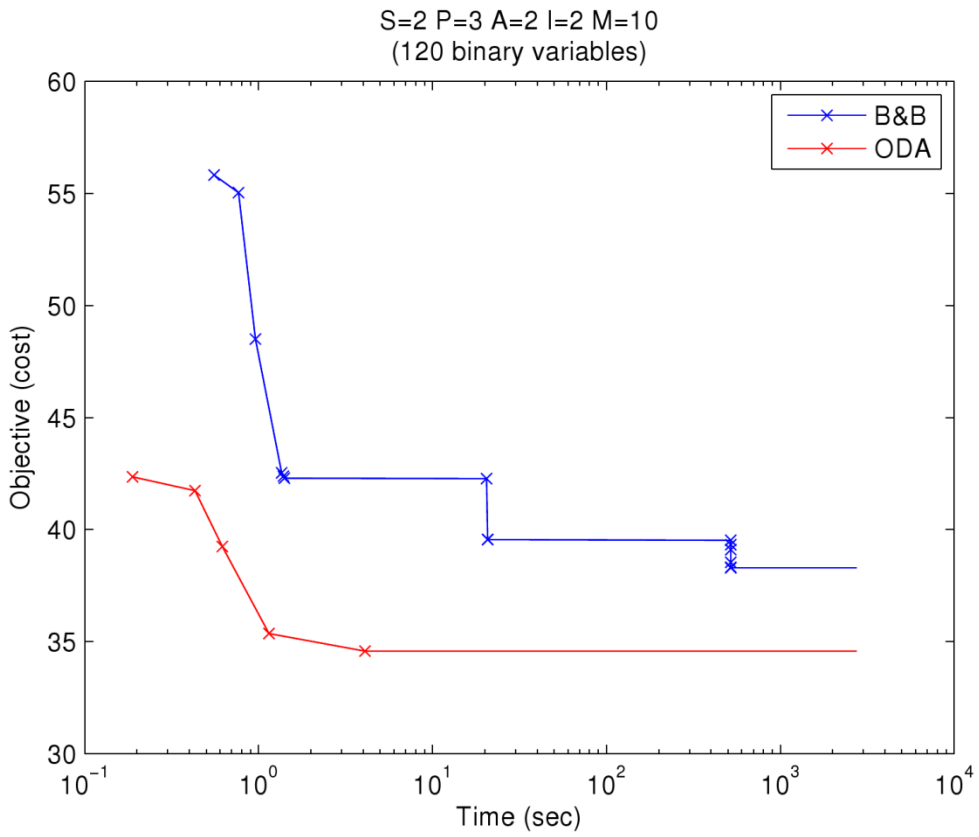

Figure 7: Comparison of optimal solution convergence between B&B and ODA with 120 binary variables

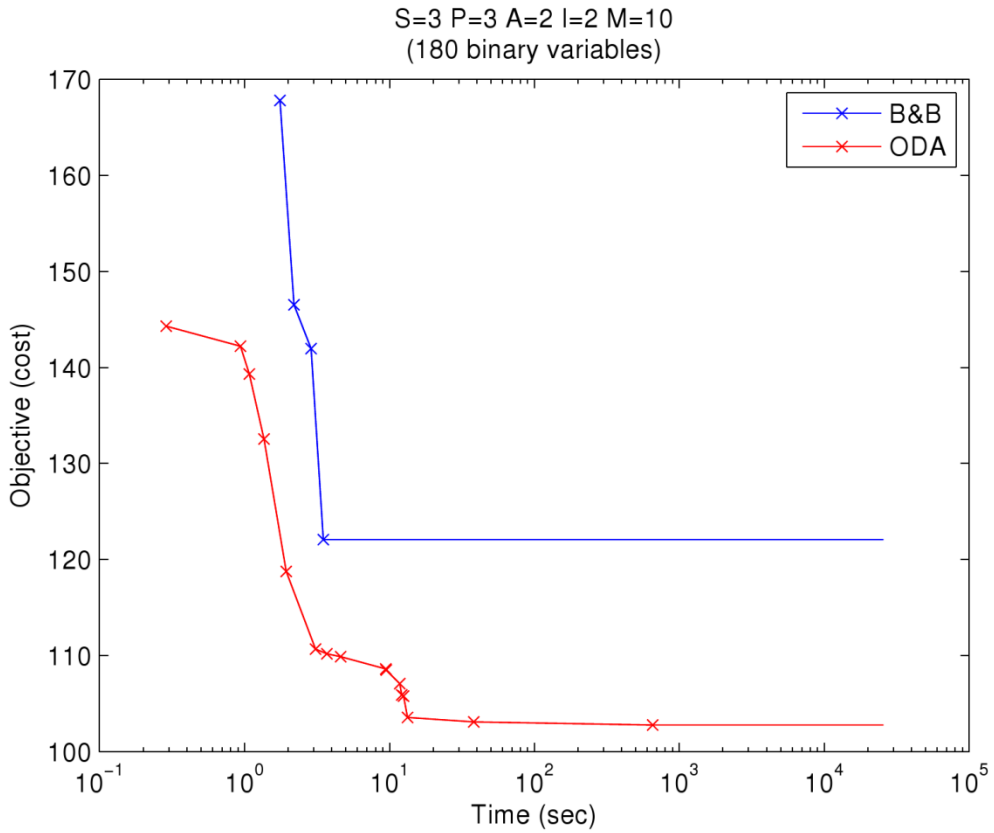

Figure 8: Comparison of optimal solution convergence between B&B and ODA with 180 binary variables

Once a problem was preprocessed, random product demands were generated and both the ODA and branch and bound (B&B) algorithms were run with the same values. The comparison in convergence on the optimal solution for two specific demand vectors can be found in Figure 7 and Figure 8. Points marked with an x indicate the earliest time a new best solution was found for each. Note that for very large problems, like the ones shown here, the B&B algorithm did not terminate in the time allowed and thus never found the improved solutions produced by ODA. Also note that in both cases, the first phase of the ODA algorithm did complete and the remainder of the time was spent in

heuristic search. This was so it would be clear that the graphs represented the best solution found by both algorithms in the time allowed.

6. ADAPTIVE PRE-PROCESSING ALGORITHM

6.1 Efficient Offline Pre-processing Algorithm

In the previous chapter, we used a very fine sampling resolution *r* with Algorithm 7 to produce as accurate a piecewise linear approximation of the cost function $CompCost_a(qty)$ as possible. This approach was chosen because we were not concerned with the computational cost of offline preprocessing, only the online solution time. In practice however, the offline preprocessing may be very costly as components may contain significant combinatorics and for each sample we must solve to optimality the MILP for the component. A key observation is that in many industrial settings, small changes in output (i.e., *qty*) do not result in significant changes in component configuration. In fact, for the multi-stage production experimental study in the previous chapter, although each assembly contained 10 machines and thus 2^{10} possible machine configurations, only 35 optimal machine configurations were observed over the operational range of the component (i.e., $AsmMinQty_a$ to $AsmMaxQty_a$). This suggests we can do significantly better than naive MILP sampling if we can classify the operational range according to machine configuration, and then use a linear program to efficiently fill sample values where we know the combinatorics are fixed.

The technique we adopt is to recursively subdivide the operational range of the component into smaller ranges based on where additional sampling is most likely to

improve the accuracy of the approximated assembly cost function. We would like to develop a search heuristic that orders the subranges of $R = [AsmMinQty_a, AsmMaxQty_a]$ based on which should be subdivided next. Let $R_i = [l_i, u_i]$ be a subrange of R. Intuitively, we would like to give preference to subranges that have the following qualities:

- The cost of $qty \in [l_i, u_i]$ is uncertain, i.e., $Cost(u_i) - Cost(l_i)$ is large
- The configuration of $qty \in [l_i, u_i]$ is uncertain, i.e., $Active(l_i) \neq Active(u_i)$

We observe that when the first quality is true, we always want to subdivide the subrange regardless of whether the second quality holds or not. That is, because we expect changes in $Cost(qty)$ to be smooth, any subrange where $Cost(u_i) - Cost(l_i)$ is large should be subdivided as this will have a proportional improvement on the accuracy of $Cost(qty)$ for $qty \in [l_i, u_i]$. On the other hand, if $Cost(u_i) - Cost(l_i)$ is very small, additional subdivision will not significantly improve the accuracy of $Cost(qty)$.

The second quality suggests that additional sampling is desired whenever range endpoints contain different configurations. Remember that $CompCost_a(qty)$ is just an approximation of the component cost function. In Algorithm 5, the solution to the approximated problem in Problem 4 is used to fix the combinatorics of the components so that a much faster linear program (LP) can be used to find a solution to the original problem. As shown in the previous chapter, when the solution to the approximated problem is correct (i.e., the values of the interface variables in the approximated solution are the same as a solution to the original problem), then the problem can be decomposed

into components and solved precisely. This means that the Online-Decomposition Algorithm can tolerate small inaccuracies in component output, $CompCost_a(qty)$, so long as the corresponding machine configuration is correct. This gives a natural preference to split subranges when the configurations are different.

Note that the degree to which we prefer to split subranges when the configurations are different over splitting those that are the same is a tradeoff between more accurately classifying *R* into partitions based on the known configurations and searching for potentially novel configurations that have yet to be encountered. This tradeoff can be parameterized and suggests the heuristic given in Algorithm 6.

Algorithm 6: Subdivision Heuristic

Input: Samples *l* and *u* of subrange *R*, tradeoff parameter *T*
Output: Heuristic on how likely subdividing R will yield improved accuracy, smaller values indicate more likely
Method:

1. Let $\Delta Cost \leftarrow Cost(u) - Cost(l)$
2. **if** $Active(u) \neq Active(l)$ **then**
3. Return $(1 / \Delta Cost)$
4. **else**
5. Return $(T / \Delta Cost)$
6. **end if**

The subdivision technique proposed has the added quality of being greedy. That is, given an additional unit of computation budget it is always better to split the next subrange as specified by the Subdivision Heuristic. This allows us to adaptively generate a sampling of the component cost function with a convenient stopping condition based on a fixed MILP budget B. See Algorithm 7.

Algorithm 7: Adaptive Pre-processing Algorithm

Input: Assembly *a*, tradeoff *T*, budget *M*, sampling resolution *r*
Output: List of samples (*qty*, *cost*, *Active*)
Method:

1. Let $s_{min} \leftarrow$ solution to Problem 5 for *a* with $\texttt{Quantity} = AsmMinQty_a$
2. Let $s_{max} \leftarrow$ solution to Problem 5 for *a* with $\texttt{Quantity} = AsmMaxQty_a$
3. Let $S = \{s_{min}, s_{max}\}$
4. Let *H* be priority queue of subranges based on the heuristic from Algorithm 6 with *T*
5. Add subrange (s_{min}, s_{max}) to *H*
6. Let $b \leftarrow 1$
7. **while** $b \leq B$ **do**
8. Let $R \leftarrow$ subrange in *H* with lowest Subdivision Heuristic
9. Let $l \leftarrow Lower(R)$ and $u \leftarrow Upper(R)$
10. Let $qty_{mid} \leftarrow {Qty(u) + Qty(l)} \big/ 2$
11. Let $s_{mid} \leftarrow$ solution to Problem 5 for *a* with $\texttt{Quantity} = qty_{mid}$
12. Insert s_{mid} between *l* and *u* in *S*
13. Add subranges (l, s_{mid}) and (s_{mid}, u) to *H*
14. Let $b \leftarrow b + 1$
15. **end while**
16. Let $last \leftarrow 1$
17. **for** $i = 2 \rightarrow |S|$ **do**
18. **if** $Active(S_i) \neq Active(S_{last}) \wedge i - 1 \neq last$ **then**
19. Add constraints to Problem 2 fixing $\texttt{Active}[m]$ for each machine in *a* based on S_{last}
20. **for** $qty = Qty(S_{last}) \rightarrow Qty(S_{i-1})$ **by** *r* **do**
21. Solve LP problem in modified Problem 2 for *qty* adding s_{qty} to *S*
22. **end for**
23. Let $last \leftarrow i$
24. **end if**
25. **end for**
26. Return *S*

Given a fixed computation budget *B* we can produce sampling *S* using Algorithm 7 offline. These samples can then be analyzed to create a piecewise linear approximation of the component cost $CompCost_a$ for use in the online-decomposition algorithm. While it is possible to simply use all of the samples in *S* as the piecewise linear function in $CompCost_a$, this is usually undesirable as commercial solvers typically represent these

using costly SOS2 constraints. Rather, we would like to use the minimum subset of points in *S* to create a piecewise linear approximation up to some error tolerance. In general this requires solving another mixed integer program [48]. Many non-optimal algorithms based on approximations exist and a discussion of which is best in the context of this problem is outside the scope of this dissertation.

6.2 Experimental Evaluation for Adaptive Speed-Up

In the previous section we propose the efficient offline Adaptive Pre-processing Algorithm as an improvement over the high resolution scanning algorithm presented in the previous chapter. The motivation behind this new algorithm was the observation that although the number of potential combinatorial choices for a given component is exponential, in practice only a small subset of configurations are used to achieve optimal production cost.

One concern we had was that the proposed algorithm would not be as sensitive as the high resolution scan, and would fail to capture subtle features of the optimal component objective function. In order to understand the tradeoff between computation budget and accuracy of the resulting piecewise linear approximation, we used the same 10 machine assembly configuration from the Multi-Stage production example in previous work and varied the MILP sampling budget. The results of this experiment can be found in Figure 9. In each of the four graphs, the black lines represent ranges of the cost function where samples on both extremes (and all points in-between) have the same machine configuration. The thicker red lines represent ranges where the samples on the

ends have different machine configurations and no additional samples were taken between the two. Finally, the blue circles show the sample points that were used in constructing a best-fit piecewise linear approximation of the sampled cost function.

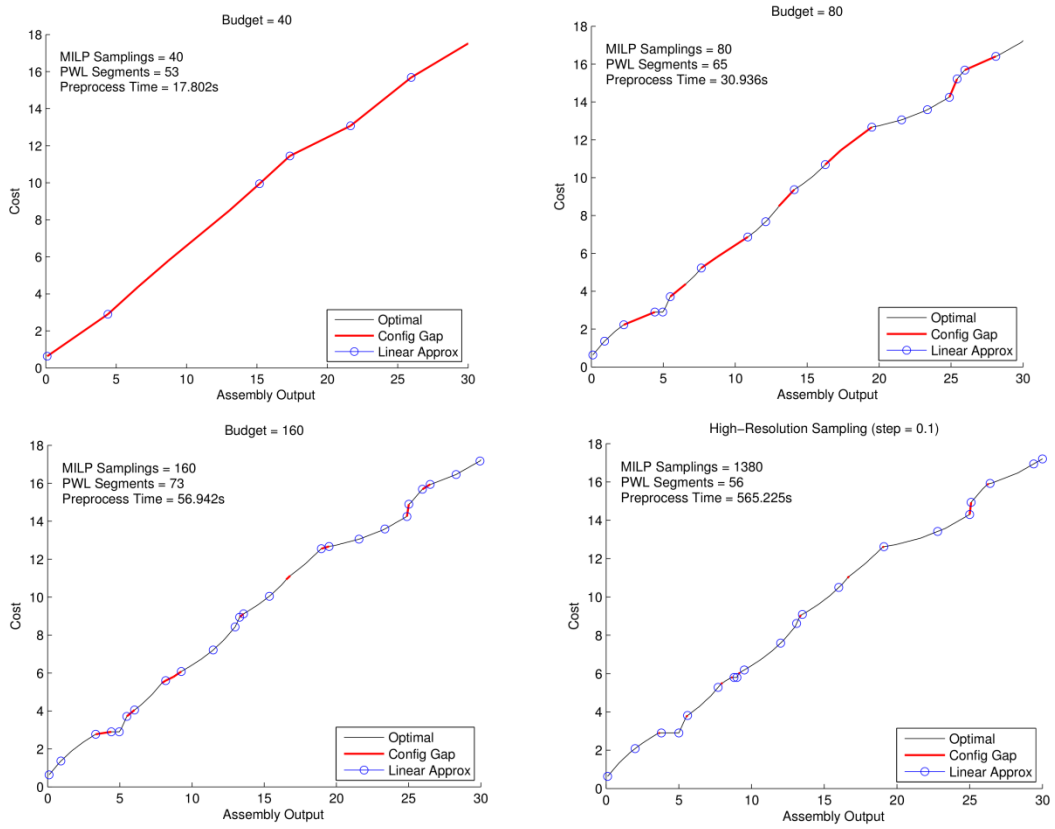

Figure 9: Comparison of approximate piecewise linear cost functions using different preprocessing budget allocations (where T=5 in Subdivisions Heuristic, Algorithm 6) to a high-resolution scan

Note that in some cases the number of MILP samples is lower than the number of piecewise linear (PWL) segments. This is because whenever the algorithm detects a "black" range where the sampled machine configuration is the same on both ends, it switches to an LP formulation to efficiently sample the points in-between. Furthermore, because the best-fit piecewise linear graph is only an approximation of the sample points, in places where sampling provides a large number of optimal cost function features, a

consistent number of PWL segments is produced. This is a convenient property as it provides a consistent time-bound for the approximated Problem 4.

As is shown, the piecewise linear approximation produced with a budget of 160 MILP samples is nearly identical to the one produced using a high-resolution scan that contained 1380 MILP samples. More to the point, the adaptive preprocessing algorithm was able to achieve this with an order of magnitude improvement in preprocessing time.

Of greater concern was the impact on overall solution accuracy with ODA when using the adaptive preprocessing algorithm. Here we were chiefly looking to see that the adaptive algorithm, while creating a close approximation of the high-resolution scan, did not fail to detect a component configuration which would negatively impact the heuristic search subroutine of ODA. It could be that when the combinatorics are fixed, running the full optimization problem finds some novel settings that would produce a much improved objective.

To show that this is not the case, we ran the online-decomposition algorithm using the approximated cost functions found in the first experiment. In each case, we varied the tradeoff parameter T to see what impact it had on the optimal cost and solution accuracy. The results of this experiment can be found in Figure 10. The larger values of T prefer exploration for new configurations over refining known configurations, and the results in Figure 10 appear to agree. The larger the value of T, the more quickly an optimal solution is found. However, in our experiment the optimal cost function only contains 35 different configuration ranges. Thus having a sample budget of 40 or greater means that all configurations are most likely discovered.

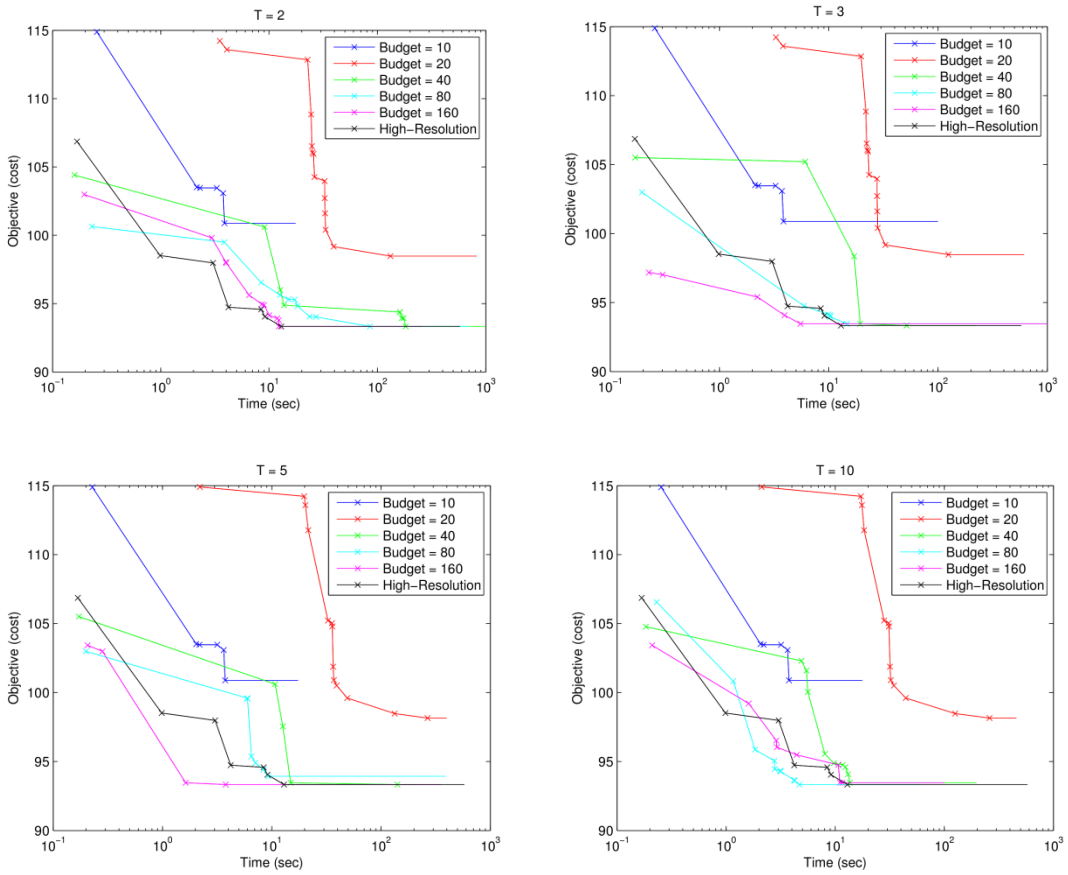

Figure 10: Comparison of approximate piecewise linear cost functions using different tradeoff weights *T*

Future work will consider automatically determining an optimal value of *T*, given the number of expected configurations (configuration density), the overall range of the component cost function, and the sampling resolution *r*.

7. PROOF OF CONCEPT PROTOTYPE.

A research prototype was created as part of this dissertation to demonstrate the feasibility of extending the database relational model with modern decision optimization technologies. One of the early design goals was to keep the user interface (UI) and user experience (UX) as close as possible to the existing contract between database developers and the tools they use. This meant restricting the commands a database developer could issue to the DGQL system to only valid SQL commands. Figure 11 shows how any user can interact with the DGQL backend by issuing commands through a standard database client interface.

Several technology choices were made to facilitate the rapid development and iteration of the research prototype. The open source database management system PostgreSQL [49] has a large developer community that contributes many third-party extensions. DGQL was developed within the pl/Java extension [50] as the language allowed for a uniform interface to both the underlying database metadata and a wide variety of industrial-strength solvers. In addition, developing high availability software inside a managed environment like Java meant that the DGQL prototype would be free from memory leaks and generally easy to debug and test.

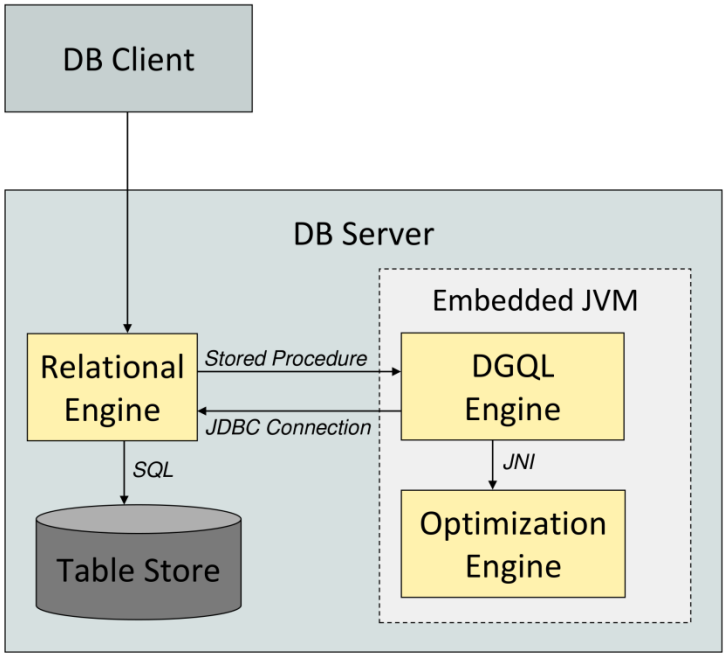

Figure 11: High-level sequence diagram describing client interaction with DGQL backend

The set of commands available to DGQL was also designed to be the minimal set of new information needed by the system to describe an optimization problem. These new commands were implemented as stored procedures which would invoke native code on the database server. The commands added for DGQL fall into three categories (1) commands that annotate the existing relational model with optimization metadata, e.g., variables and constraints, (2) commands which indicate that optimization should be invoked atomically based on the current database state, and (3) commands that set tuning parameters to be by the underlying DGQL engine and the specific optimization software that is invoked, e.g., integer tolerances, branch and bound timeouts, optimality gap, etc. The full API reference for these commands can be found in Table 2.

Table 2: List of implemented DGQL commands in research prototype

Annotation Commands
`dgql.add_variable(schema,view,column)` Indicate that *column* of *schema.view* should be instantiated by the system
`dgql.add_variable(schema,view,column,min,max)` Indicate that *column* of *schema.view* should be instantiated by the system and restricted to values between *min* and *max* inclusive.
`dgql.add_constraint(schema,view,expression)` Indicate that *expression* should evaluate to true for every row of *schema.view*

Optimization Commands
`dgql.minimize(schema,view)` Indicate that all variables in the database should be instantiated such that objective *schema.view* is minimized subject to all existing constraints
`dgql.maximize(schema,view)` Indicate that all variables in the database should be instantiated such that objective *schema.view* is maximized subject to all existing constraints

Parameter Commands
`dgql.set_option(schema,view,key,value)` Set *key* = *value* when optimizing the objective *schema.view*
`dgql.get_option(schema,view,key)` Return the value *key* of used when optimizing the objective *schema.view*

When an optimization command is invoked, the DGQL backend constructs a computation tree for the specified *schema.view* inside the current database transaction. This prevents another client request from inadvertently modifying the computation model described by the DGQL variables or constraints, or the relational tables that they reference. The computation tree is then analyzed by the DGQL backend to determine the type of optimization problem described by the root node for the objective *schema.view*. If the objective is linear in columns that have been annotated as DGQL variables, then the optimization falls into the class of linear programming (LP) problems. If those columns

are all over just integer domains, then the optimization is an integer linear programming (ILP) problem. If some annotated columns are integer and others are over continuous domains, then it is a mixed-integer linear programming (MILP) problem. The type of optimization problem indicates which underlying optimization tool should be selected.

The current DGQL research prototype only supports two optimization technologies: ILOG CPLEX Solver [51] and ILOG CP Optimizer [52]. ILOG CPLEX Solver is selected when the optimization problem detected is a linear program (LP) or a quadratic program (QP) and contains variables that are either real valued, integer valued, or both, i.e., mixed-integer. Optimization problems in this class can be used to model a broad set of important industrial problems, e.g., supply chain logistics, market price discovery, optimal investment allocation, etc. and decades of research have gone into fine tuning the Branch & Bound [33] and Branch, Price & Cut [53] algorithms that solve them.

ILOG CP Optimizer is selected when the optimization problem contains variables that are only integer valued and the objective is non-linear. This class of problems is also of industrial importance and is most often associated with optimal scheduling, bin packing, magic square and other assignment problems. Problems in this class are often solved with using extra domain-specific information, e.g., callbacks that supply user-defined variable weighting functions. However, ILOG CP Optimizer is based on a general purpose search strategy for constraint programming (CP) that measures the impact certain variables have on reducing the size of the search space and gives them higher priority during solution exploration [54]. This makes it well suited for use with

DGQL where we do not expect the user to be able to provide additional information and many of the variables are introduced programmatically during DGQL reduction.

This leaves an important class of problems outside the reaches of this research prototype. When the DGQL backend encounters an optimization problem that is non-linear with real valued or mixed-integer variables it returns an error. Although many good industrial solvers exist for NLP problems (e.g., KNITRO [55], MINOS [56] and SNOPT [57]) and MINLP problems (e.g., DICOPT [58] and SCIP [59]), integration with these solvers is outside the scope of this dissertation and is left as future work.

8. CONCLUSIONS AND FUTURE RESEARCH

The Decision Guidance Query Language (DGQL) is designed to (re-)use SQL programs for decision optimization with the goals of making DSGS implementation agile and intuitive and leveraging existing investment in SQL-implemented information systems. This dissertation has defined its formal syntax and semantics and provided a canonical implementation of DGQL by reduction to mathematical programming. Additionally, through experimental evaluation, we have shown that the overhead of using a high-level language is small when compared with manually-crafted MP formulations. With added benefit of agile and intuitive high-level language, DGQL provides a practical solution for decision optimization in DSGS, especially when an underlying database is involved and database applications already exist.

The online-decomposition algorithm (ODA) developed and based on offline preprocessing is broadly applicable to a class of distributed manufacturing network problems. This algorithm provides an order of magnitude improvement in both solution time and the quality of solution found when compared to branch and bound alone. The adaptive pre-processing algorithm (APA) shows that it is possible to introduce a computation bound on pre-processing so that the benefit of ODA can be applied to optimization problems starting from a "cold-start". Moreover, the ability to adaptively enhance the pre-processed subproblem approximation allows systems to iteratively

improve in DGQL solution time the more frequently the static portion of the optimization problem is queried.

Many research questions remain open. In the language design space, extending DGQL semantics to recognize object relational mappings (ORM) and service definition, abstraction and composition is a natural next step. Adding new annotations that indicate the boundary between static and dynamic program components would allow the DGQL implementation to automatically identify and extract subproblems for use with the online decomposition and adaptive pre-processing algorithms.

The DGQL relational algebra precisely defines the semantics of table and view based optimization problems. Many well understood operations research techniques for reducing problem complexity involve identifying patterns in MP formulations and transforming the problem definition into an alternate yet equivalent set of equations. An interesting question is how to exploit the additional structure in the DGQL relational algebra to define query transformations that can automatically take advantage of these techniques.

Finally, DGQL has been shown to be a viable technology to solve many research and development optimization problems, as evidenced by the broad set of application domains where it has been successfully applied [60] [61] [62] [63] [64] [65]. However, the nascent eco-system around DGQL presents a challenge to large scale industrial adoption. Additional tools are needed to aid DGQL development akin to those that exist for DBMS design, software life-cycle management, regression testing, instrumentation and monitoring, etc. as well as tools specific to mathematical programming e.g., detecting

irreducible infeasible sets (IIS), introducing slack variables and relaxations, adding heuristics/callbacks to guide problem space exploration, etc. Nevertheless, even in its current form DGQL delivers on the promise of simplifying the development and deployment of Decision Support and Guidance Systems.

APPENDIX: RESULTING PUBLICATIONS

The work conducted during doctoral study resulted in 15 peer reviewed and accepted publications. Of these 11 were presented at conferences and 4 were published in academic journals. They can be partitioned into papers that describe the core DGQL system and algorithms, and application or related papers where DGQL was used in an innovative technical solution.

Publications Directly Related to Dissertation

N. Egge, A. Brodsky, I. Griva, An Efficient Preprocessing Algorithm to Speed-Up Multistage Production Decision Optimization Problems, *In Proceedings of the 46th Annual Hawaii International Conference on System Sciences* (HICSS-46), Maui, Hawaii: Jan. 2013

N. Egge, A. Brodsky, I. Griva, Optimizing Distributed Manufacturing Networks via Offline Preprocessing of Nodes in Decision Guidance Query Language, *International Journal of Decision Support System Technology* (IJDSST 2012), Vol. 4, No. 3, (2012), pp 25-42

A. Brodsky, N. Egge, X. S. Wang, Supporting Agile Organizations with a Decision Guidance Query Language, *Journal of Management Information Systems* (JMIS 2012), Vol. 28, No. 4, (Spring 2012), pp 39-68

N. Egge, A. Brodsky, I. Griva, Online Optimization through Preprocessing for Multi-Stage Production Decision Guidance Queries, *International Conference on Data Engineering Workshop on Data-Driven Decision Guidance and Support Systems* (ICDE DGSS 2012), Washington D.C., USA: Apr. 2012

N. Egge, A. Brodsky, I. Griva, Toward Online Optimization Through Preprocessing and Decomposition in Decision Guidance Query Language. *EURO Working Group on Decision Support Systems* (EWG-DSS 2011), Paris, France: Nov. 2011

A. Brodsky, N. Egge, and X. S. Wang, Reusing Relational Queries for Intuitive Decision Optimization, *In Proceedings of the 44th Annual Hawaii International Conference on System Sciences (HICSS-44)*, Kauai, Hawaii: Jan. 2011

A. Brodsky, N. Egge, and X. S. Wang, Decision Guidance Query Language and System, *In Proceedings of the 15th IFIP WG 8.3 International Conference on Decision Support Systems (DSS 2010)* Lisbon, Portugal: Jul. 2010

A. Brodsky, M. Bhot, M. Chandrashekar, N. Egge, X. S. Wang, A decisions query language (DQL): high-level abstraction for mathematical programming over databases. *In Proceedings of the International Conference on Management of Data and 28th Symposium on Principles of Database Systems* (PODS 2009), Providence, Rhode Island: Jun. 2009

Other Publications

C. K. Ngan, A. Brodsky, N. Egge, E. Backus, A Decision-Guided Energy Framework for Optimal Power, Heating, and Cooling Capacity Investment, *15th International Conference on Enterprise Information Systems (ICEIS 2013*), Angers, France: Jul. 2013

B. Goodhart, V. Yerneni, A. Brodsky, V. Rudraraju, N. Egge, SmartCart: A Consolidated Shopping Cart for Pareto-Optimal Sourcing and Fair Discount Distribution, *International Conference on Data Engineering Workshop on Data-Driven Decision Guidance and Support Systems (ICDE DGSS 2013*), Brisbane, Australia: Apr. 2013

A. Motro, A. Brodsky, N. Egge, A. D'Atri, Optimizing Procurement Decisions in Networked Virtual Enterprises, *International Journal of Decision Support System Technology* (IJDSST 2012), Vol. 4, No. 3, (2012), pp 43-67

G. Shao, D. Kibira, A. Brodsky, N. Egge, Decision Support for Sustainable Manufacturing using Decision Guidance Query Language, *International Journal of Sustainable Engineering*, Vol. 4, No. 3, (July 2011), pp 251-265

A. Brodsky, S.C. Mana, M. Awad, N. Egge, A Decision-Guided Advisor to Maximize ROI in Local Generation & Utility Contracts. *In Proceedings of Innovative Smart Grid Technologies* (ISGT 2011), Anaheim, California: Jan. 2011

A. S. Alrazgan, A. Brodsky, A. Nagarajan, and N. Egge, Learning Occupancy Prediction Models with Decision-Guidance Query Language. *In Proceedings of the 44th Annual Hawaii International Conference on System Sciences* (HICSS 2011), Jan. 2011

S. Farley, A. Brodsky, J. McDowall, and N. Egge, SimQL: Simulation-Based Decision Modeling Over Stochastic Databases, *In Proceedings of the 15th IFIP WG 8.3 International Conference on Decision Support Systems* (DSS 2010), Jul. 2010

REFERENCES

[1] Alexander Brodsky and X. Sean Wang, "Decision-guidance management systems (DGMS): Seamless integration of data acquisition. learning. prediction and optimization," in *HICSS*, 2008.

[2] Robert Fourer, David M. Gay, and Brian W. Kernighan, "A Modeling Language for Mathematical Programming," *Management Science*, vol. 36, pp. 519-554, 1990.

[3] Johannes Bisschop and Alexander Meeraus, "On the Development of a General Algebraic Modeling System in a Strategic Planning Environment," *Applications: Mathematical Programming Studies*, vol. 20, pp. 1-29, 1982.

[4] Yuan-Chi Chang et al., "The onion technique: Indexing for linear optimization queries," in *SIGMOD Conference*, 2000, pp. 391-402.

[5] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou, "Prefer: A system for the efficient execution of multi-parametric ranked queries," in *SIGMOD Conference*, 2001, pp. 259-270.

[6] Zhen Zhang et al., "Boolean + Ranking: Querying a Database by K-Constrained Optimization," in *SIGMOD Conference*, 2006, pp. 359-370.

[7] Michael Gibas, Ning Zheng, and Hakan Ferhatosmanoglu, "A general framework for modeling and procession optimization queries," in *VLDB*, 2007, pp. 1069-1080.

[8] Ihad F. Ilyas, George Beskales, and Mohamed A. Soliman, "A survey of top-k query processing techniqes in relation database systems," *ACM Computing Suveys*, vol. 40, no. 4, October 2008.

[9] Marc Hoppe, *Inventory Optimization with SAP*, 2nd ed. Boston: Galileo Press, 2008.

[10] Ronald F. Boisvert, Sally E. Howe, and David K. Kahaner, "Gams: a framework for the management of scientific software," *ACM Transactions of Mathematical Software*, vol. 11, no. 4, pp. 313-355, December 1985.

[11] Donald Chamberlin and Raymond Boyce, "SEQUEL: A structured English query language," in *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, Ann Arbor, Michigan, 1974, pp. 249-264.

[12] Christian Bauer and Gavin King, *Java Persistence with Hibernate*. Greenwich, CT: Manning Publications, 2006.

[13] J. P. Shim et al., "Past, present, and future of decision support technology," *Decision Support Systems*, vol. 33, no. 2, pp. 111-126, 2002.

[14] Robert Fourer, David M. Gay, and Brian Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, 2nd ed.: Duxbury Press, 2002.

[15] Marcel Roelofs and Johannes Bisschop, *AIMMS User's Guide*. Bellevue, WA: Paragon Decision Technology, 2010.

[16] Richard E. Rosenthal, *GAMS - A User's Guide*. Washington, DC, USA: GAMS Development Corporation, 2014.

[17] Linus Schrage, *Optimization Modeling with LINGO*, 5th ed. Chicago: LINDO Systems Inc, 1999.

[18] Pascal Van Hentenryck, *The OPL Optimization Programming Language*. Cambridge, MA: MIT Press, 1999.

[19] A Atamtürk, E. L. Johnson, J. T. Linderoth, and M. W.P. Savelsbergh, "A relational modeling system for linear and integer programming," *Operations Research*, vol. 48, no. 6, pp. 846-857, November - December 2000.

[20] H. K. Bhargava, R. Kirshnan, and S. Murkherjee, "On the integration of data and mathematical modeling languages," *Annals of Operations Research*, vol. 38, no. 1, pp. 69-95, November - December 1992.

[21] J. Choobineh, "SQLMP: A data sublanguage for representation and formulation of linear mathematical models," *INFORMS Journal on Computing*, vol. 3, no. 4, pp. 358-375, Fall 1991.

[22] D. R. Dolk, "An introduction to model integration and integrated modeling environments," *Decision Support Systems*, vol. 10, no. 3, pp. 249-254, October 1993.

[23] G. Mitra, C. Lucas, S. Moody, and B. Kristjansson, "Sets and indices in linear programming modeling and their integration with relational data models," *Computationals Optimization and Applications*, vol. 3, no. 4, pp. 263-283, July 1995.

[24] Peter Fritzson and Vadim Engelson, "Modelica - a unified object-oriented language for system modeling and simulation," in *ECOOP'98 - Object-Oriented Programming, 12th European Conference*, vol. 1445, Brussels, Belgium, 1998, pp. 67-90.

[25] Alexander Brodsky and Hadon Nash, "CoJava: Optimization modeling by nondeterministic simulation," in *CP'06 Proceedings of the 12th international conference on Principles and Practice of Constraint Programming*, Nantes, France, 2006, pp. 91-106.

[26] Patrick Viry, *Object-Oriented Modeling with OptimJ*. Strasbourg, France: Ateji, 2008.

[27] Moshé M. Zloof, "Query by Example," in *Proceedings of the AFIPS National Computer Conference*, Anaheim, CA, 1975, pp. 431-438.

[28] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database Systems: The Complete Book*, 2nd ed. Englewood Cliffs, N.J.: Prentice Hall, 2008.

[29] G. Özsoyoğlu, Z. M. Özsoyoğlu, and V. Matos, "Extending relational algebra and relational calculus with set-valued attributes and aggregate functions," *ACM Transactions on Database Systems*, vol. 12, no. 4, pp. 566-592, December 1987.

[30] L. G. Kachiyan, "A polynomial algorithm in linear programming," *Soviet Mathematics Doklady*, vol. 20, pp. 191-194, 1979.

[31] G. Dantzig, *Linear Programming and Extensions*. Princeton, N.J.: Princeton University Press, 1963.

[32] A. Shrijver, *Theory of Linear and Integer Programming*. New York, N.Y.: John Wiley & Sons, 1998.

[33] A. H. Land and A. G. Doig, "An Automatic Method of Solving Discrete Programming," *Econometrica*, vol. 28, no. 3, pp. 497-520, 1960.

[34] J. A. Tomlin, "Special Ordered Sets and an Application to Gas Supply Operations Planning," *Mathematical Programming*, vol. 42, no. 1, pp. 69-84, 1988.

[35] J. Li, D. E. Blumenfeld, N. Huang, and J. M. Alden, "Throughput analysis of production systems: recent advances and future topics," *International Journal of Production Research*, vol. 47, no. 14, pp. 3823-3851, July 2009.

[36] M. Manitz, "Queueing-model based analysis of assembly lines with finite buffers and general service times," *Computers & Operations Research*, vol. 35, no. 8, pp. 2520-2536, August 2008.

[37] M. Jahangirian, T. Eldabi, A. Naseer, L. K. Stergioulas, and T. Young, "Simulation in manufacturing and business: A review," *European Journal of Operational Research*, vol. 203, no. 1, pp. 1-13, May 2010.

[38] I. Grossmann, M. Erdirik-Dogan, and R. Karrupiah, "Overview of planning and scheduling for enterprise-wide optimization of process industries," *At-Automatisierungstechnik*, vol. 56, no. 2, pp. 64-79, January 2008.

[39] Z. A. Banaszak, M. B. Zaremba, and W. Muszynski, "Constraint programming for project-driven manufacturing," *International Journal of Production Economics*, vol. 120, no. 2, pp. 463-475, August 2009.

[40] Shu-Shun Liu and Chang-Jung Wang, "Optimizing project selection and scheduling problems with time-dependent resource constraints," *Automation in Construction*, vol. 20, no. 8, pp. 1110-1119, December 2011.

[41] A. Alonso-Ayuso, L. F. Escudero, and M. T. Ortuño, "On modelling planning under uncertainty in manufacturing," *SORT-Statistics and Operations Research Transactions*, vol. 31, no. 2, pp. 109-150, July-December 2007.

[42] N. Sahindis, "Optimizing under uncertainty: state-of-the-art and opportunities," *Computers & Chemical Engineering*, vol. 28, no. 6-7, pp. 971-983, June 2004.

[43] N. Li, Z. Jiang, L. Zheng, Z. Zhang, and C. Zhuang, "An overlapping decomposition method for performance analysis of the two-loop closed production system in semiconductor assembly factory," *International Journal of Computer Integrated Manufacturing*, vol. 24, no. 9, pp. 811-820, July 2011.

[44] S. D. Pittmann, B. B. Bare, and D. G. Briggs, "Hierarchical production planning in forestry using price-directed decomposition," *Canadian Journal of Forest Research* , vol. 37, no. 10, pp. 2010-2021, October 2007.

[45] R. Nystrom, R. Franke, I. Harjunkoski, and A. Kroll, "Production campaign planning including grade transition sequencing and dynamic optimization," *Computers & Chemical Engineering*, vol. 29, no. 10, pp. 2163-2179, September 2005.

[46] R. Nystrom, I. Harjunkoski, and A. Kroll, "Production optimization for continuously operated processes with optimal operation and scheduling of multiple units," *Computers & Chemical Engineering*, vol. 30, no. 3, pp. 392-406, January 2006.

[47] AMPL Optimization. (2009) AMPL. [Online]. http://ampl.com/

[48] A. Toriello and J. P. Vielma, "Fitting piecewise linear continuous functions," *European Journal of Operational Research*, vol. 219, no. 1, pp. 86-95, January 2012.

[49] The PostgreSQL Global Development Group. (2014) PostgreSQL. [Online]. http://www.postgresql.org/

[50] Tada AB. (2009) PL/Java. [Online]. http://pgfoundry.org/projects/pljava/

[51] IBM ILOG. (2009) CPLEX. [Online]. www.ibm.com/software/commerce/optimization/cplex-optimizer/

[52] IBM ILOG. (2009) CP Optimizer. [Online]. www.ibm.com/software/commerce/optimization/cplex-cp-optimizer/

[53] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. Savelsbergh, and P. H. Vance, "Branch-and-price: column generation for solving huge integer programs," *Operations Research*, vol. 46, no. 3, pp. 316-329, 1998.

[54] Phillipe Refalo, "Impact-based search strategies for constraint programming," in *Principles and Practice of Constraint Programming – CP 2004*, vol. 3258, Toronto, Canada, 2004, pp. 557-571.

[55] Ziena Optimization LLC. (2014) KNITRO. [Online]. http://www.ziena.com/knitro.htm

[56] B. A. Murtagh and M. A. Saunders, "MINOS 5.5 User's Guide," Systems Optimization Laboratory, Department of Operations Research, Stanford University, SOL 83-20R 1998.

[57] Philip E. Gill, Walter Murray, and Michael A. Saunders, "SNOPT: An SQP algorithm for large-scale constrained optimization," *SIAM Review*, vol. 47, no. 1, pp. 99-131, 2005.

[58] J. Viswanathan and I. E. Grossman, "A Combined Penalty Function and Outer Approximation Method for MINLP Optimization," *Computers and Chemical Engineering*, vol. 14, no. 7, pp. 769-782, 1990.

[59] Zuse Institute Berlin. (2014) SCIP Optimization Suite. [Online]. http://scip.zib.de/

[60] A. Alrazgan, A. Nagarajan, A. Brodsky, and N. Egge, "Learning Occupancy Prediction Models with Decision-Guidance Query Language," in *Proceedings of the 44th Annual Hawaii International Conference on System Sciences*, 2011, pp. 1-10.

[61] A. Brodsky, S. C. Mana, M. Awad, and N. Egge, "A Decision-Guided Advisor to Maximize ROI in Local Generation & Utility Contracts," in *Proceedings of Innovative Smart Grid Technologies*, Anaheim, 2011, pp. 1-7.

[62] G. Shao, D. Kibira, A. Brodsky, and N. Egge, "Decision Support for Sustainable Manufacturing using Decision Guidance Query Language," *International Journal of Sustainable Engineering*, vol. 4, no. 3, pp. 251-265, July 2011.

[63] A. Motro, A. Brodsky, N. Egge, and A. D'Atri, "Optimizing Procurement Decisions in Networked Virtual Enterprises," *International Journal of Decision Support System Technology*, vol. 4, no. 3, pp. 43-67, 2012.

[64] B. Goodhart, V. Yerneni, A. Brodsky, V. Rudraraju, and N. Egge, "SmartCart: A Consolidated Shopping Cart for Pareto-Optimal Sourcing and Fair Discount Distribution," in *Proceedings of the International Conference on Data Engineering Workshop on Data-Driven Decision Guidance and Support Systems*, Brisbane, Australia, 2013.

[65] C. K. Ngan, A. Brodsky, N. Egge, and E. Backus, "A Decision-Guided Energy Framework for Optimal Power, Heating, and Cooling Capacity Investment," in *Proceedings of the 15th International Conference on Enterprise Information Systems*, Angers, France, 2013, pp. 357-369.

BIOGRAPHY

Nathan E. Egge graduated from T.C. Williams High School, Alexandria, Virginia, in 1997. He received his Bachelor of Science from Virginia Tech in 1999 and his Master of Science from Virginia Tech in 2001. He has over a decade of industry experience in enterprise software and data analytics and currently works on open source media as a codec engineer at Mozilla Research.