Game-Theoretic Approaches for Complex Systems Optimization

by

Shih-Fen Cheng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Industrial and Operations Engineering)
in The University of Michigan
2006

Doctoral Committee:

Professor Robert L. Smith, Co-Chair
Professor Michael P. Wellman, Co-Chair
Associate Professor Satinder Singh Baveja
Associate Professor Marina A. Epelman

To my parents, my parents-in-law, and my wife,
for their unconditional love and support.

ACKNOWLEDGMENTS

This thesis includes joint work with Robert Smith, Michael Wellman, Marina Epelman, Daniel Reaume, Archis Ghate, Daniel Reeves, Kevin Lochner, Blake Nicholson, and Stephen Baumert. In Section 1.2, I provide a brief summary on the connection between chapters in this thesis and joint work with these co-authors. Kevin O'Malley was one of the primary developers that laid the foundation for AB3D, which is the platform for market game simulations in Part II.

I would like to thank my thesis committee — Robert Smith, Michael Wellman, Marina Epelman, and Satinder Singh Baveja — for their careful reading and insightful comments. Their feedbacks are invaluable in improving the thesis.

This thesis could not be completed without the guidance from my advisors, Michael Wellman, Robert Smith, and Marina Epelman (although she is not officially listed due to Rackham's rule, she is really my third advisor for the efforts she devoted in advising me). Each of them has very different advising style, however, their passion for research, teaching, and advising is unmatched. Their positive attitude towards my work have helped shaping my research career, and they provided ideal and vivid images on what passionate researchers should be like. They also showed great considerations for my career development, especially during the final stage of my Ph.D. study. I owe a great deal to them.

I am grateful to the members of both Decision Machine Group and Dynamic Systems Optimization Laboratory: Daniel Reeves, Kevin Lochner, Archis Ghate, Blake Nicholson, Chris Kiekintveld, Yagil Engel, Irina Dolinskaya, and Stephen Baumert. Their feedbacks on my work and my various presentations are greatly appreciated. I especially would like to thank Dan and Kevin for their numerous proof-readings of my writings and also their social support. Without them, my life as a graduate student would certainly be less delightful.

I also would like to thank my coworkers and friends at AATPC and Mustardseed, for their continuous support and prayers. And finally, I would like to thank my family, especially my wife, Cindra, for her continuous patience and effort in pushing me through the Ph.D. program, for taking care of our son, Ian, and for voluntarily being the first audience on almost all my research talks.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

LIST OF APPENDICES

APPENDIX

ABSTRACT

Game-Theoretic Approaches for Complex Systems Optimization

by

Shih-Fen Cheng

Co-Chairs: Robert L. Smith and Michael P. Wellman

A complex system is an artificial system that cannot be modeled analytically or optimized in an effective manner, usually because it possesses the following properties: (1) the system can only be modeled as a simulation, (2) the size of the problem is untenable, so that even if the system could be modeled analytically, it would be impractical to solve it exactly, (3) necessary information required for problem solving is distributed in nature. This thesis presents methods for modeling and optimizing systems with the above challenging properties.

We first discuss the important modeling decision of whether to include stochasticity. By employing a real-world case study, we show that a standard numerical procedure can indeed help us make this decision. Next, we use the challenging problem of finding coordinated signal timing plans to motivate the need of a new paradigm for simulation optimization. We employ the game-theoretic paradigm of sampled fictitious play (SFP) to iteratively converge to a locally optimal solution. The key to the empirical success of SFP is parallelization. Through parallelization, SFP is robustly scalable to realistic size networks modeled with high-fidelity simulations. Compared to other less adaptive approaches, significant savings are achieved. This procedure is standardized so that we can use it to solve many unconstrained discrete optimization problems. However, for constrained problems, additional effort is required in using SFP. We introduce the idea of feasible space mapping which, when combined with SFP, can be used in decomposing and approximating large-scale dynamic programming models. With a large scale decision making problem in automotive manufacturing, we demonstrate that high quality solutions can be obtained by this approach in several orders of magnitude faster time than the traditional global algorithm.

Finally, for distributed problems, we address the decentralization issue with a market-based approach. The market-based approach involves: (1) agent strategy development, (2) empirical game-theoretic analysis, (3) assessing efficiency of the solution obtained by the market-based approach. We first introduce the market-based approach, with special attention on the strategy-pruning techniques. We then use task allocation for dynamic information processing environments as an example to illustrate the methodology and demonstrate its effectiveness.

CHAPTER 1

Introduction

1.1 Scope of the Research

This thesis is devoted to the optimization of complex artificial systems. A "complex artificial system", by our definition, is a system with following properties: (1) the system can only be modeled as a simulation, (2) the size of the problem is untenable, so that even if the system could be modeled analytically, it would be impractical to solve it exactly, (3) necessary information required for problem solving is distributed in nature.

In some cases, although the problem may look complicated at first sight, with a careful modeling effort, we can use a far simpler model in representing the original problem without losing modeling fidelity. With sufficient simplification, the global optimum to the problem usually can be found in a reasonable amount of time. We should always look for such opportunities to simplify the problem before abandoning our attempt to solve the problem exactly.

However, in many practical examples, when all our efforts at simplifying the model have failed (here, failure in simplification means that if we simplify the model any further, the resulting model will be unrealistic and unrepresentative), we usually end up with a model that is too huge to be solved by any exact algorithm. For different reasons to be cited later, for both centralized and distributed cases, one of the more effective ways to solve these optimization problems is through decomposition. Game theory will be shown to be a useful tool for performing analysis in these decomposed resource allocation problems. For centralized problems, the main questions considered are:

- How can we decompose the optimization problem? What is the general procedure one can use to solve an optimization problem in the above "complex-system" settings?
- What are the properties of a solution obtained in a game-theoretic manner?
- What is the complexity of obtaining solutions in decomposed problems? How does it compare to other competing algorithms, especially those that find global optima?

In Part I of this thesis, we try to answer the above three questions by investigating two representative examples in simulation optimization and large-scale Markov decision process. The study of these applications can help us in constructing a general framework for solving a general class of optimization problems in complex artificial systems.

For distributed problems, we consider markets as a mechanism for directing resource allocation. In studying decentralized resource allocation problems, we are interested in both empirical game-theoretic analysis and market-based approaches. The empirical game-theoretic analysis focuses on techniques for estimating market games via running simulations and solving for Nash equilibria (to be defined in Chapter 2). On the other hand, the market-based approaches focus on solving decentralized resource allocation problems with market mechanisms. For these two topics, the main questions considered are:

- How efficiently do markets allocate resources, when compared to other alternatives and global allocation?
- How do we identify and quantify possible sources for the loss of efficiency in markets?
- How can we design approximated approaches (with proper error bounds) for searching for Nash equilibria in a large game?

In Part II of this thesis, we try to answer the above three questions from two fronts, empirical game-theoretic analysis and market-based approaches, respectively.

1.2 Organization

Chapter 2 provides a review of the basics of game theory. Important notation and terms like "game" and "equilibrium" are defined. Related theorems are also listed for completeness.

Chapter 3 presents a case study on the important modeling decision of whether to include stochasticity [Cheng *et al.*, 2006]. Stochasticity is a common modeling feature in many applications, however, including it without careful consideration would sometimes bring in only limited benefits at extremely high cost in computation. The case study presented in Chapter 3 demonstrates the use of a standard procedure for making this decision empirically.

In cases where models inevitably become too big to be solved exactly, we need to consider approximation algorithms. Part I covers how to approximately solve complex discrete optimization problems under various conditions. The common theoretical tool used from Chapter 4 to Chapter 7 is sampled fictitious play (SFP). The basics of SFP and the motivation for using it in searching for solutions are introduced in Chapter 4.

Chapter 5 presents a parallel implementation of SFP on a challenging coordinated traffic signal control problem [Cheng *et al.*, 2007]. The purpose of this chapter is to demonstrate how one can use SFP as an off-the-shelf tool to optimize a black-box type

objective function with unknown properties and lengthy evaluation time. As demonstrated in this chapter, for time-sensitive applications, parallelization of the algorithm is extremely important, and SFP can be easily scaled up in a parallel mode in order to meet this need.

Chapter 6 discusses how to approximate an optimal control policy with SFP in a Markov decision process [Cheng *et al.*, 2005a]. The Markov decision process studied in this chapter is used in modeling a joint optimization problem in general production systems. The major challenge addressed in this chapter is the handling of non-trivial constraints[1]. Chapter 7 then concludes the first part.

Part II is devoted to the study of market-based approaches for decentralized resource allocation problems. Chapter 8 poses the challenges of decentralization, and motivates the use of market mechanisms in dealing with it. Chapter 9 then provides an overview on the empirical analysis methodologies that have been outlined by MacKie-Mason and Wellman [2006].

Chapter 10 investigates the technique of strategy pruning, and its implications for the quality of the solution [Cheng and Wellman, 2007]. By pruning unpromising strategies aggressively, and accepting errors along the way, the method proposed in Chapter 10 enables us to analyze games we could not handle previously.

Chapter 11 presents a case study on a general task allocation scenario. This chapter provides a thorough analysis of a typical decentralized resource allocation problem. It serves as an example of how to use market-based approaches in real-world applications. We then conclude the second part in Chapter 12.

[1]It refers to any constraint that is not in the format of $l \leq x \leq u$, where l and u are constants.

CHAPTER 2

Preliminaries: Basics of Game Theory

Let Γ be a finite game in strategic form, i.e., a game with finite number of players, with each player's action set being finite and non-empty, and the payoff function being well defined for all joint actions. Let $\mathcal{N} = \{1, 2, \ldots, |\mathcal{N}|\}$ be the set of players in Γ. For each player $i \in \mathcal{N}$, let S_i be the finite set of feasible actions. We use s_i to denote an element of S_i. Let $\mathcal{S} = S_1 \times S_2 \times \ldots S_{|\mathcal{N}|}$ be the set of feasible joint actions by all the players. We use s to denote an element of $\mathcal{S}$. The payoff function of player $i \in \mathcal{N}$ is denoted by $u_i : \mathcal{S} \to \Re$. For convenience, we use the tuple $[\mathcal{N}, \{S_i\}, \{u_i(s)\}]$ to represent a game Γ.

For player $i \in \mathcal{N}$, let Δ_i be the set of mixed strategies, i.e.,

$$\Delta_i = \{f_i : S_i \to [0, 1] : \sum_{s_i \in S_i} f_i(s_i) = 1\}.$$

Let $\Delta = \Delta_1 \times \Delta_2 \times \ldots \Delta_{|\mathcal{N}|}$. For player $i \in \mathcal{N}$, we extend u_i to be its payoff function in the mixed extension of Γ. That is, for any $f \in \Delta$,

$$u_i(f) = u_i(f_1, f_2, \ldots, f_{|\mathcal{N}|}) = \sum_{s \in \mathcal{S}} u_i(s_1, s_2, \ldots, s_{|\mathcal{N}|}) f_1(s_1) f_2(s_2) \ldots f_{|\mathcal{N}|}(s_{|\mathcal{N}|}),$$

where we have assumed that players choose their actions independently.

For $g \in \Delta$ and $f_i \in \Delta_i$, we use (f_i, g_{-i}) to denote $(g_1, g_2, \ldots, g_{i-1}, f_i, g_{i+1}, \ldots, g_{|\mathcal{N}|})$, which is a joint mixed strategy. We say that g is a *Nash equilibrium* if all players play strategies that are best responses to the others, as made precise in the following definition:

Definition 2.1 *A strategy profile g is a Nash equilibrium (NE) of game Γ iff for every $i \in \mathcal{N}$, $f_i \in \Delta_i$, $u_i(g) \geq u_i(f_i, g_{-i})$.*

Nash [1950] proved that every finite game in strategic form has a mixed-strategy NE. We also define an approximate version of NE.

Definition 2.2 *A strategy profile g is an ϵ-Nash equilibrium (ϵ-NE) of game Γ iff for every $i \in \mathcal{N}$, $f_i \in \Delta_i$, $u_i(g) + \epsilon \geq u_i(f_i, g_{-i})$.*

A *belief path* $\{f(t)\}_{t=1}^{\infty}$ is a sequence in Δ. We say that the belief path $\{f(t)\}_{t=1}^{\infty}$ converges to equilibrium if every accumulation point of $\{f(t)\}_{t=1}^{\infty}$ is an equilibrium. That is, a belief path that converges to an equilibrium is eventually arbitrarily close (in Euclidean norm in an appropriately defined Euclidean space containing Δ) to some equilibrium of the mixed extension of Γ.

Two special classes of games are of particular interest in this thesis: games with identical interests, and games that are symmetric with respect to payoffs. These two classes of games are defined as follows:

Definition 2.3 *A game in strategic form is said to have identical interests if for all* $s \in \mathcal{S}$, $u_1(s) = u_2(s) = \ldots = u_{|\mathcal{N}|}(s)$.

Definition 2.4 *A game in strategic form is symmetric if for all* $i, j \in \mathcal{N}$*: (a)* $S_i = S_j$*, and (b)* $u_i(s_i, s_{-i}) = u_j(s_j, s_{-j})$ *whenever* $s_i = s_j$ *and* $s_{-i} = s_{-j}$.

For a detailed description of important concepts in game theory, please see Fudenberg and Tirole [1991].

CHAPTER 3

When to Include Stochasticity: A Case Study of End-State Planning Problem in Production Lines

Building models for optimization problems is sometimes more like art than science. Important modeling decisions, like identifying important features, evaluating the value of each candidate feature, and balancing between simplicity and realism, are usually based on intuition and the process of trial-and-error.

In many applications, properly accounting for stochasticity is usually the single most important consideration in building models. The decision on whether to model stochasticity or not is tough to make and it may be tempting to include it whenever uncertainty is observed in the model. However, extending a model without careful consideration usually results in an unsolvable model. Worse, even in cases where we could solve the augmented models, it is not clear if the benefits we would enjoy would always be significant.

In this chapter, we look at a challenging real-world scenario on end-state planning in production lines. Whether we should incorporate stochasticity or not is an important decision we must make in this case study. We use the expected difference between the perfect information model and the nominal model to estimate the potential gain we can get by considering stochasticity. This measure provides valuable information in making our final modeling decision.

This chapter is organized as follows. Section 3.1 presents the background and the motivation for formulating the end-state problem. Section 3.2 introduces an abstract production line model; with this model, the problem of finding an optimal shutdown schedule when considering both end-state and production goals is then formally stated. Section 3.3 presents an efficient dynamic programming formulation for finding the optimal shutdown schedule. Section 3.4 describes some special cases where the problem can be solved even more efficiently. Section 3.5 summaries computational experiments, with special attention placed on computing the potential benefit of considering stochasticity. Finally, Section 3.6 concludes the chapter, and discusses our lessons from this case study.

3.1 Introduction

Maximizing equipment utilization is essential to the profitability of capital-intensive production processes. Although much research addresses the problem of how to minimize system downtime, little has been written about how to most effectively use the remaining downtime for a variety of critical tasks such as preventive maintenance, calibrations, installations, and upgrades that can be performed only when the system is down. Complicating this challenge is the fact that the contents of the production system when it shuts down may constrain the performance of such downtime tasks. For example, in a production line consisting of stations separated by buffers, consider the task of upgrading a particular station. Safety or accessibility needs might dictate that this station be empty of jobs if the upgrade is to be performed. Moreover, validating the upgrade requires a supply of jobs of appropriate types immediately upstream of the station, together with sufficient empty space downstream to accept these jobs after they are processed. The challenge of achieving as many such requirements (which will be called end-state goals in the rest of the chapter) as possible while trading off potential lost production or overtime costs is often an exceedingly difficult optimal control problem. To our knowledge, this problem has not yet been addressed in the literature. In this chapter we present a dynamic programming model for computing a production system control policy that optimizes the expected value of the system shut-down. Note that although we specifically discuss an automotive assembly application, this methodology is applicable to any system involving work-in-process inventory. Examples include oil refineries, semiconductor manufacturing, transactional back-office operations, and new product development and introduction pipelines. Cheung *et al.* [2004] described one such example faced by chemical production facilities.

3.2 A Graph Model of the End-State Planning Problem

In this section we introduce the use of a graph model in representing production line. Relying on this representation, we then formally define the optimization problem of satisfying end-state goals considering the cost of overtime and lost production time.

3.2.1 A Graph Model for Representing Production Lines

A typical production line contains three types of elements: work stations that are used in accomplishing certain manufacturing tasks (part processing, assembling, to name a few), buffers that are used to store work in process, and connectors that connect work stations and buffers. In most cases, work in process can only be stored in buffers or work stations, therefore, when defining end-state goals, we assume that only work stations and buffers will have end-state goals defined.

An end-state goal for each production line element (work station or buffer) is represented as a collection of constraints on its content (which can be a list of allowable types of jobs, or simply a job count) when the production line comes to a full stop. In

general, satisfying all requirements may not always be possible because: 1) the provided build schedule[1] may cause conflicts among stations or buffers, and 2) satisfying all requirements may require unreasonable overtime, or it may require the line to be shut down very early, which can be prohibitively expensive. These potential conflicts, as mentioned above, are what make the end-state planning problem challenging.

By defining work stations and buffers (both referred to as *line elements* in the rest of this chapter) as nodes, and connectors/conveyors as arcs, we can describe a general class of production lines as directed graphs. However, to simplify the problem, we will focus on the most commonly seen topology, serial line topology, for the rest of the chapter. Moreover, in our graph model, we will assume that shutdown decisions are only made at nodes (i.e., line elements). A graph for the serial line topology can be seen in Figure 3.1. Note that for the convenience of later explanation, we will assume that line elements are labeled from the tail of the line to the head of the line. Jobs numbered from 1 to J will enter the line at line element N and exit at line element 1.

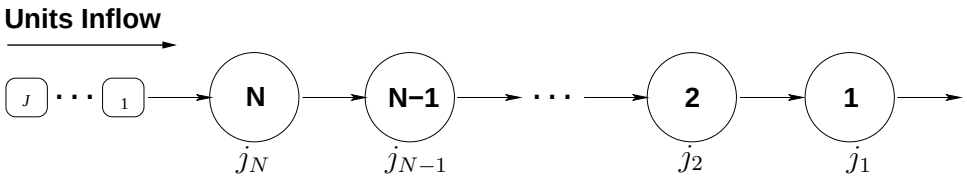

Figure 3.1: A serial production line. The jobs enter the production line at line element N, and exit at line element 1.

We will now formally introduce notation used in defining the end-state planning problem:

- Let N be the number of line elements in the production line.
- Let $\mathbf{N}$ be the set of line elements. Line elements are numbered starting from the end of the line. Thus, the last line element will have an ID of 1, while the first line element will have an ID of N. The reason for this numbering system will become clear later.
- Let $\mathbf{A} = [a_{ij}], i, j \in \mathbf{N}$ be the adjacency matrix. If there is a link leading from line element i to line element j, $a_{ij} = 1$, otherwise $a_{ij} = 0$.
- Let $\mathbf{G} = (\mathbf{N}, \mathbf{A})$ be the directed graph representing the production line.
- Let J be the number of jobs flowing into the production line.
- Let $\mathbf{J}$ be the ordered set of IDs for the jobs flowing into the production line. It is assumed that jobs in this set will enter the production line one by one, starting with the first job.

[1]A build schedule for a production line is a list of jobs to be processed in order. Usually some vital job-related information will also be included in the list. In our case, the style of each job is required.

- Let m_n be the capacity of line element n, $n \in \mathbf{N}$.
- Let r_n be the tuple that describes the list of jobs contained in n, $n \in \mathbf{N}$.
- In our formulation, we define a goal associated with each line element n as R_n, where R_n is a set of acceptable tuples in the format $(i_n^1, \ldots, i_n^{m_n})$, $n \in \mathbf{N}$, where i_n^k refers to a particular job type.
- Let T_d be the desired line shutdown time, e.g., the end of the normal shift. A penalty is assessed if our shutdown schedule induces a shutdown time other then T_d.

3.2.2 The Formal Definition of the End-State Planning Problem

The goal of the end-state planning problem, as described earlier, is to find shutdown schedules for all line elements in a production line, so that the value of meeting end-state goals minus costs from running overtime or lost production time, is maximized. To make this statement precise, we need to specifically define: 1) what constitutes a shutdown schedule? and 2) how do we know if an end-state goal is satisfied?

The shutdown schedule at some line element n, $n \in \mathbf{N}$, can either be an absolute time, t_n, or a job ID, j_n, where j_n specifies the job ID of the last job to be released from line element n. Given that the service time at the line element may be stochastic, we choose to use j_n in order to have better control over the production line (since the time when a job reaches a line element may be uncertain).

At the end of the horizon (when all line elements are shut down), if the tuple of jobs within line element n, r_n, is in the set R_n, a value v_n will be awarded, otherwise, a value p_n will be penalized. Note that in practice, our goal R_n may be generated from a fairly general statement (e.g., 5 vehicles regardless of their styles), and thus checking goal fulfillment by tuple matching is obviously very inefficient in these cases (the size of R_n will be very large in these cases). Here we suggest the tuple matching mechanism just to explain the concept. In practice, the goal matching can be more specialized, e.g., we can use predicates $\leq$, ==, and $\geq$ on the number of jobs in a line element, and we can define an end-state goal as a logical statement: (number of jobs in line element n) == 5.

Assuming the system can be simulated, we can view the simulation as a function, $F(\{j_n\}, T_{\text{max}})$, that takes the decisions at the line elements, $\{j_n\}$, and the upper bound on the production line running time, T_{max}, as inputs. The outputs of the simulation are $(T_s, \{r_n, n \in \mathbf{N}\})$, where T_s represents the time at which the production line comes to a full stop[2] as a result of the decisions, and r_n represents the state of line element n at the time the line stops. In our formulation, T_d is defined as the desired stopping time. When $T_s > T_d$, overtime cost will be incurred. Alternatively, if $T_s < T_d$, a penalty associated with lost production time will be charged. We denote unit overtime cost by p_o, and unit

[2]Note that since line elements in the production line may stop at different times, the shutdown time of the line is defined as the maximal shutdown time observed.

lost production penalty by p_l. The problem can thus be formally defined as:

$$\max_{j_1,\ldots,j_N} \quad \sum_{n\in\mathbf{N}} [I_n v_n - (1-I_n)p_n] - p_o(T_s - T_d)^+ - p_l(T_s - T_d)^- \tag{3.1}$$

s.t.

$$(T_s, \{r_n, n \in \mathbf{N}\}) = F(\{j_n, n \in \mathbf{N}\}, T_{\text{max}})$$

$$I_n = \begin{cases} 1 & , \quad r_n \in R_n \\ 0 & , \quad \text{o/w} \end{cases} \quad , \forall\ n \in \mathbf{N}$$

$$j_n \in \mathbf{J}, n \in \mathbf{N}$$

3.3 Deterministic Dynamic Programming Formulation

The key challenge we must address in the model is deriving analytical expressions for $r_n, n \in \mathbf{N}$, and T_s.

3.3.1 Deriving End States from the Shutdown Schedule

To derive end states from the shutdown schedule, we require that the shutdown schedule be jointly feasible in the sense that for each line element n, $n \in \mathbf{N}$, job j_n will eventually be processed at line element n.

For two consecutive line elements $n-1$ and n, j_n must be at least j_{n-1}. And since the difference of j_n and j_{n-1} indicates the number of jobs left in line element $n-1$, we require that $j_n - j_{n-1} \leq m_{n-1}$. Summarizing the above two observations, the values for j_n are thus constrained as follows:

$$j_n \in \begin{cases} \mathbf{J} & n = 1 \\ \{j_{n-1}, j_{n-1}+1, \ldots, j_{n-1}+m_{n-1}\} & n > 1 \end{cases} \tag{3.2}$$

With all j_n feasible, the end state of line element n can then be obtained as follows:

$$r_n = \{j_n + 1, \ldots, j_{n+1}\}, \quad n < N, \tag{3.3}$$

where line element n is empty if $j_n = j_{n+1}$. However, the contents of line element N cannot be decided directly, because both j_n and j_{n+1} are required in order to decide line element n's content. To handle this special case, we can define a dummy element in front of line element N, so that its content can be explicitly controlled. With the introduction of the dummy element, r_N can also be determined similarly to (3.3).

3.3.2 Computing Shutdown Time from the Shutdown Schedule

Let $t_{j,n}$ be the processing time for job j at line element n, and $e_{j,n}$ be the time when job j exits line element n. For job j, at the time when it finishes processing at element n, it can move on to the line element $n-1$ if there is spare capacity available, otherwise it will wait in the current element until the job in element $n-1$ finishes processing. From

this observation, and the assumptions that the production line is linear and all parameters are deterministic, we can compute $e_{j,n}$ iteratively:

$$e_{j,n} = \max\{e_{j,n+1} + t_{j,n}\ ,\ e_{j-1,n} + t_{j,n}\ ,\ e_{j-m_{n-1},n-1}\}\ ,\ j = 1, \ldots, J\ ,\ n = 1, \ldots, N. \tag{3.4}$$

Equation (3.4) describes three requirements for a job to move from line element $n+1$ to line element n. To simplify the formulas, let $e_{j,n}$ be 0 if either $j \leq 0$ or $n \leq 0$ or $n > N$. We will explain what each term means as follows. The first term, $(e_{j,n+1} + t_{j,n})$, states that job j must exit preceding element $n+1$ before entering element n. The second term, $(e_{j-1,n} + t_{j,n})$, states that preceding job $j-1$, must finish processing at element n before job j can be processed at element n. The third term, $e_{j-m_{n-1},n-1}$, represents the time when the first job in element $n-1$'s queue exits. This time matters if element $n-1$'s queue is full when job j finishes processing at element n. In this case, job j cannot exit element n until the head job in element $n-1$ exits. Taking the maximum over these three terms guarantees that all requirements are met when job j exits element n.

Obviously, the production line shutdown time can be directly computed from $\{e_{j,n}\}$ and collection of decisions $\{j_n\}$, as:

$$T_s = \max_{n \in \mathbf{N}} \{e_{j_n,n}\}. \tag{3.5}$$

Let T_n be the maximal shutdown time from line element 1 to n, then the production line shutdown time can also be computed recursively by:

$$T_n = \max\{e_{j_n,n}, T_{n-1}\}, \tag{3.6}$$

and $T_s = T_N$.

3.3.3 Dynamic Programming Model

From the above assumptions and derivations, we can see that this problem can be cast as a sequential decision process, where each line element, starting from line element 1, successively makes a decision. From Equation 3.2, we can see that for line element n to make a "feasible" decision, it must know j_{n-1}. Also, as shown in (3.4), the time when each job leaves each line element can be computed a priori and is considered given input data for the problem.

From the above descriptions, we can see that the minimal amount of information required to make an optimal decision at each line element includes: n, the current line element ID; j, the decision from the downstream line element; and T_{n-1}, the maximal shutdown time up to line element $n-1$. The reward/penalty for choosing decision j_n at line element n can be obtained by first computing end-state tuples according to equation (3.3), and by looking up the end-state tuples in the goal set, we can have the reward/penalty. Note that we can calculate the reward/penalty at line element n only after we have made a decision for line element $n+1$. This is due to the fact that the contents of element n are not known until the decision at element $n+1$ is made (see Equation 3.3). Because of this, we will have to insert a dummy element in front of line element

N in order to control the content of line element N. This dummy element is assumed to have zero cycle time and capacity large enough to hold all the jobs. With these two assumptions, the addition of this dummy element will not affect other part of the model except granting us the ability to control the content of line element N.

When we reach line element $N+1$, the beginning of the line, we will have T_s, and the overtime/lost production cost can be computed accordingly.

The DP formulation is formally described as follows:

- The state for the DP is defined as (n, j, T):
  - n is the stage variable, representing the ID of the current line element,
  - j is the decision from element $n-1$, it serves as the lower bound on j_n,
  - T is maximal shutdown time of line elements from 1 to $n-1$.

  Note that for $n = 1$, there is no element $n-1$, thus there is only one state for $n = 1$, and that is $(n, 0)$.

- Feasible decision at state (n, j, T):

$$j_n \in \begin{cases} \mathbf{J} & , \quad n = 1 \\ \{j, j+1, \ldots, j+m_{n-1}\} & , \quad n > 1 \end{cases} \tag{3.7}$$

- Reward function at state (n, j, T) with decision j_n:

$$\begin{aligned} V(n, j; j_n) &= (I_{n-1} \cdot v_{n-1} - (1 - I_{n-1}) \cdot p_{n-1}), 2 \le n \le N+1 \\ r_{n-1} &= \{j+1, \ldots, j_n\} \\ I_{n-1} &= \begin{cases} 1 & , \quad r_{n-1} \in R_{n-1} \\ 0 & , \quad \text{o/w} \end{cases} \end{aligned} \tag{3.8}$$

- Overtime and lost production: overtime and lost production is only charged at the line element $N+1$, by using the formula:

$$L(T) = p_o(T - T_d)^+ + p_l(T_d - T)^+ \tag{3.9}$$

- Functional equation at state (n, j, T): maximal value one can get by acting optimally from line element n to N, if current state is (n, j, T).

  For $n = 1$:

$$f(n, 0) = \max_{j_n \in \mathbf{J}} \{f(n+1, j_n, e_{j_n, n})\} \tag{3.10}$$

  For $2 \le n \le N$:

$$f(n, j, T) = \max_{j_n \text{ feasible}} \{V(n, j; j_n) + f(n+1, j_n, \max\{T, e_{j_n, n}\})\} \tag{3.11}$$

  For $n = N+1$:

$$f(n, j, T) = \max_{j_n \text{ feasible}} \{V(n, j; j_n) - L(T)\} \tag{3.12}$$

- The answer: $f(1, 0)$

Starting Production Line with Initial Content

In the DP model described in the previous section, we implicitly assumed that the production line is started empty, with job 1 just about to enter the line. This restriction can be easily lifted. We can emulate the effect of starting the production line with given initial content by pruning proper states from the DP.

Suppose we are given an initial state of the system, indicating the position of each job. Let p_k be job k's initial position (a line element ID). If job k has not entered the system, let p_k be ∞. We can see that since job k starts at line element p_k, all line elements upstream (with ID greater than p_k) cannot use job k as shutdown decision. As a result, states (n, j, T), where $n > p_k$, $j \leq k$, and $\forall\, T$, will be pruned.

Computational Complexity of DP Formulation

Here we compute an upper bound on the computational effort required in solving the above dynamic program. The number of floating point operations required for computing a functional value for each state (n, j, T) is approximately:

$$A_n \;=\; \begin{cases} (m_n+1)(t_v+4)+m_n & , \quad 1 \leq n \leq N \\ (m_n+1)(t_v+1)+m_n & , \quad n = N+1, \end{cases}$$

where t_v is an upper bound on number of floating point operations required to compute $V(n, c; c_n)$. Given that $n = 1, \ldots, N+1$, $j = 1, \ldots, J$, $T = 1, \ldots, T_{\text{max}}$, the total number of floating point operations required is:

$$\begin{aligned} & \sum_{n=1}^{N} J \cdot T_{\text{max}}((m_n+1)(t_v+4)+m_n) + J \cdot T_{\text{max}}((m_{N+1}+1)(t_v+1)+m_{N+1}) \\ = \;& N \cdot J \cdot T_{\text{max}}((\bar{m}_n+1)(t_v+4)+\bar{m}_n) + J \cdot T_{\text{max}}((m_{N+1}+1)(t_v+1)+m_{N+1}) \\ = \;& J \cdot T_{\text{max}}\left(N \cdot \bar{m}_n(t_v+5) + m_{N+1}(t_v+2) + 2t_v + 5\right), \end{aligned}$$

where $\bar{m}_n$ is the mean capacity of the line elements.

From the model data based on the GM Lansing Grand River assembly plant, we can roughly conclude that $\bar{m}_n$ is 1.167. Taking, for example, $N = 66$, $J = 200$, $T_{\text{max}} = 4800$ (seconds), we can obtain a numerical lower bound for the complexity:

$$\begin{aligned} & J \cdot T_{\text{max}}\left(N \cdot \bar{m}_n(t_v+5) + m_{N+1}(t_v+2) + 2t_v + 5\right) \\ \approx \;& 7.394 \cdot 10^7 t_v + 3.697 \cdot 10^8, \end{aligned}$$

Modern CPUs, with operating frequency measured in GHz, can provide computational performance in the range of several GFLOPS (10^9 floating point operations per second). Suppose we are equipped with a machine with one GFLOPS capability, and let t_v be 100 (a number used for illustrative purpose); the problem can then be solved within 10 seconds. Even with $t_v = 1000$, the problem can still be solved within 2 minutes.

3.4 Special Cases: Strip-All and Exact Job-Count Goals

When the problem features certain goal structures, we can find optimal shutdown plans much faster by exploiting these goal structures. To be more specific, we will look at two commonly seen goal structures: (1) strip-all goals that seek to remove all jobs from the system. This type of goals are commonly seen during major shutdowns, like the semiannual shutdowns in automotive plants. (2) Job-count goals that specify desired number of jobs, regardless of types, in certain line elements.

To further simplify these special cases, we assume that achieving these goals has highest priorities, and thus we will always try to fulfill these goals if feasible.

In section 3.4.1, we will discuss the strip-all goals. In section 3.4.2, we will discuss exact job-count goals.

3.4.1 Strip-All Goals

First note that in order to keep a line element empty, we only have to make the same decisions for the current element and the previous element. Since our goal is to strip all line elements, this implies that decisions at all line elements should be the same. Therefore, the only effective decision we need to make is for line element 1 (the tail of production line). Once this decision is made, decisions for all other line element $n > 1$ will be the same (as discussed earlier in the section, we want to fulfill all the goals when feasible). Checking the reward function, we see that $L(T) = p_o(T - T_d)^+ + p_l(T_d - T)^+$ is the cost we want to minimize. Since $e_{i,j}$, the time when job i exits line element j, is monotonically increasing in i (given some j), and monotonically decreasing in j (given some i), T can be found as $e_{i,1}$ that minimizes $L(T)$. This can be achieved by performing a binary search on $e_{i,1}, i = 1, \ldots, J$, with complexity $O(\ln J)$.

3.4.2 Exact Job-Count Goals

Job-count goals is usually stated as: "line element i should be left with at least/at most/exactly n jobs". In this section, we will focus on the *exact* job-count goals.

Suppose $n_i > 0, i \in \mathbf{B} \subseteq \mathbf{N}$, is number of jobs that should be left at line element i at shutdown. n_i is then an exact job-count goal specified for line element i. For all other line elements $j \in \mathbf{N} \setminus \mathbf{B}$, strip-type goals are assumed, i.e., $n_j = 0, j \in \mathbf{N} \setminus \mathbf{B}$. Similar to the case where we have strip-all goals, when exact job count, n_i, is specified for the line element i, it implies that the difference between j_i and j_{i+1} should be n_i. Therefore, when the decision at certain line element i is fixed at j_i, the decisions at other line elements can be determined as follow:

$$j_k = \begin{cases} j_{k-1} + n_{k-1} & k = i+1, i+2, \ldots, N \\ j_{k+1} - n_k & k = i-1, i-2, \ldots, 1 \end{cases} \tag{3.13}$$

Following the procedure in section 3.4.1, we can find a j_1 that minimizes $L(T)$. By using (3.13), we can find decision at all other line elements. If $e_{j_i,i} \leq T_d, \forall i \in \mathbf{N}$, we are done.

Otherwise, (a) pick arbitrary $i \in \mathbf{B}$ with $e_{j_i,i} > T_d$, search for new $\hat{j}_i$ such that $e_{\hat{j}_i,i} \leq T_d$. (b) Update all $j_k, k \in \mathbf{N}$ by using (3.13). Repeat step (a) and (b) until $e_{j_i,i} \leq T_d, \forall i \in \mathbf{N}$. In the worst case, the complexity will be $O(|\mathbf{B}|(\ln J + N))$.

3.5 Computational Experiments

To demonstrate the benefit of using DP for the end-state planning problem, we use a hypothetical yet realistic end-state situation from the real production line, with parameters tweaked in order to preserve business secrecy.

In this section, we first describe the scenario. After that, we compare the performance of optimal policy obtained by our DP model and other "rule-of-thumb" policies. Finally, we examine the potential benefit we can gain if we explicitly consider stochasticity.

3.5.1 Description of the Scenario

An automotive plant is preparing for the launch of a new model, concurrently with the production of old models. This requires the installation of new equipment, calibration of new and old equipment, and verification that new equipment or newly calibrated equipment can still produce the existing model. At this time, suppose the plant is just starting to produce manufacturing validation builds — the first prototype builds of the new model built at the plant. Call the current model types 1, 2, and 3 and the model being launched type 4.

In our case study, we focus on two zones, engine compartment (EC) and underbody (UB) (as seen in Figure 3.2 and Figure 3.3), in an automotive body shop line. In both Figure 3.2 and Figure 3.3, larger squares labeled with identification numbers represent stations, smaller round squares labeled with capacities represent buffers. These two zones are both linear and connection is made from EC zone to UB zone. Therefore these two zones combined can be treated as a linear production line (as required by our model).

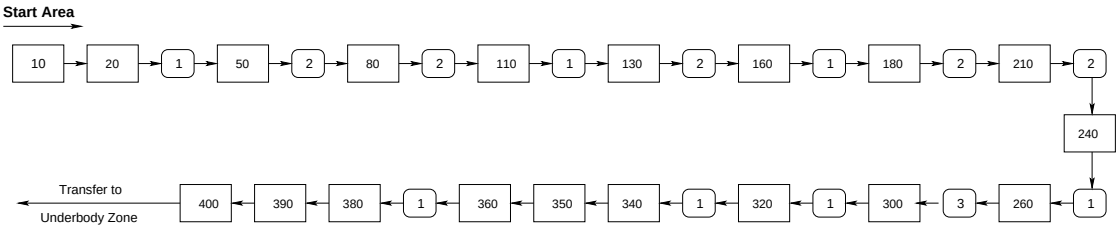

Figure 3.2: Schematic graph for the engine compartment zone.

Let lost production costs be 10/minute when early shutdowns are required, and overtime production costs be 5/minute (these are representative of real values with proper scalings). Let the goals be classified as low, medium, and high value. Low value goals earn 1 if achieved, cost 1 if not achieved. Medium value goals earn 5 if achieved and cost 3 if not achieved. High value goals earn 20 if achieved and cost 7 if not achieved (these

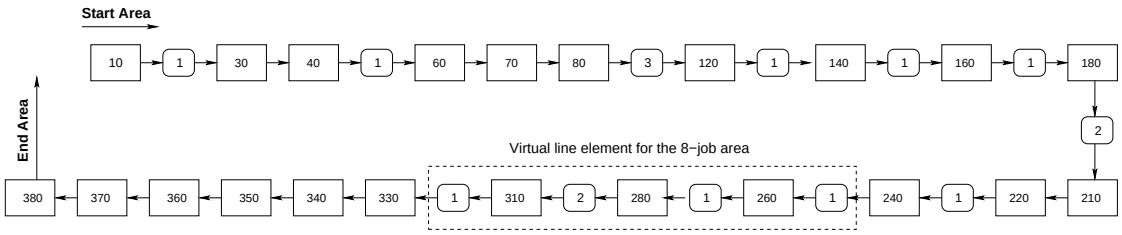

Figure 3.3: Schematic graph for the underbody zone.

values are approximate, but of the right scale). The desired goals are defined as follows:

1. The launch activities are causing more frequent downtime in the EC and UB areas, as these are the first impacted by new equipment. To prevent starving of the downstream system it is desirable to fill every station and buffer position in these areas with a vehicle of some sort. As each extra job will only marginally impact throughput, this implies a low value goal for each station and each buffer.

2. EC stations 20, 50, 80, 130, 180 and 260 are load stations. These should be empty to allow verification that material can be loaded into them from newly modified conveyor systems. (Note that this is in conflict with the goal above). Each of these goals is of medium priority, since the tests can be delayed, although this will slightly delay the launch. If critical, vehicles could be manually offloaded, at cost, from the line to empty these stations.

3. EC station 160 should have a job in it of type 4 to allow for training of the welding robot to follow a new weld-path for it. This is a high priority goal since this test is critical to launch timing. The buffers immediately before and after this job should be empty to allow engineers leeway to stop the line to better examine issues as this validation build progresses through the system. These latter goals are of low priority, since the only impact of not achieving them is lower throughput.

4. EC station 300 and 320 were re-calibrated yesterday to better process the new model. Unfortunately, there is worry that this may have caused problems with the calibration for model type 2. These two stations and their immediately preceding buffers should contain models of type 2 to allow for testing. These goals are of high priority since it is unacceptable to produce low quality current vehicles and it is very difficult to test the calibration in any other way. The remainder of the line after these stations should be empty to allow for jobs to be moved through stations 300 and 320. These goals are of medium priority, since they could be manually achieved at medium cost if not achieved through the actual end-state.

5. In the underbody line, new equipment is being installed for station 350. To ensure adequate working space, the area from station 330 to 370 inclusive must be emptied. These are medium priority goals since vehicles may be removed, at some cost.

6. In the underbody line, the 8-job area from the buffer prior to station 260 up to the buffer prior to station 330 should include 2 jobs of each type to allow for testing of the new equipment. Achieving the sequence of types such as 1-2-3-4-1-2-3-4 in this area is a high priority goal, since it is important to test whether the new equipment can adjust to a change of models. In fact, any sequence where the four types are cycled through in two sets of four jobs is acceptable to satisfy this high-priority goal. (Note: This is an example of mutually exclusive goals.) A less good test would be a sequence where each type appears twice among the 8 positions, like 1-2-1-2-3-4-3-4. This is a medium priority goal. An even lower value goal is for each type to appear at least once among the 8 positions. This is a low priority goal.

7. All of the respot stations (140, 180, 210, 220, 240, 260) are slated for re-calibration this evening for jobs of type 1 or 4. Having either a job of type 1 or 4 in each such station is a medium priority goal.

8. To enable precise measurements, the geometry setting stations, 80 and 120, should be emptied. This is a medium priority goal. Verification of these measurements requires that the job immediately preceding these stations be of type 4. These are medium priority goals.

Special attention should be paid to item 6 since the goals defined in item 6 are associated with a range of line elements, instead of a single line element as required by our model. To model this type of goals without modifying the DP model, we define a *virtual line element* for the range specified by the planner. In our example, the area beginning at the buffer prior to station 260 up to the buffer prior to station 330 in the underbody zone are viewed as a virtual line element (as drawn in Figure 3.3), with capacity 8 (the sum of capacities of contained line elements), and processing time equals to the sum of the processing times of contained line elements[3].

Finally, we have the following information:

- Desired shutdown time T_d: 4,200 seconds.
- Maximal allowed shutdown time $T_{\max}$: 4,800 seconds.
- Number of jobs: 200
- The line is initialized empty.

With this information, we are ready to solve for the optimal policy.

[3]Let n be the virtual line element. If goals are defined on line elements inside this virtual line element, we must modify the reward function when we compute the optimal decisions at line element $n+1$ (the decision made at line element $n+1$ determines the content of the virtual line element). For each feasible decision j_{n+1}, besides $V(n+1, j; j_{n+1})$ (as defined in Equation (3.8)) which looks at the goals defined on the virtual line element, we must also consider values and penalties from the goals associated with sub-line elements of the virtual line element. This sub-problem can be solved by a DP formulation similar to the grand DP.

3.5.2 The Optimal Policy and Alternatives

The test instance can be solved within 90 seconds on a Pentium-4 3.4 Ghz PC under RedHat Linux. With the optimal policy, the production line stops at 4,189 seconds (11 seconds earlier than the desired time), and out of 93 goals defined, we have achieved 69 goals, with the value from goals equals to 189 (the value from goals includes both rewards for achieving goals and penalties for missing goals). For each line element, we can compute the maximal achievable value by considering conflicting goals. To illustrate the gap between potential values and realized values, we plot both maximal achievable values and realized values in Figure 3.4. In Figure 3.4, grey bars represent the upper bounds on values achievable at all line elements, and the black bars are the value realized. If achieved value matches the bound, it is all black, otherwise, the gap is revealed in grey. We also plot each line element's shutdown time in Figure 3.5.

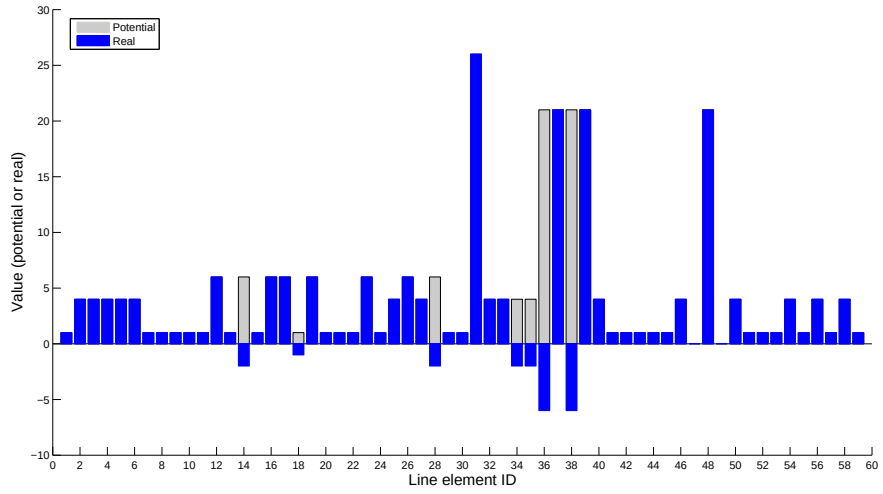

Figure 3.4: Maximal achievable value and value obtained in optimal policy.

As described in Section 3.5.1, every line element within the system is associated with at least one goal, and many line elements are associated with more than one goals, usually conflicting with each other. With 93 goals in the system, and having to make a reasonable trade-off between overall system shutdown time and which goals to satisfy, it is fair to say that it is impossible for a human planner to manually come up with shutdown plans near the quality of the optimal policy we obtain.

Just as a quick comparison, we can use one rule of thumb obtained from the field to see how well one can perform under customary rules. This rule of thumb states that the plant should be shut down as close to the desired shut down time as possible, and if any goal can be achieved while meeting this objective, it will be acceptable.

Again, even for this simple principle (stops production line at some predetermined time), it is extremely difficult to come up with a plan that would comply to this constraint, and maximizing the value we can get by meeting the goals. In fact, it is as hard as the

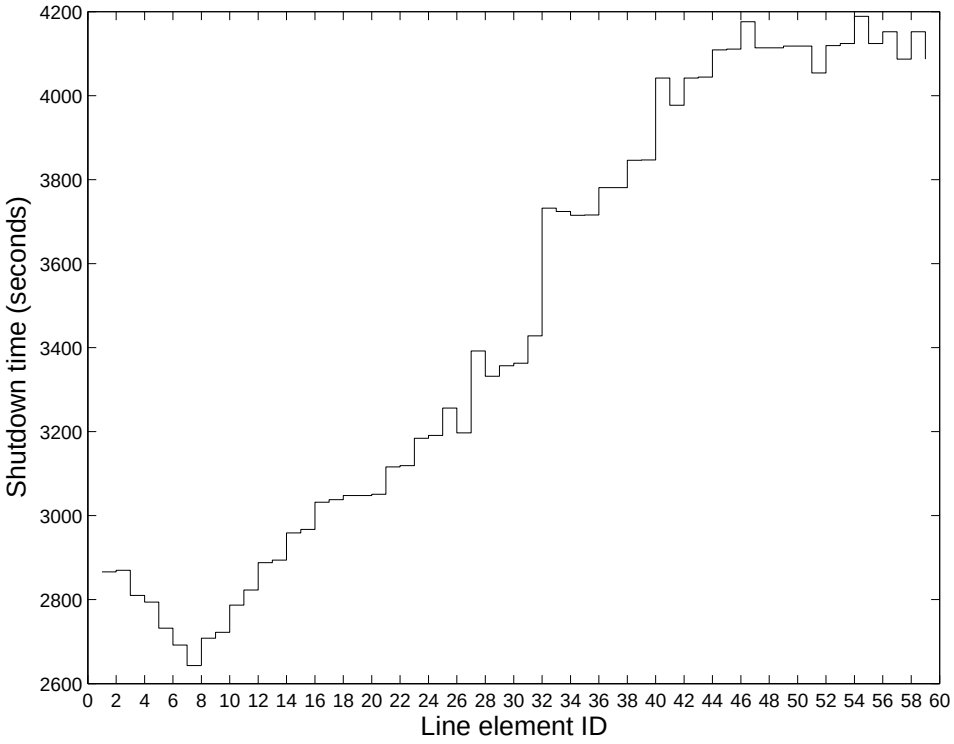

Figure 3.5: Shutdown time for each line element.

original problem. However, we can quantify the least loss this constraint would bring to the value we can get.

In our experiment, we simply assume that the planner (who has this constraint in mind) can somehow come up with an optimal plan under this constraint. The difference between this plan and the true optimal plan can then be viewed as the lower bound on the value that can be lost by implementing such rule (a very conservative one, since a human planner is not optimizing the goal satisfaction while shutting down the line).

Empirically, this rule can be emulated by setting overtime and lost production time cost to extremely high values, and running the same solver again. The resulting policy should then return a stopping time as close to the desired time as possible (while meeting goals optimally).

For this case, the found policy will shutdown the system exactly at 4,200 seconds, as desired (the production shutdowns 11 seconds earlier in the original case). However, the number of goals that can be achieved drops from 69 to 65, and the value from goals also drops from 189 to 156, implying that by requesting that we must stop as close to the desired time as possible, we are missing the opportunity of meeting high-value goals.

3.5.3 The Potential Benefits of a Stochastic Model

Up to this point we have assumed that the model is deterministic. However, one may wonder whether it is necessary to include stochasticity in the model in order to better

describe the scenario. Extending our model in order to incorporate stochastic events is not straightforward, and it makes our model significantly larger. Therefore, before delving into the details on the expansion of the model, we would like to quickly measure the potential benefit we can get by considering stochasticity. In this section, we first describe the origin of the stochasticity, and then we discuss how to estimate the value of stochastic model without having to construct one.

In the case studied here, the stochasticity comes from the operation of each line element. In the deterministic model, it is assumed that line elements operate smoothly without breakdowns. However, unexpected glitches happen at times, and they usually cause unexpected delay in the job processing. The following parameters are used to characterized the operation of every line element:

- Cycle time: time required to process a particular job. According to the operational experience, it is fair to assume deterministic cycle times for all line elements.
- Mean cycles between failure (MCBF): as its name suggests, MCBF specifies on average, how many cycles are required to see the next failure. It is assumed that "cycles between failure" is a random variable following an exponential distribution.
- Mean time to repair (MTTR): MTTR specifies how much time is required to repair a downed line element and restore its operation. It is also assumed that "time to repair" is a random variable following an exponential distribution.

The realizations of "cycles between failure" determine when a line element goes down during the planning horizon (each line element may go down multiple times). If line element n is down when it is processing job j, an additional amount of repair time, drawn from the time-to-repair distribution, will be required besides standard cycle time to complete the job. With this simple rule and the above information, we can then generate $e_{j,n}$, for all job j and line element n, using Monte Carlo simulation.

For every randomly generated instance, we can measure the performances of policies generated under different modeling assumptions, namely, perfect information model, stochastic model, and deterministic model, by executing each policy in this realized instance. The differences among these three models are the amount of information available to them. The perfect information model, as its name suggests, has access to all the realized information. For the stochastic model, the distributional information of random variables is available. For the deterministic model, only the means of random variables are available. By performing this type of analysis on a large number of instances, we can then estimate the expected performance of the policies generate under the above three modeling assumptions.

Let the expected performance for perfect information model, stochastic model, and deterministic model be EV_{PI}, EV_{S}, and EV_{D}, respectively. Since $EV_{\text{PI}} \geq EV_{\text{S}}$, the value of upgrading to a stochastic model from a deterministic model , $(EV_{\text{S}} - EV_{\text{D}})$, can be bounded as follows:

$$EV_{\text{S}} - EV_{\text{D}} \leq EV_{\text{PI}} - EV_{\text{D}} \tag{3.14}$$

Since the realizations of $e_{j,n}$ are available to the perfect information model, we can find the optimal policy for the perfect information model by using the same deterministic solver loaded with realized $e_{j,n}$. With this setup, we can estimate EV_{PI} and EV_{D} for the scenario as described in Section 3.5.1. Surprisingly, for 30 random instances we generated, $EV_{\text{PI}} = EV_{\text{D}}$. This implies that even when we consider the stochastic events of line elements breaking down, the policy generated deterministically performs as well as the policy generated with perfect information. According to (3.14), for this scenario, there is no point in including the stochastic features in the model.

3.6 Conclusion

In this chapter, we demonstrate a simple numerical procedure for measuring the value one can get from "upgrading" deterministic models to stochastic ones. As shown in this case study, the use of such tool can keep the model simple while providing a confidence on the error bound for neglecting stochasticity.

However, it is not always possible to ignore stochasticity. In cases where we are forced to extend the model, we have to carefully consider the trade-off between model complexity and the benefits of being more realistic. After all, no mater how realistic the model is, if we cannot solve it, it has limited value to us. Beginning in next chapter, we will engage in a discussion of methods that can help us deal with these additional model complexities, hence allowing us to build much complicated models than previously allowed.

PART I

Sampled Fictitious Play Algorithm for Large-Scale Discrete Optimization Problems

CHAPTER 4

An Introduction to the Sampled Fictitious Play Algorithm

As mentioned in Chapter 1, optimization problems in complex artificial systems are difficult to solve due to (1) discreteness, (2) lack of nice properties in objective function, and (3) size. Decentralization issues will be put off until Part II, and for now assume all problems can be solved centrally. The above three difficulties, when combined together, will result in combinatorial explosions of decision spaces, and in almost all cases no exact polynomial algorithm is known to exist. As a result, a great number of heuristics that aim at approximating a global optimum have been developed for a wide variety of such problems. Unfortunately, those heuristics are usually problem-specific and are not easily applicable to other classes of problems.

In recent years, researchers have been actively working on heuristics that can be used in solving a general class of combinatorial optimization problems. It was Glover [1986] who first coined the term *metaheuristic*, when he described *tabu search* as a method that superimposes on another heuristic. Since then, metaheuristic is widely used in referring to the study of general-purpose heuristics.

Effective metaheuristics usually have following characteristics:

- Most metaheuristics have used randomness to deal with impractically large solution spaces. In many cases, if every element within the solution space can be reached with nonzero probability, some forms of convergence results can be established.

- Many metaheuristics have their roots in natural phenomena. Notable examples include *genetic algorithms* (GAs), *ant colony optimization*, and *tabu search*, which were inspired by phenomena in biology; and *simulated annealing*, which was inspired by the annealing in metallurgy.

(For detailed discussion, see Dréo *et al.* [2006].)

The methodology used in this part falls in the general area of metaheuristics. The main idea of the approach is *divide and conquer*, i.e., decompose the original intractable problem into smaller, tractable subproblems, and solve these subproblems instead. However, naive divide and conquer will only work on problems that have separable objective functions. For problems with a considerable amount of interactions among subproblems, we have to carefully consider the impact these interactions have on objective function values and feasibilities, and devise a scheme that coordinates these subproblems properly.

In order to effectively coordinate a large number of subproblems, we turn to game theory, which has its roots in economics. Modern game theory was first introduced by von Neumann and Morgenstern [1947] and quickly became a popular tool in explaining and predicting behavior of groups of rational decision makers (*players* in game theory terminology) when their well-beings are associated with the joint actions of all decision makers (players). If each subproblem is associated with choices of a player in the game, and the objective function value is viewed as a *common payoff* for every player, the original optimization problem can then be represented as a *game of identical interests*. The notion of a solution to a game is that of a NE, which for a game of identical interests can be viewed as a coordinate-wise local optimum. Thus, instead of searching for an optimum for the original problem, after we successfully turn an optimization problem into a game, we search for the NE.

4.1 Searching for the NE

For games with large numbers of players, trying to locate a NE is a very challenging task. The most critical issue related to computing a NE, is the exponential growth of the size of a game in the number of players. In some real-world examples, we may have tens of thousands of players. Storing payoff values for all strategy profiles is impossible in these cases, let alone searching for a NE with the payoff matrices. Therefore, the algorithm we used in searching for a NE in a game should explore the payoff matrix incrementally, thus avoiding having to retain the whole payoff matrix (which is impossible in large games) from the very beginning.

In this thesis, we will use a simple-to-implement iterative algorithm which is a variation of Fictitious Play (FP). Convergence results for the FP algorithm and its variants are stated in Monderer and Shapley [1996] and Lambert *et al.* [2005]. We refer interested readers to Lambert *et al.* [2005] for a complete treatment. Besides FP, McKelvey and McLennan's work on GAMBIT [1996] is an excellent reference for various computational methods for finding NEs.

The intuition behind FP lies in the theory of learning in games. In a classical FP process (see, for example, Brown [1951]), every player assumes that other players are playing unknown stationary mixed strategies, and tries to learn them iteratively. The estimates of the unknown stationary mixed strategies are represented as *belief distributions*, or *beliefs*, and are shared among all players. The belief distribution for player i is a mixed strategy calculated by finding the relative frequency of all strategies from the history of its past plays. During each iteration, each player finds its *best reply* against the belief distribution of other players (i.e., its belief of how they will play). These best replies are then included in the history of past plays and the beliefs are updated accordingly. To start the FP process, an arbitrary joint strategy is used. The FP algorithm doesn't converge to equilibrium in general. However, for games of identical interests, as in our case, the sequence of beliefs generated by the FP algorithm are guaranteed to converge to equilibrium [Monderer and Shapley, 1996].

The best reply operation of the classical FP algorithm outlined above is too computationally expensive to implement in practice. Lambert *et al.* [2005] thus suggested a variant they called *sampled fictitious play* (SFP) that is computationally practical. SFP is very similar to FP except the best reply evaluation in each iteration is done against samples randomly drawn from the belief distribution instead of the belief distribution itself. A convergence result for SFP with gradually increasing sample sizes is proved in Lambert *et al.* [2005]. In practice, however, samples of size one are often used at each iteration.

The SFP algorithm, with sample size one, is described below:

1. **Initialization:** An initial joint strategy is chosen arbitrarily. It is then stored in the history.
2. **Sample:** A strategy is independently drawn from the history of each player (i.e., for each player, each past play is selected with equal probability).
3. **Best Reply:** For every player, the best reply is computed by assuming that all other players play the strategies drawn in step 2.
4. **Update:** The best replies obtained in step 3 are stored in the history.
5. **Stop?** Check if the stopping criterion is met; if not, go to step 2, otherwise stop.

The pseudo-code for the SFP algorithm and the sampling subroutine is listed in Figure 4.1. This pseudo-code is specified for a game with P players. Here, **D** and **B** are P-dimensional vectors whose components contain individual strategies of the players, and $(\cdot)^T$ denotes the transpose operation. **H** is a "history" matrix, where $\mathbf{H}(k, j)$ represents player j's best reply in the k^{th} iteration. Notation $\mathbf{H}(k, :)$ represents the k^{th} row of matrix **H**, while $\mathbf{H}(:, j)$ is the column containing the history of past plays of player j. This representation of the history allows convenient access to relevant information for sampling in step 2.

Algorithm 4.1 implements the SFP algorithm in a straightforward way. Line 1 generates an initial solution (joint strategy) by calling function INITIALSOLUTION, thus populating the 0^{th} row of history matrix **H**. Line 4 performs uniform sampling from each

ALGORITHM 4.1: SFP

1: $\mathbf{H}(0,:) \leftarrow$ INITIALSOLUTION()
2: $k \leftarrow 0$
3: **while** STOPCRITERION() is **false do**
4: $\quad \mathbf{D} \leftarrow$ SAMPLE($\mathbf{H}, k$)
5: $\quad \mathbf{B} \leftarrow$ BESTREPLY($\mathbf{D}$)
6: $\quad \mathbf{H}(k+1,:) \leftarrow \mathbf{B}^T$
7: $\quad k \leftarrow k+1$
8: **end while**

D = SAMPLE($\mathbf{H}, k$)

1: **for** $j = 1$ to P **do**
2: $\quad u \leftarrow$ DISCRETEUNIFORM($0, k-1$)
3: $\quad \mathbf{D}(j) \leftarrow \mathbf{H}(u, j)$
4: **end for**
5: **return D**

Figure 4.1: Sampled Fictitious Play (sample size 1).

player's history independently. Line 5 computes a best reply **B** to the sampled decision **D**. Line 6 appends **B** at the end of the history matrix **H**. Note that except for $k = 0$, each row k of matrix **H** stores best replies computed in iteration k. The above three lines are then repeated until STOPCRITERION returns **true**. Since the BESTREPLY subroutine simply solves a collection of P one-dimensional optimization problems whose input is the sampled decision **D**, it can be executed in parallel. As we will see in Chapter 5, the parallelization of the best reply computation is the most important feature that makes SFP algorithm efficient.

Although this is not explicitly specified in the general pseudo-code, we will keep track of the "incumbent" solution, i.e., the pure strategy with best performance observed so far, throughout the algorithm. At termination, the SFP algorithm returns the current and therefore best incumbent solution.

4.2 Remarks

The SFP-like algorithm was first implemented and used as an optimization scheme by Garcia *et al.* [2000], who applied it to a dynamic traffic assignment problem. When compared to previously established methods, the SFP algorithm was able to obtain solutions of the same quality significantly faster. However, it was Lambert *et al.* [2005] who formally introduced SFP and established related convergence results. Based on this work, Lambert and Wang [2003] further demonstrated the effectiveness of the SFP algorithm as compared to simulated annealing for a communication protocol design problem.

We are well aware of the fact that in order to best solve specific applications, empirical tunings, which usually involve domain-specific knowledge, are required. In this thesis,

however, we are interested in proposing SFP as a general approach, so that it can easily be implemented and used on a variety of problems.

The next two chapters address two important issues in using SFP as a general optimization tool. First, given an unknown black-box type objective function with finite discrete variables, we are interested in setting up the problem so that SFP can be used as a standard tool. The following concerns must be addressed in order to achieve this:

- How can we formulate the problem as a game?
- How should we define each player's BESTREPLY function?
- How can we take advantage of the parallel nature of the algorithm?

Second, SFP is by construction an algorithm that only works on unconstrained problems. We are interested in extending it so that it can also be used on constrained optimization problems. Chapter 6 presents a case study on approximating the solution to the stochastic dynamic programming, and it is shown that with proper feasible space transformation techniques, SFP can also be used in solving some constrained problems.

CHAPTER 5

Optimizing Large Scale Simulations by Parallel Computing

As discussed in Chapter 4, we are interested in establishing SFP as a general optimization tool. In this chapter, we look at a case study on the coordinated traffic signal control in a large network. By using this challenging problem as an example, we show important steps in using SFP. Also, we address a critical issue, i.e., parallel implementation, in using SFP on real problems.

This chapter is organized as follows. Section 5.1 introduces the problem of coordinated traffic signal control. We state why it is important, why it is hard, and what we can do about it. Section 5.2 formally describes the coordinated traffic signal control problem, defining terminology and the problem in detail. Section 5.3 presents the coordinated traffic signal control problem in game-theoretic terms, and explains the details of the algorithm's implementation. In Section 5.4, the test case and results of experiments are discussed.

5.1 Introduction

Since Webster and Cobbe [1958] first published their research on pre-timed isolated traffic signal control, significant progress in traffic signal control has been made. With the introduction of advanced computer, control, and communication technologies in traffic networks, signal control systems are now able to receive more network-related information and respond in a more congestion-adaptive manner. From past research, we can see that, in general, the more information a signal controller uses, the better performance it can achieve. However, the complexity of algorithms for designing signal timing plans correspondingly grows as more information is being utilized. Another factor that complicates the problem is the number of signalized intersections considered. In the general case, with non-periodic signal timing plans allowed, the size of the problem grows exponentially as the number of considered signals increases. Therefore in practice, the tradeoff between the accuracy of the algorithm, the amount of traffic-related information used, and the size of the network remains an issue.

Based upon amount of information used in the control schemes, we can classify related research into the following categories:

1. **Offline**: Pre-timed signal control schemes for both isolated and coordinated signal control belong to the offline category. Since pre-timed signal timing plans are computed in an offline manner, they can only use information related to historical flow statistics and network configuration. Webster's method [Webster and Cobbe, 1958] and its extensions, SIGSET [Allsop, 1971], and SIGCAP [Allsop, 1976] are examples of isolated control methods (only a single signalized intersection is considered). MAXBAND [Little, 1966; Little *et al.*, 1981] and its extensions, and TRANSYT [Robertson, 1969] are notable examples of coordinated control methods (a group of signalized intersections is considered simultaneously).

2. **Online**: The use of sophisticated surveillance technologies, including inductive loop detectors and surveillance cameras at signalized intersections, enables traffic signal controllers to make use of real-time traffic information. This information, including, but not limited to, vehicle counts, link volume and link occupancy, proved to be very useful in computing real-time signal timing plans for both isolated and coordinated signal control. Most modern traffic signal control technologies belong to the online category. For the isolated control case, it was Miller [1965] who first proposed a control strategy based on online traffic information. Other more recent methods include SCATS [Sims, 1979], PRODYN [Henry *et al.*, 1983; Henry and Farges, 1989], OPAC [Gartner, 1983; Gartner *et al.*, 2001], UTOPIA [Mauro and DiTaranto, 1989], SPPORT [Yagar and Han, 1994], COP [Sen and Head, 1997]. It should be noted that although many of the above control strategies (e.g., OPAC, PRODYN and SCATS) are also used in coordinated control, the coordinations are mostly done heuristically due to the combinatorial complexity of the problem. Other notable research that focuses on the coordinated control problem includes SCOOT [Hunt *et al.*, 1981], CRONOS [Boillot *et al.*, 1992], REALBAND [Dell'Olmo and Mirchandani, 1995], Lin and Wang [2004], and Heung et al. [2005].

3. **Predictive**: Based on offline and online information, the next promising extension is to come up with predictions of future network congestion, and compute the signal timing plans in anticipation of predicted future traffic conditions. An example of such an approach is RHODES [Mirchandani and Head, 2001; Mirchandani and Wang, 2005]. It uses a combination of current real-time information and planned timing plans from upstream signals to predict future arrivals.

Among these three categories, the control schemes with offline and online information are well-studied and are widely implemented. In comparison, control schemes that are capable of using predictive information are still mostly experimental and researchers are just beginning to explore the benefits of using such information.

The method we propose in this chapter does make use of such predictive information. We rely on information on time-dependent origin-destination flows, which can be

used to predict link congestion in the future. We believe that high quality predictive information will become more and more accessible due to the following two important technological advances. The first important advance is high quality estimation of dynamic origin-destination trip flows [Ashok and Ben-Akiva, 2000, 2002]. The second is the use of vehicle-based GPS systems and other vehicle tracking technology in vehicle routing. With such equipment, we can precisely collect the origin-destination information for the "smart" vehicles (i.e., vehicles outfitted with such equipment). Also, by using these vehicles as traffic probes, we can get better estimates of current link congestions. By combining the above two branches of research, high quality predictive information required by our method should become available. The first goal of the chapter is thus to introduce an algorithm that is capable of incorporating this predictive information in computing adaptive traffic signal timing plans.

Another goal of this chapter is to address the difficulty of finding solutions to the combinatorial problem that arises in general coordinated traffic signal control. The size of the set of solutions that need to be considered grows exponentially as the number of intersections and/or the length of the time horizon considered increases. Moreover, functions typically used to measure performance of the network, such as, for example, average trip time experienced by the drivers, have to be evaluated via computationally intensive traffic simulators. These functions also lack structural properties that traditional optimization algorithms rely upon, calling for novel methods for searching the solution space. Our algorithm allows for parallel execution, which makes real-time signal control possible even in a large network. The applicability of our approach (called **CoSIGN**, for "Coordinated SIGNals") is demonstrated by a test case study based on the real traffic network of Troy, Michigan.

5.2 Traffic Signal Control Problem Formulation

We consider the problem of finding an optimal coordinated traffic signal plan for a group of signalized intersections over a given time horizon. A problem instance is defined by specifying the topology of the traffic network, the time horizon, as well as the time-dependent origin-destination flows over this time horizon. In particular, for every origin-destination pair in the network, the timing of vehicles' departures from the origin for the destination and the route it takes are presumed to be known. The goal is to minimize the average travel time experienced by all drivers in the network during the given time horizon (we use the terms "driver" and "vehicle" interchangeably).

We formulate this coordinated traffic signal control problem as a discrete optimization problem, where the planning horizon is divided into N time periods of equal length of δ seconds, and the decision variables are the *signal phases*[1] prevailing during each of the N time periods, at each of the I signalized intersections. The following notation will be used in describing the coordinated traffic signal control problem:

[1] A signal phase is a collection of traffic movements that receive right-of-way simultaneously. Therefore, all movements within a phase must be non-conflicting.

- $\mathbf{I} = \{1, 2, \ldots, I\}$: set of signalized intersections;
- $\mathbf{N} = \{1, 2, \ldots, N\}$: set of time periods (each time period is δ seconds long);
- $\mathbf{S}_i = \{1, 2, ..., S_i\}$: set of permissible signal phases for intersection i, $i \in \mathbf{I}$;
- $s_{i,n} \in \mathbf{S}_i$: a decision variable representing the signal phase at intersection i during time period n.

The problem can be formally written as:

$$\begin{array}{ll} \min & \textsc{AverageTravelTime}(\{s_{i,n}, i \in \mathbf{I}, n \in \mathbf{N}\}) \\ \text{s.t.} & \\ & s_{i,n} \in \mathbf{S}_i, \forall i \in \mathbf{I}, \forall n \in \mathbf{N} \end{array} \tag{5.1}$$

where the mapping from the vector of decision variables, $\{s_{i,n}\}$, to the objective value is represented by the function $\textsc{AverageTravelTime}(\cdot)$, which reflects the performance measure we discussed above. The dependence of this function on the decisions made in the problem, i.e., the signal timing plans over the planning horizon, is inherently complex and possesses neither analytical representation nor known structural properties (such as monotonicity or subadditivity). In effect, we are faced with a problem of optimizing a "black-box" function. In particular, in our research, all function evaluations are provided by a traffic simulation program, as described in Section 5.3.2.

One immediate concern resulting from this formulation is the exponential explosion of possible joint decisions as N and I get larger. In the worst case, all joint decisions, with number bounded by $(\max_i\{S_i\})^{N \cdot I}$, have to be enumerated and evaluated in order to find an optimal solution to assure global optimality. For a practical size problem, this is impossible. Therefore, we take the approach of searching for a high-quality locally optimal solution instead. Still, considering the complexity and scale of the problem, it is not obvious how even this can be achieved within reasonable time.

5.3 CoSIGN: SFP Algorithm for the Traffic Signal Control Problem

As mentioned above, traffic signal control problems are usually solved by either restricting the space of solutions by searching for parameters of predetermined cyclic patterns, or by limiting the number of signals considerably. Instead, our approach will be to search for solutions to the full-scale coordinated signal planning problem by using the SFP algorithm.

To solve a problem with the SFP algorithm, we must first formulate it as a game. In the following sections, we will describe how to construct a game-theoretic model for the traffic signal optimization problem. Based on this formulation, we can then specify the performance measure used to evaluate signal timing plans and describe the best reply subroutine using this performance measure.

5.3.1 Formulating Coordinated Traffic Signal Control Problem as a Game

With the same notation as defined in Section 5.2, we can formulate the problem as a game:

- **Player:** each tuple (i, n), $i \in \mathbf{I}$, $n \in \mathbf{N}$, is a player. Let **P** be the set of all players, and $P = I \cdot N$, be the number of players.
- **Strategy Space:** for each player $(i, n) \in \mathbf{P}$, its strategy space is the set $\mathbf{S}_i$. Player (i, n)'s decision is denoted by $\mathbf{D}(i, n)$.
- **Payoff function:** by collecting decisions $\mathbf{D}(i, n)$ from all players, a signal timing plan for the planning horizon is formed. By sending this plan to the traffic simulator, we can find the average travel time experienced by all drivers, which is the payoff function value for all players.

5.3.2 Simulation by INTEGRATION-UM

Accurate evaluation of the average travel time can be accomplished by invoking a computer traffic simulator. In our experiment, the simulation is done by INTEGRATION-UM, developed by Van Aerde *et al.* [1989] and modified by researchers at the Intelligent Transportation Systems Research Center of Excellence at the University of Michigan. INTEGRATION-UM is an event-based, mesoscopic deterministic traffic simulator. In order to perform a simulation, we need to provide INTEGRATION-UM with following inputs:

- **Network topology definitions**: the transportation network is modeled as a directed graph in INTEGRATION-UM. To fully specify the network topology, we first define intersections and connection points as the nodes in the graph. There are two types of nodes in INTEGRATION-UM: zone centroids, which can be used as origins and destinations for the vehicle trips, and normal nodes, which can be used as intersections or connecting points. The roads are then defined as directed links connecting these nodes. Important physical properties of each link, including length, capacity, free-flow travelling speed[2], and the signal timing plan and the phase controlling this link (if any), must also be provided.
- **Traffic signal settings**: signal timing plans in the original version of INTEGRATION-UM were assumed to be cyclic. Cyclic plans were specified by parameters that define cyclic patterns, i.e., cycle length, green split, offset, and lost (yellow) time. We modified INTEGRATION-UM in order to take players' joint strategy as input. Note that with a short enough time period δ, the player model can emulate any cyclic pattern. Unlike cyclic plans, the signal timing plans specified

[2]Free-flow travelling speed of certain link is the speed driver experiences when he/she is the only user of that link.

by players' joint decisions incur lost time at intersection i only when players (i, n) and $(i, n+1)$ in two consecutive periods n and $n+1$ have different decisions.

- **Traffic flows**: INTEGRATION-UM assumes that the network is empty at the start of the simulation and all the traffic entering the network is generated by multiple "flows." Each flow, implicitly assumed to consist only homogeneous motorized vehicles, is defined by specifying origin, destination, flow rate (in number of vehicles per hour), and flow starting and ending times. As mentioned in Section 5.1, this information is usually not directly available, therefore we must combine data from several sources, including survey, real time adjustments, and predictions, in order to come up with reasonable estimates. This is where accurate predictive information can really help us. With better predictive information, the simulation will better describe real traffic congestion, and this implies that CoSIGN will be optimizing a more realistic traffic simulation. As a result, for the signal timing plan generated by CoSIGN, the gap between its performance in the simulation and in the real traffic network should also become smaller.

A detailed description of specifications of INTEGRATION-UM can be found in Wunderlich's PhD dissertation [Wunderlich, 1994].

We selected INTEGRATION-UM as our traffic simulator purely on the basis of convenience of implementation, since its source code was readily available to us. We would like to emphasize that since our system architecture is flexible with regard to the type of simulator used, any traffic simulator could have been used here. The only requirement is that it must be able to accept the signal timing plan generated by our algorithm as input, and output necessary information to our solver, as described below.

5.3.3 SFP with Simulation-Based Best Reply Computation

A crucial step in implementing SFP is the computation of best replies in line 5 of Algorithm 4.1. Since for the coordinated signal control problem the objective function can only be evaluated through the execution of the traffic simulator, the only way to accurately compute each player's best reply is by pure enumeration of all player's strategies. In a problem with I intersections and N time periods, best reply computations for all players would generally require $(N \sum_{i=1}^{I} S_i)$ simulations.

In practice the number of simulations can be decreased somewhat by observing the following facts:

1. In line 4 of Algorithm 4.1, a joint strategy **D** is sampled. One can evaluate this strategy (using the simulator) and pass the resulting objective function value as a parameter to the best reply function. Recall that, for each player, best reply is obtained by comparing the objective function values of the sampled joint strategy and the joint strategies obtained by substituting this player's strategy with other elements of its strategy set. Since the value of the former is provided to the best reply subroutine, $(N \cdot I)$ simulations can be saved.

2. Given a sampled joint strategy **D**, there may exist some intersections/time periods when there is only light traffic waiting to pass through. Since the performances of all strategies of the corresponding players are likely to be very close, best reply computations (and hence calls to the simulator) can be skipped for those players. We can define a threshold α, and calculate a best reply for a player (i, n) by invoking the simulator only if its combined traffic volume[3] is greater than α. (In our experiments, we used $\alpha = 0$, skipping best reply computations only when no traffic was traveling through the intersection in a time period.) When the traffic volume is less than or equal to α, the best reply of this player can be essentially selected arbitrarily. To increase the exploration of the joint strategy space, we drew a random strategy uniformly from the player's strategy set in this case.

To take advantage of the second observation, in addition to the objective function value (i.e., average travel time), we need information on the traffic volume at each intersection during each time period, obtained from time-dependent traffic statistics for the sampled strategy. Since this information only needs to be obtained in the beginning of each iteration, we distinguish between executing INTEGRATION-UM in two different modes: mode MAX, where both average travel time and the time-dependent traffic statistics are outputted, and mode MIN, where only average travel time is outputted. (The latter mode is much less time consuming than the former.)

SFP algorithm for the coordinated signal control problem with simulation-based best reply computation scheme described as above will be called **CoSIGN** and used throughout the chapter. The stopping criterion used in **CoSIGN** is the number of SFP iterations.

The pseudo-code for the simulation-based best reply function is listed in Algorithm 5.1. Below is the list of functions used in Algorithm 5.1 (here **D** denotes a joint strategy):

- INTEGRATION-UM$_{\text{MIN}}$(**D**): the function runs the simulation and returns the objective function value.
- INTEGRATION-UM$_{\text{MAX}}$(**D**): the function runs the simulation and returns the objective function value and time-dependent traffic statistics. The objective function value is stored in v, while the time-dependent traffic statistics data are stored in **F**, a matrix where $\mathbf{F}(i, n)$ represents traffic volume at intersection i during time period n.
- RANDOM($\mathbf{S}_i$): the function uniformly picks an element from $\mathbf{S}_i$ and returns it.

The pseudo-code in Algorithm 5.1 implements the ideas discussed earlier. A common evaluation of the simulator in MAX mode is performed in line 1. For each player, if the traffic volume is below the threshold α (as checked in line 4), a phase of the corresponding signal is randomly selected in line 17. Otherwise, the algorithm loops through and evaluates all phases of the signal (except the phase used in **D**, which is already evaluated), starting in line 8.

[3]Combined traffic volume for player (i, n) is defined as the number of vehicles that would drive past intersection i, during time period n, suppose they are given right of way.

Algorithm 5.1: **B**=BESTREPLY(**D**)

```
1:  (v, F) ← INTEGRATION-UM_MAX(D)
2:  for all i ∈ I do
3:     for all n ∈ N do
4:        if F(i, n) ≥ α then
5:           v_min ← v
6:           B(i, n) ← D(i, n)
7:           D′ ← D
8:           for all s ∈ S_i, s ≠ D(i, n) do
9:              D′(i, n) ← s
10:             v_s ← INTEGRATION-UM_MIN(D′)
11:             if v_s < v_min then
12:                v_min ← v_s
13:                B(i, n) ← s
14:             end if
15:          end for
16:       else
17:          B(i, n) ←RANDOM(S_i)
18:       end if
19:    end for
20: end for
21: return B
```

Figure 5.1: Simulation-based best reply function.

Notice that whenever the simulator is executed in either MIN or MAX modes, we will be able to read the performance measures and therefore update the incumbent pure strategy. This best pure strategy will be delivered as the solution at the end of the algorithm execution, as described in Section 4.1.

5.4 Case Study: Troy, Michigan, Network

In order to test performance of the CoSIGN algorithm, we used a realistic traffic network model built by Wunderlich [Wunderlich *et al.*, 2000; Wunderlich and Smith., 1992; Wunderlich, 1994]. This case study model has been constructed based on the real traffic network of Troy, Michigan, and, to ensure fidelity, carefully calibrated against empirical measurements. To maintain this fidelity, we did not modify the model in any way except to insert the signal timing plans we generated. A map snapshot of the Troy network is shown in Figure 5.2. The corresponding model of the network topology is shown in Figure 5.3.

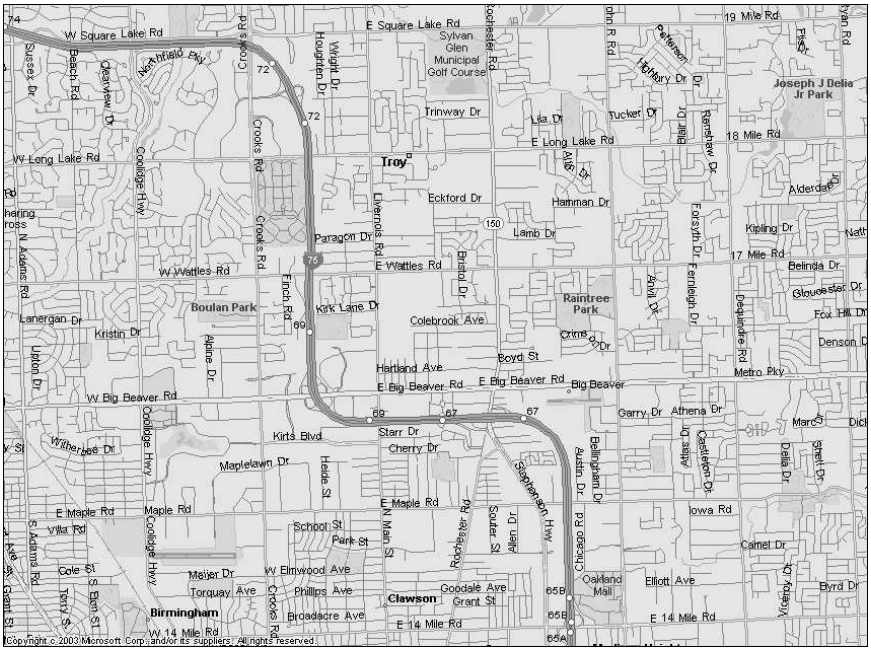

Figure 5.2: The snapshot of Troy's area map.

Here are the parameters used in our experiments:

- Length of the time period: $\delta = 10$ seconds
- Number of time periods: $N = 720$
- Number of signalized intersections: $I = 75$
- Number of players: $P = N \cdot I = 54{,}000$

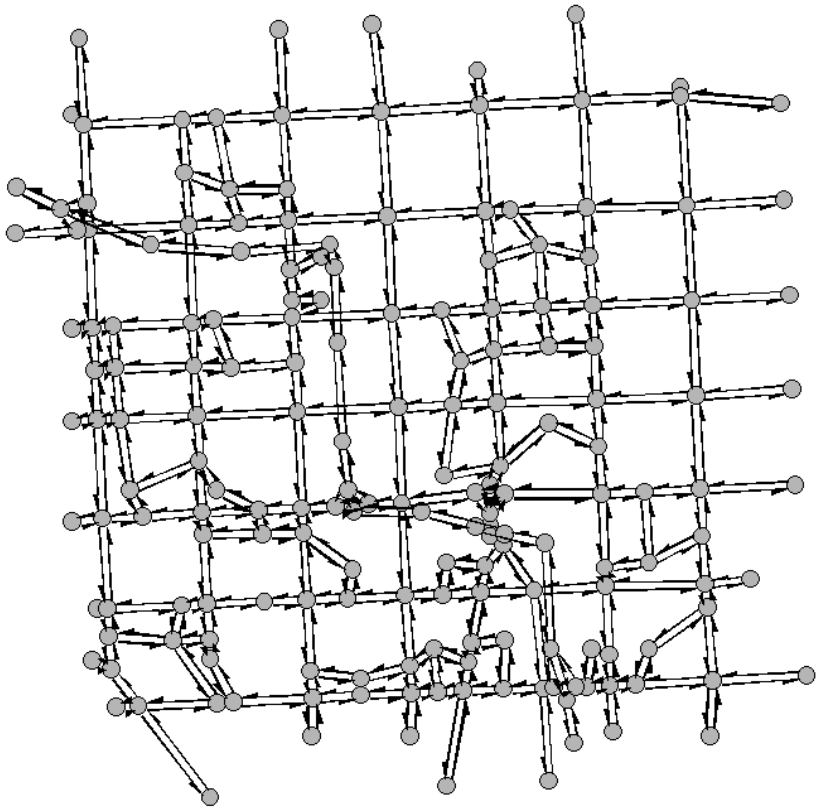

Figure 5.3: The Troy network topology model, composed of 529 links, 200 nodes and 72 zone centroids that can serve as origins or destinations.

- Stopping criterion: 20 iterations of CoSIGN are executed

The original cyclic pattern of traffic signals embedded in the model was used as the initial solution. We assumed that all vehicles will follow fastest free-flow paths[4] from their origins to destinations.

5.4.1 Competing Timing Plans and Algorithms

The goals of this section are twofold: to demonstrate the potential benefits of coordinated traffic signal control using predictive traffic information (as discussed in the Introduction), as well as evaluate the effectiveness of our algorithmic approach, the CoSIGN algorithm, for this task. Towards these goals, we compared CoSIGN to the following alternatives:

- **Static**: fixed cyclic signal timing plans were supplied by the city of Troy and embedded in the original model. When implemented, these signal timing plans were defined by cycle time, offsets, and phase splits. Since real-time signal plan opti-

[4]The fastest free-flow paths are computed with the assumption that free-flow speeds prevail on all links over the planning horizon.

Algorithm 5.4: CD()

1: $\mathbf{D}^0 \leftarrow$ INITIALSOLUTION()
2: $k \leftarrow 0$, $p \leftarrow 1$, $u \leftarrow 1$
3: **while** $u < P$ **do**
4: $\quad \hat{s}_p \leftarrow \text{BESTREPLY}_p(\mathbf{D}^k)$
5: $\quad \mathbf{D}^{k+1} \leftarrow (\hat{s}_p, D^k_{-p})$
6: $\quad$ **if** $\mathbf{D}^{k+1} = \mathbf{D}^k$ **then**
7: $\quad\quad u = u + 1$
8: $\quad$ **else**
9: $\quad\quad u = 1$
10: $\quad$ **end if**
11: $\quad k \leftarrow k + 1$, $p \leftarrow (p \bmod P) + 1$
12: **end while**

Figure 5.4: Coordinate Descent (CD) algorithm.

mization was not available in Troy at the time the model was built, these plans are kept constant throughout the planning horizon.

- **Automatic Signal Re-timing (ASR)**: although real-time signal timing plan optimization was not available in Troy when the model was constructed, the INTEGRATION-UM simulator provides an automatic cycle and phase split optimization tool, which can be used to evaluate the potential impact of such schemes. When the tool is turned on, cycle lengths and green splits at all signals are recalculated at user-specified intervals, using current traffic volume information. For detailed description of this algorithm, refer to Appendix A.

Since static and ASR timing plans control each signal in isolation, the benefits of coordinated signal control can be demonstrated by comparing CoSIGN to static and ASR control schemes. This comparison is conducted in Section 5.4.2.

- **Coordinate Descent (CD)**: a straightforward way to solve a discrete optimization problem of the form (5.1) is to start with some initial solution, loop through all variables (i.e., coordinates) one by one, and solve each single-variable problem while keeping the values of all other variables fixed. The result from the single-coordinate optimization is used to update the current solution. The process stops when a solution cannot be further improved after looping through all variables. In our setting, CD can be formally implemented as follows (here $\mathbf{D}^k$ denotes the joint strategy at iteration k, $(s_p, \mathbf{D}^k_{-p})$ denotes the same joint strategy with the strategy of player p replaced by s_p, and the subroutine BESTREPLY_p evaluates the best reply strategy for player p only):

  The stopping criterion in line 3 of CD is based on the number of consecutive non-improving iterations, u. If $u = P$ (recall that P is the number of variables in this problem), the objective function value cannot be improved after looping through all P variables, and thus we stop.

The CD algorithm by construction considers coordinated signal timing plans, thus we also expect it to enjoy the benefits of coordination, as CoSIGN does. However, CD is a "serial" algorithm in that it considers the variables sequentially, with the output of one single-variable optimization serving as an input into the next one. In a real traffic network (like the Troy network), where the number of variables is large and the time required to invoke a single simulation is non-negligible, the time required to obtain any significant improvement through running CD algorithm may be prohibitively long. To demonstrate the benefits of parallelization, we will explore the possibility of parallel execution of CoSIGN and compare it to CD in subsections 5.4.3 and 5.4.4.

5.4.2 Benefits of Signal Coordination and Predictive Information

Results of experiments comparing CoSIGN to the static and ASR signal timing plans can be seen in Table 5.1. The performance measure is the average travel time experienced by all drivers in the traffic network, evaluated by INTEGRATION-UM. For the *normal-flow case* taken from Wunderlich's model, around 26,000 vehicles were allowed to flow into the network from the beginning of the simulation to the 24^{th} minute mark. This traffic volume, as well as the flow patters used in our experiments, are consistent with the traffic patterns observed in Troy at the time the model was constructed. After the inflow was stopped, the simulator was allowed to run an additional 96 minutes in order to clear all traffic. To evaluate performance under different traffic conditions, we created two similar scenarios, *light-flow case* and *heavy-flow case*, where the same traffic flow pattern and time horizon were used, but the flow rate was decreased (increased) by 50%, so that approximately 13,000 (39,000) vehicles were allowed to flow into the network.

Table 5.1: Performance of three competing algorithms [a].

		Avg. travel time (min.)		
		Light flow	Normal flow	Heavy flow
Static		10.1 (+13%)	19.4 (+29%) [c]	43.8 (+58%)
Best ASR		9.4 (+5%)	17.2 (+14%)	38.2 (+38%)
CoSIGN [b]	Best	8.8	14.9	25.9
	Mean	8.9	15.1	27.6
	Worst	9.0	15.3	29.8

[a] Average travel times are used for performance comparison purpose.

[b] Fifteen independent CoSIGN runs are executed in all flow, scenarios, and best, mean and worst are obtained accordingly.

[c] The number in each cell is corresponding average travel time (in minutes) for that case. The percentages listed in row "Static" and "Best ASR" are margins computed with "CoSIGN — Mean" as base. For example, +29% in Static-Normal flow cell means that the average travel time of static timing plan, under normal flow, is 29% more than that of CoSIGN on average.

Note that as depicted in line 4 of Algorithm 4.1, a random sample is drawn from the history during the beginning of each iteration. This randomness makes CoSIGN a stochastic algorithm. Therefore, to assess performance of CoSIGN, we report summary statistics (mean, best and worst values) of solutions found by 15 independent runs of CoSIGN on each problem instance. Although there is some variability in quality of obtained solutions, stemming from the stochastic nature of the algorithm, CoSIGN finds a signal plan that significantly improves on the starting solution in each instance.

Table 5.1 compares average travel times of signal plans found by multiple CoSIGN executions to that of a static signal plan and the one found by ASR. From Table 5.1 we can see that the plans found by CoSIGN (both on average and even in the worst case) perform better than the other two, under all flow conditions, and the margin of advantage increases as flow gets heavier. Since the static signal timing plan is not adaptive to traffic conditions, this result is to be expected. As for the ASR algorithm, although it is responsive to the real-time traffic condition, its underlying assumption is that the network is undersaturated, and this condition is more likely to be violated in the heavy-flow case than in the light-flow and normal-flow case. This leads to relative deterioration of performance of the ASR approach in the heavy-flow case.

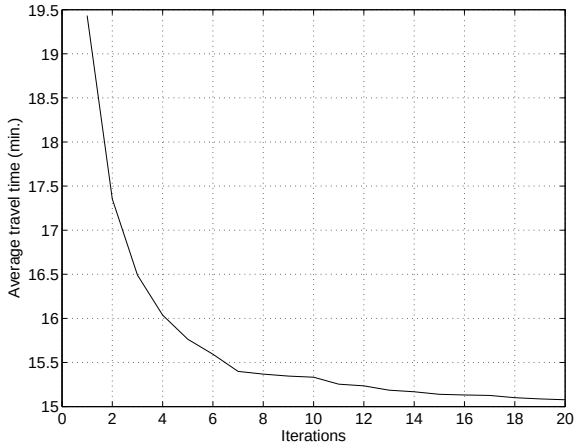

Figure 5.5: The evolution of best values as a function of iteration count for the normal-flow case.

It should also be noted that in the ASR implementation within INTEGRATION-UM, the interval between signal re-timings is a user-specified parameter. Our experiments with various settings of this parameter demonstrated its critical importance to the performance of ASR. Results reported in Table 5.1 reflect the performance of ASR with the re-timing interval that was empirically found to be the best for each experiment. (These "best" intervals had different lengths under different traffic conditions, and we found no discernible pattern of dependence of the method's performance on the interval length; e.g., more frequent re-timings did not necessarily lead to improvements.) In other words, the reported margin of CoSIGN over ASR is a conservative bound, and in practice, with re-timing intervals determined mostly ad hoc, this margin will be much larger.

In Figure 5.5, we plot the evolutions of mean best value (average travel time of current

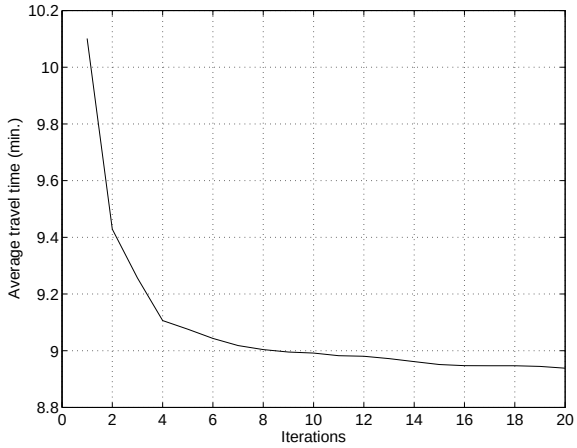

Figure 5.6: The evolution of best values as a function of iteration count for the light-flow case.

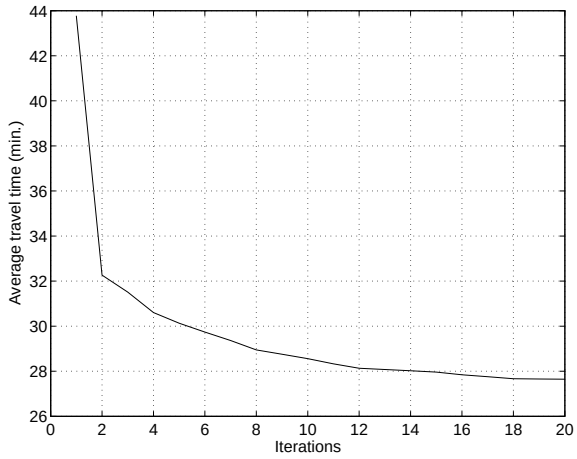

Figure 5.7: The evolution of best values as a function of iteration count for the heavy-flow case.

incumbent solution) versus iteration number for the normal-flow case. Similar evolutions are drawn for the light-flow and heavy-flow cases in Figure 5.6 and Figure 5.7 respectively. Figures 5.5, 5.6 and 5.7 motivate our choice of terminating CoSIGN after 20 iterations: most of the improvements were achieved within the first 10 iterations, and improvements around 20^{th} iteration were small.

Another interesting statistic we observe in these computational experiments is the average travel time experienced by drivers leaving their origins at different times. For all three flow scenarios, we consider 24 groups of vehicles, grouped according to their departure times, where the i^{th} group contains vehicles departing within the i^{th} minute. For each such group, the average travel time of all vehicles in the group is then plotted as a data point. In Figures 5.8, 5.9, and 5.10, average travel times of each group for each control scheme are plotted against all possible departure minutes $(1, 2, \ldots, 24)$. From these figures we can conclude that as flow grows heavier, CoSIGN performs relatively

better than the two alternatives.

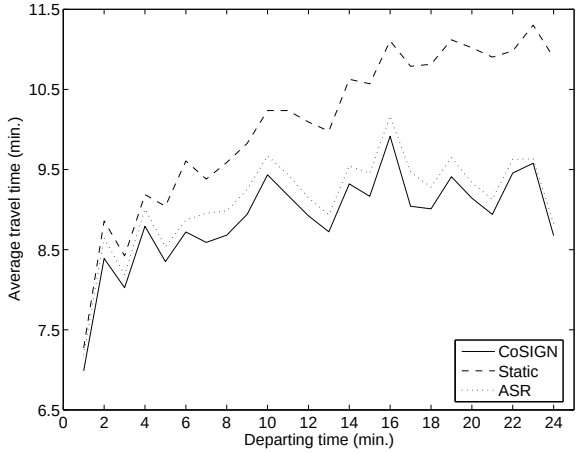

Figure 5.8: Average travel time as a function of vehicles' departing time, for the light-flow case.

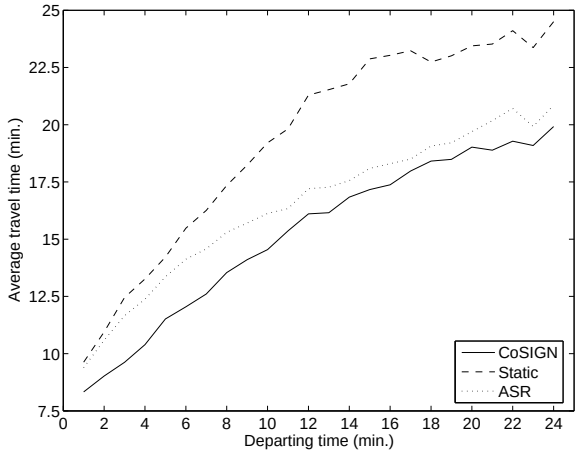

Figure 5.9: Average travel time as a function of vehicles' departing time, for the normal-flow case.

5.4.3 Parallelized Implementation of CoSIGN

We have demonstrated the benefits of a coordinated signal control algorithm that takes into account predictive traffic information in the previous subsection. However, another important consideration is the time required to execute such an algorithm. In a straightforward serial implementation on a Pentium-4 2.8GHz PC with 1GB RAM, running RedHat Linux, 20 iterations of CoSIGN took 169.04 hours for the normal-flow case, and 397.6 hours for the heavy-flow case.

Since CoSIGN is expected to be responsive to current traffic conditions and forecasts, its execution time should be short enough to fit into the desired update interval. One way

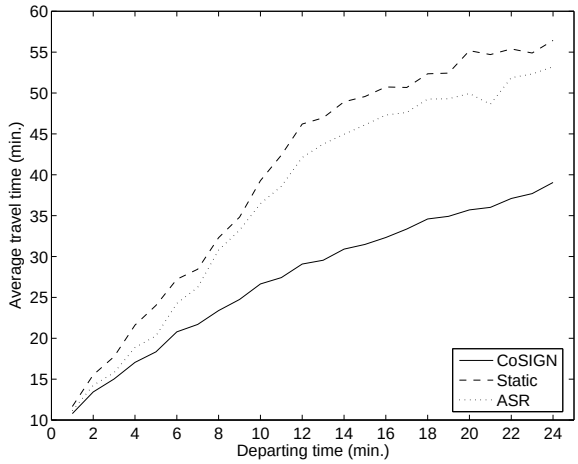

Figure 5.10: Average travel time as a function of vehicles' departing time, for the heavy-flow case.

to significantly reduce the "wall-clock" running time without sacrificing the precision or scope of the solution is through parallelization. In this subsection we will describe how to parallelize CoSIGN and discuss the impact that degree of parallelization has on the running time of the algorithm.

As mentioned earlier, computation between line 2 and line 17 in Algorithm 5.1 can be parallelized. With K identical CPUs available, we can divide the best reply evaluations for all players into K tasks, and assign each task to a CPU. Each task will take the sampled joint strategy, $\mathbf{D}$, its associated objective value, v, and the set of players, $\mathbf{P}_j$, as input parameters. The output of each task will be the best replies, $\mathbf{B}_j$, for players in $\mathbf{P}_j$. Note that since $\bigcup_{j=1}^{K} \mathbf{P}_j = \mathbf{P}$, we have $\bigcup_{j=1}^{K} \mathbf{B}_j = \mathbf{B}$. Regardless of the degree of parallelization, as long as samples drawn in line 4 of Algorithm 4.1 and in line 17 of Algorithm 5.1 remain the same, CoSIGN will evaluate the same set of solutions and return the same output.

In order to asses the impact of parallelization without resorting to repeatedly re-running CoSIGN on clusters of CPUs of various sizes, we instead analytically relate the running time of CoSIGN to the degree of parallelization, and rely on a single run of CoSIGN to make performances estimates.

We will use the following notation:

S_{MAX}:	time required to execute INTEGRATION-UM$_{\text{MAX}}(\cdot)$
S_{MIN}:	time required to execute INTEGRATION-UM$_{\text{MIN}}(\cdot)$
P:	number of players
N_{CoSIGN}:	number of CoSIGN iterations executed ($N_{\text{CoSIGN}} = 20$ in our implementation)
K:	number of available CPUs

In our calculations we neglect time spent on communications between CPUs and sam-

plings in the implementation of CoSIGN since the time spent on simulations dominates total execution time. Also, we assume that at every iteration, K tasks for best reply evaluation are created in a balanced manner, i.e., they require approximately equal time for execution.

In BESTREPLY function, one call to INTEGRATION-UM$_{\text{MAX}}(\cdot)$ and at most $(N \sum_{i=1}^{I}(S_i - 1))$ calls to INTEGRATION-UM$_{\text{MIN}}(\cdot)$ will be made. Let P_T be the number of calls made to INTEGRATION-UM$_{\text{MIN}}(\cdot)$ in one iteration. The wall-clock running time of BESTREPLY function with K CPUs utilized as described above is bounded above by

$$T_{BR} \leq S_{\text{MAX}} + \left\lceil \frac{P_T}{K} \right\rceil S_{\text{MIN}} \tag{5.2}$$

(this is an upper bound since, as discussed in section 5.3.3, best reply computations are skipped for some of the players). Therefore, the total wall-clock running time of N_{CoSIGN} iterations of CoSIGN will be

$$\begin{aligned} T(K) &= N_{\text{CoSIGN}} \cdot T_{BR} \\ &\leq N_{\text{CoSIGN}} \left(S_{\text{MAX}} + \left\lceil \frac{P_T}{K} \right\rceil S_{\text{MIN}} \right). \end{aligned} \tag{5.3}$$

To obtain a tighter bound, let P_s be the average number of simulations actually used per iteration, after we consider the savings described in subsection 5.3.3; we can then replace (5.3) with

$$T(K) = N_{\text{CoSIGN}} \left(S_{\text{MAX}} + \left\lceil \frac{P_s}{K} \right\rceil S_{\text{MIN}} \right) \approx N_{\text{CoSIGN}} \left\lceil \frac{P_s}{K} \right\rceil S_{\text{MIN}}. \tag{5.4}$$

In the Troy test case with normal traffic flows, we observed during a typical run of CoSIGN (with $N_{\text{CoSIGN}} = 20$) $S_{\text{MIN}} = 1.3$ seconds and $P_s =$ 21,582 (note that this is about a 60% reduction in the number of simulations). Hence (5.4) becomes:

$$T(K) \leq 20 \left\lceil \frac{\text{21,582}}{K} \right\rceil 1.3 \text{ seconds} = 20 \left\lceil \frac{\text{21,582}}{K} \right\rceil \frac{1.3}{60} \text{ minutes}. \tag{5.5}$$

For instance, for $K = 134$, 70 minutes of wall-clock computation time will be needed to execute CoSIGN. For $K = 256$, the required time is 37 minutes, and for $K = 1024$ — just 9 minutes. We chose these illustrative values of K since such computational facilities are readily available at educational institutions such as the University of Michigan and University of Texas. To give the reader a broader sense of the impact that different degrees of parallelization have on the wall-clock time required by CoSIGN, we plotted (5.5) in Figure 5.11.

To demonstrate that parallelization is indeed feasible, we implemented a parallel version of CoSIGN on cluster systems managed by the Center for Advanced Computing[5] at the University of Michigan. The specifications of the cluster systems are as follows:

[5]http://cac.engin.umich.edu

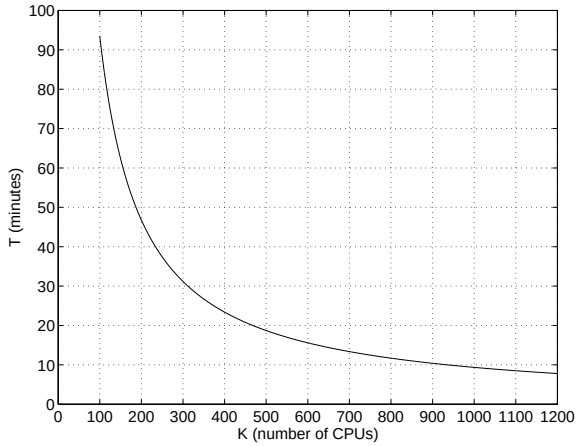

Figure 5.11: Running time of CoSIGN versus degree of parallelization K.

- morpheus: the 208 processor Athlon cluster is composed of 17 nodes of dual Athlon 1600MP CPUs, 29 nodes of dual Athlon 2400MP CPUs, and 58 nodes of dual Athlon 2600MP CPUs.
- nyx: the 450 processor Opteron cluster is composed of 225 nodes of dual Opterons, ranging from Opteron 240s (@ 1400 MHz) to Opteron 244s (@ 1800 MHz).

In our experiments, the typical number of processors used was either 8, 16, or 32, due to the job scheduling policy.

Note that these systems are equipped with CPUs slower than the one we have run our serial experiment on, therefore the curve in Figure 5.11 is not directly applicable. However, a corresponding plot for running time versus degree of parallelization can be easily reconstructed by measuring S_{MIN} on each system.

One of the main assumptions in our derivation is that the time spent on communication can be neglected. We verified this assumption by looking at the timing analysis from our parallel experiments. We observed that in all cases, the percentage of time spent on communication is less than 0.005%. Therefore, at least in our current experiments, the communication time is indeed negligible.

5.4.4 Relative Performance of Parallelized CoSIGN vs. Coordinate Descent

As noted in prior sections, CoSIGN is a heuristic that searches for an optimal solution to the coordinated traffic signal control problem. Although we have empirically shown the algorithm's benefits based on a realistic test case, the solution found in 20 iterations is not guaranteed to be an optimal solution to the problem, even in the local sense. In fact, while the average vehicle travel time in the normal flow case was 15.60 minutes under the signal plan found by CoSIGN, the Coordinate Descent (CD) algorithm described in Section 5.4.1, given sufficient time, found a plan with average time of 13.13 minutes. It

should be noted, however, that it took CD 362,500 iterations over several days of running time to identify this solution.

A meaningful way to compare practical performance of any two heuristic algorithms, such as CoSIGN and CD, on a problem is to compare the objective values of solutions they find given the same amount of wall-clock time. As we demonstrate in this section, as the number of processors made available to CoSIGN increases, its wall-clock running time decreases, and the quality of solutions found by CD in the same time deteriorates dramatically.

As in the previous subsection, we do not resort to multiple algorithm runs, but rather use analytical estimates of running times of CD and CoSIGN to perform the comparison.

Recall that the CD algorithm is initialized with some initial solution, and in each step afterwards, uses a simulation to evaluate the current player's alternative decision. In each of these steps, the solution will be modified if the current player's alternative decision improves the solution. As this process suggests, the CD algorithm cannot be parallelized and must be executed serially. Therefore, the wall-clock time required to execute N_{CD} iterations of CD is

$$(N_{\text{CD}} + 1)S_{\text{MIN}}. \tag{5.6}$$

(We did not invoke the threshold test to bypass potentially unnecessary simulations in CD since that would require running INTEGRATION-UM$_{\text{MAX}}$ at every iteration. Since S_{MAX} exceeds S_{MIN} by 50% to 150%, depending on the number of vehicles in the network, the added computational effort would outweigh potential savings.)

Let $N_{\text{CD}}(K)$ denote the number of iterations CD would be able to perform if it were allowed the same amount of wall-clock time as it takes to execute N_{CoSIGN} iterations of the parallelized CoSIGN algorithm running on a cluster of K processors, i.e., $T(K)$. Setting $(N_{\text{CD}}(K) + 1)S_{\text{MIN}} = T(K)$ and using the formulas above, we obtain:

$$\begin{aligned} N_{\text{CD}}(K) &\leq \frac{N_{\text{CoSIGN}}\left(S_{\text{MAX}} + \lceil P_T/K \rceil \cdot S_{\text{MIN}}\right)}{S_{\text{MIN}}} - 1 \\ &= N_{\text{CoSIGN}}\left(\frac{S_{\text{MAX}}}{S_{\text{MIN}}} + \left\lceil \frac{P_T}{K} \right\rceil\right) - 1. \end{aligned} \tag{5.7}$$

(Recall that $P_T = N\sum_{i=1}^{I}(S_i - 1)$.) Once again, if P_s is the actual average number of simulations used per iteration by CoSIGN, we can obtain a tighter bound:

$$N_{\text{CD}}(K) \leq N_{\text{CoSIGN}}\left(\frac{S_{\text{MAX}}}{S_{\text{MIN}}} + \left\lceil \frac{P_s}{K} \right\rceil\right) - 1. \tag{5.8}$$

In the Troy test case with normal traffic flows, $N_{\text{CoSIGN}} = 20$, and $P_s =$ 21,582, and the numeric form of (5.8) becomes:

$$N_{\text{CD}}(K) \leq 20\left(\frac{S_{\text{MAX}}}{S_{\text{MIN}}} + \left\lceil \frac{21{,}582}{K} \right\rceil\right) - 1 \approx 20\left\lceil \frac{21{,}582}{K} \right\rceil. \tag{5.9}$$

The number of iterations CD will be able to complete in the same amount of wall-clock time as CoSIGN is inversely proportional to the number of processors available to CoSIGN.

As mentioned in the beginning of the section, we did perform one multi-day run of CD for the normal flow scenario in the Troy network. We can now compare the performance of the algorithms as follows: for a particular value of K, we estimate $N_{\text{CD}}(K)$ based on (5.9) and consult the output of the CD run to obtain the average travel time for the signal plan found by CD in $N_{\text{CD}}(K)$ iterations. The resulting comparison is presented in Figure 5.12, where we plot the average travel time of solutions found by CD in $N_{\text{CD}}(K)$ iterations versus K for the normal-flow case. A similar graph for the heavy-flow case is plotted in Figure 5.13. (These graphs may appear a bit counterintuitive at first, as the increase in the number of CPUs results in worse objective function values found. To interpret these graphs, recall that addition of CPUs decreases the amount of wall-clock time allotted to CD, allowing for fewer iterations and less progress.) For comparison, the average travel times of 15.08 minutes (for the normal flow case) and 27.62 minutes (for the heavy flow case) obtained by CoSIGN are also plotted on the same graph. (Recall that these are the mean performance measures of solutions found by several runs of CoSIGN on each problem instance.)

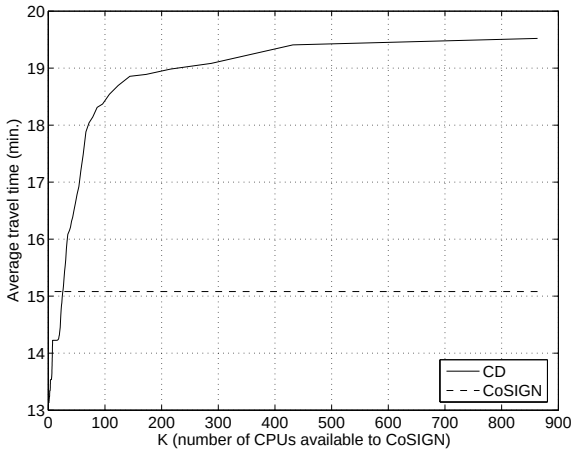

Figure 5.12: Average travel time of solution found by CD when given the same wall-clock time as the parallel execution of CoSIGN with K processors, vs. K.: for the normal-flow case.

As Figure 5.12 indicates, CD underperforms CoSIGN in this comparison if the latter is allowed 26 CPUs or more. Moreover, if CPUs number in the hundreds, CD makes almost no progress from the initial solution in the time it takes CoSIGN to complete its run. Similar result can be observed in Figure 5.13, where CD underperforms CoSIGN in this comparison if the latter is allowed 16 CPUs or more.

Even though in the long (very long!) run CD found a better solution than CoSIGN, since wall-clock times available in practice are limited, the parallelized CoSIGN algorithm will always be superior to CD in practice. Since CD is an inherently sequential algorithm, multiple available CPUs can be utilized by running CD for the specified number of iterations starting at different initial solutions on each CPU and reporting the best solution found. However, based on our empirical experience, CD makes very slow progress in each iteration. Therefore, it will not in fact achieve significant improvement over the

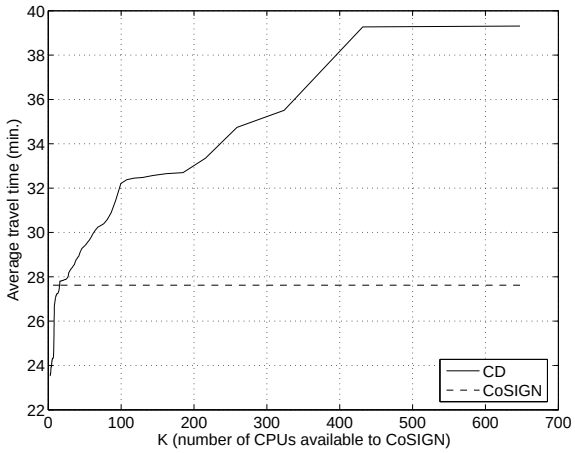

Figure 5.13: Average travel time of solution found by CD when given the same wall-clock time as the parallel execution of CoSIGN with K processors, vs. K.: for the heavy-flow case.

starting points it is provided.

CHAPTER 6

Approximate Large-Scale Dynamic Programming: A Special Case

Chapter 5 suggests a general parallel implementation of SFP algorithm for solving unconstrained discrete optimization problems. However, to solve constrained optimization problems, we have to modify the original SFP procedure. The purpose of this chapter, is to provide an example on how this can be achieved. The benefit of being able to quickly solve large problems later becomes clear when we use the solver repeatedly to solve instances generated by modifying problem data in a controlled manner. It is shown that we can obtain managerial insights by using this numeric approach.

This chapter is organized as follows. Section 6.1 describes the background and the importance of the joint optimization problem in production systems. In Section 6.2, we formulate the joint optimization problem as a Markov decision process. In Section 6.3, we formally state how the game-theoretic approach can be applied to solve the original Markov decision process. In Section 6.4, we discuss the results of numerical experiments and how we can use our approach to develop managerial guidelines. Finally, Section 6.5 concludes the chapter.

6.1 Introduction

Automotive original equipment manufacturers (OEMs) are faced with the challenge of significantly increasing efficiency to offset net vehicle price reductions and increasing benefit costs. At the same time, ever increasing consumer expectations of responsiveness and customization are driving a need for operational flexibility. Management must carefully weight these competing goals when making decisions on capital investments, pricing, and operational policies.

In this chapter, we focus on addressing the problem of optimally investing capital in new production facilities and equipment. Thus, the first key decision to be made is: 1) *What equipment to install?* This involves determining the number, capacity, and flexibility of production lines. These decisions are governed by constraints on available capital

and must factor in forecasts of future demand patterns. Although demand for a vehicle model depends on dynamic exogenous factors such as economic conditions and consumer trends, it can be partially controlled by adjusting the selling price. This introduces the second key decision: 2) *What should be the selling price of each vehicle model?* These prices, combined with the dynamic exogenous economic factors, yield demands for each vehicle model. These demands in turn drive production requirements. Thus the third key decision is: 3) *What are the productions targets?* Note that even if production meets or exceeds demand, it may not always be optimal to fulfill all demands. For example, it may be preferable to stockpile inventory of some models to reap higher selling prices due to seasonality effects. Thus the fourth key decision is 4) *How many vehicles should we sell?* Note: Although OEMs generally do not hold inventory and book revenue as soon as vehicles leave the plant, they do incur some dealer inventory costs through discounted inventory financing. Consequently, the dealer network could be conceptually viewed as an extension of an OEM.

The optimization problem described above is hierarchical in nature, involving decisions at strategic, tactical, and operational levels by different decision-makers. Higher-level decisions constrain and set the context for lower-level, while the potential results of lower-level decisions in turn impact higher-level decisions. Due to their different levels in the decision hierarchy, each decision may have its own horizon, ranging from very long for strategic decisions such as capital investment to quite short for operational decisions such as production levels. The joint optimization problem is extremely complicated, and it is not clear how to make optimal decisions.

To understand the problem abstractly, we will first establish a mathematical model that approximates the joint optimization problem. It should be noted that when formulating the problem, a high level of fidelity is not our top priority as this would require consideration of an inordinate number of uncertainties as well as numerous exogenous, qualitative, and strategic factors. Even if such an optimization problem were tractable, the required data - much of it stochastic in nature – would be exceedingly difficult to collect. Instead, we propose simpler models for which data can actually be obtained, with the goal of generating strategic and operational insights that may be effectively used by decision-makers to improve performance. In order to obtain such insights, it will be desirable to repeatedly solve the problem with controlled problem data, so that we can observe the correlations among important system features. To meet this end, our algorithm must be efficient enough in solving single problem instance, so that within reasonable amount of time, we can collect necessary amount of data for testing various hypotheses about the system.

As we will see in the later sections, even the simplified model we proposed is very difficult to be solved exactly. Thus the first issue we must address is how to efficiently solve the problem, either exactly or approximately. And if the problem is solved approximately, how far is it from the real global optimum.

In practice, as more and more desired features being added to the model, it will eventually become impossible to describe the model analytically and a simulator has to be used. Therefore the second issue we must address is to make sure that the algorithm we choose is capable of optimizing a black-box simulator besides a nicely formed function.

There has been a recent boom in the revenue management-inventory control literature. Research in the past has considered different forms of revenue management. For a recent review on this topic, please refer to Swaminathan and Tayur [2003]. Various researchers have considered adaptive pricing and stocking problems (Alpern and Snower [1988], Subrahmanyan and Shoemaker [1996], and Burnetas and Smith [2000]). Petruzzi and Dada [2002] considered deterministic demand parameterized by one parameter. Chen and Simchi-Levi [2004a,b] considered coordinating pricing and inventory decisions in the presence of stochastic demand over a finite as well as an infinite horizon. Federgruen and Heching [1999], Feng and Chen [2003], and Feng and Chen [2004] considered similar problems. However, to the best of our knowledge there is no literature that focuses on joint optimization of investment, pricing, production and sales. In this chapter we propose to use the game-theoretic paradigm of sampled fictitious play to partly address this issue. To precisely capture the effectiveness of the algorithm in reality, we will include major features of the manufacturing system, but only to the extent that the problem can still be solved to the optimum, so that we can compare the result of the algorithm to the global optimum.

6.2 The Joint Optimization Problem

As described in the introduction, the joint optimization problem is composed of four important decisions. These four decision modules are formally introduced in 6.2.1. The modeling assumptions and the model are described in 6.2.2. Finally in 6.2.3, we point out the complexity of this problem.

6.2.1 Decision Modules

Following the description in Section 6.1, four important decision modules are defined as follows. Note that for simplicity, we assume that the planning horizon is discretized into N periods with equal length.

- **Capital Investment (CI)**: in general, CI module will decide the type (dedicated, reconfigurable, or flexible) and the capacity of the production line. However, to simplify the analysis, we assume that we can only build a dedicated production line that produces only one type of vehicle. Thus, only decision for CI is the production line capacity. Unlike all other modules, where decisions are made at each epoch, the decision on CI is only made at the beginning of the planning horizon, before the first epoch.

- **Revenue Management (RM)**: at the n^{th} epoch ($n = 1, 2, \ldots, N$), the unit price of the vehicle will be decided by the RM module. Note that in the general case where we have multiple vehicle types, a price should be specified for each type. However, since we limit ourself to a dedicated production line that produces only one type of vehicle, our decision for the RM module is just a scalar (instead of a price vector). The pricing decision will then generate the demand for the vehicles

through a demand function (may be deterministic or stochastic).

- **Production Scheduling (PS)**: at the n^{th} epoch ($n = 1, 2, \ldots, N$), the production goal for the current period is decided by the PS module. Note that the production goal cannot exceed the production line capacity decided by the CI module.
- **Sales Planning (SP)**: at the n^{th} epoch ($n = 1, 2, \ldots, N$), the projected sales goal is decided by the SP module. Notice that our sales goal may exceed the real demand in the market, in this case, our real sales will be up to the demand.

6.2.2 The Markov Decision Process

When formulating the model, we would like to include most important features of the problem, while at the same time avoid unnecessary complications. In our investigation, we choose to focus on the stochasticity of the reliability of the production line and the demand function. As discussed in Chapter 3, it is crucial to validate the value of the feature we want to include in the model. In the joint optimization problem we are dealing with here, the validation is straightforward. Since the stochasticity on demand and reliability level will have a direct impact on our decisions on sales, production planning and product pricing, the decisions obtained by ignoring the stochasticity may not even be feasible for particular instantiations of the scenario. Therefore, to construct a satisfactory model, these two features must be included. These two features alone will make the problem non-trivial and difficult numerically.

Assumptions

- The planning horizon is discretized into $N + 1$ periods, $0, 1, \ldots, N$. The capital investment decision is made at period 0. All other decisions, including revenue management, production scheduling and sales, are made at the beginning of all subsequent periods, $n = 1, 2, \ldots, N$.
- We assume that the capacity of the production line can only be chosen from a fixed finite set, and a fixed building cost is associated with each capacity choice. This cost can either be paid by a lump sum deducted in period 0, or it can be paid in installments. In the latter case, we assume that same amount of installment is charged in each period n ($n = 1, 2, \ldots, N$). In our model, we assume that the building cost is always paid in installments.
- All the problem data and decision variables related to the volume of the production are for one shift (8 hours) only. In practice, multiple shifts (usually three, but in the case where additional capacity is needed, a fourth shift can be arranged using weekend time) can be arranged at the production facility, therefore the actual production output may be several times its capacity. However, multiple shifts will only complicate the computation of the cost and production output, without providing much insights into the problem. Therefore, we assume only one shift is used in our model.

- The production line is assumed to be unreliable. Reliability of the production line can be modeled at various operation levels, from micro level to macro level. At micro level, the reliability is modeled at station-level, and the actual production output in each period is collectively decided by the statuses of all stations. Since the interaction among stations can be extremely complicated, in practice we have to use Monte Carlo simulation in order to obtain production output. At macro level, we consider the production line as a whole and assume that its reliability (and thus production output) is governed by a probability distribution. Since we would like to have an analytical expression for the operation of the production line, we will model the reliability at macro level.

- Since the production line is unreliable and breakdown actually happens, we will need to staff the maintenance crew and decide proper maintaining schedule. However, since we are viewing the reliability issue from a macro point of view, the detail on the maintenance of the production line will not be considered in our model.

- The demand function is assumed to be stochastic, reflecting the fact that the market's demand as a function of price cannot be precisely predicted when the pricing decision is made. To simplify the formulation, we assume that we have a finite set of possible demand functions, and for each period, one function will be randomly selected from this set. This set is assumed to be known to the planner.

- No backlog is allowed. If the current inventory plus production is not enough to satisfy the demand in some period, the demand is lost.

- The manufacturing cost depends both on the line capacity and the period when the production occurs.

- The holding cost of carrying i vehicles in the inventory in period n is a fixed fraction of the manufacturing cost for i units of products, supposing that they are to be produced in period n.

Notation

- $\mathbf{N} = \{0, 1, \ldots, N\}$: set of time periods.

- $\mathbf{M} = \{m_1, m_2, \ldots, m_M\}$: set of feasible production line capacities.

- $\mathbf{P} = \{p_1, p_2, \ldots, p_{|\mathbf{P}|}\}$: set of feasible pricing decisions.

- γ: the discount rate.

- $C(m), m \in \mathbf{M}$: the installment to be paid in each period for the initial investment of building a production line with capacity m. $C(m)$ is computed so that if production line is designed to operate for L periods, the discounted sum of L payments equals the lump sum payment of the building cost. i.e., $\sum_{n=1}^{N} \gamma^{n-1} C(m) =$ cost for building line with capacity m.

- $c(n, x^p, x^r, m), n \in \mathbf{N}, m \in \mathbf{M}, x^r \leq x^p \leq m$: the cost of producing x^r units of products in period n, with original production goal x^p and the capacity of m. The portion of production that is planned but cannot be realized due to machine breakdown will not incur material and component cost. However, since the staffing of workers is arranged a priori, the labor cost will still be charged during the breakdown. This implies that the production cost is the sum of two costs: the labor cost, $c_l(n, x^p, m)$, and the material and component cost, $c_m(n, x^r, m)$. $c(n, x^p, x^r, m) = c_l(n, x^p, m) + c_m(n, x^r, m)$.

- ρ_n: As stated in our assumption, the reliability of the production line is modeled in a macro manner. Here we use ρ_n to represent the fraction of available production capacity in period n. By definition, $\rho_n \in [0, 1]$. We assume that in each period, the production line can be operated at one of service levels listed in set **L**, where $\mathbf{L} = \{l_1, l_2, \ldots, l_{|\mathbf{L}|}\}$. We further assume that the probability that the production line operates under certain service level l_k is the same for all period, and will be denoted as P_{l_k}.

- $\mathbf{D} = \{D_1(\cdot), \ldots, D_{|\mathbf{D}|}(\cdot)\}$: the set of possible demand functions. In our model, we assume that each element in **D** is chosen with equal probability. In our model, we assume that the general form of the demand function is exponential, with constant elasticity (we are using similar modeling assumptions as in Hagerty *et al.* [1988]). To simplify the pricing part of the problem, we assume that the only factor that influences the demand is our own pricing decision (thus excluding competitor's pricing and exogenous variables from the demand function). $D_i(p)$ can be formally represented as follows:

$$\begin{aligned} D_i(p) &= e^{\alpha_i} p^{\beta_i} \\ \log D_i(p) &= \alpha_i + \beta_i \log p \\ \alpha_i &\in \{\alpha_1, \ldots, \alpha_a\} \\ \beta_i &\in \{\beta_1, \ldots, \beta_b\}. \end{aligned} \tag{6.1}$$

- $d_n(\cdot) \in \mathbf{D}$: the realized demand function in period n.

- $h(n, i)$: the cost of holding i units of inventory from period n to period $n + 1$. According to the earlier assumption, $h(n, i) = \lambda \cdot c(n, i, i, m)$, where λ is a prespecified constant.

The Model

The problem is a natural sequential decision process, with decisions being made sequentially from period 0 to period N. In period 0, we make capital investment decision m, where $m \in \mathbf{M}$. In period $n \geq 1$, the decisions for RM, PS and SP are made at each epoch. Just as in traditional production control problem, the information required to make optimal decisions for PS, RM and SP is current period and the level of inventory beginning that period. In addition, since decision for CI sets the upper bound on the production, its decision, m should also be required in each period. This enables us to define

the state space for period $n \geq 1$ as the triple, (m, n, i), where m is the capacity of the production line, n is the current period, and i is the inventory entering period n.

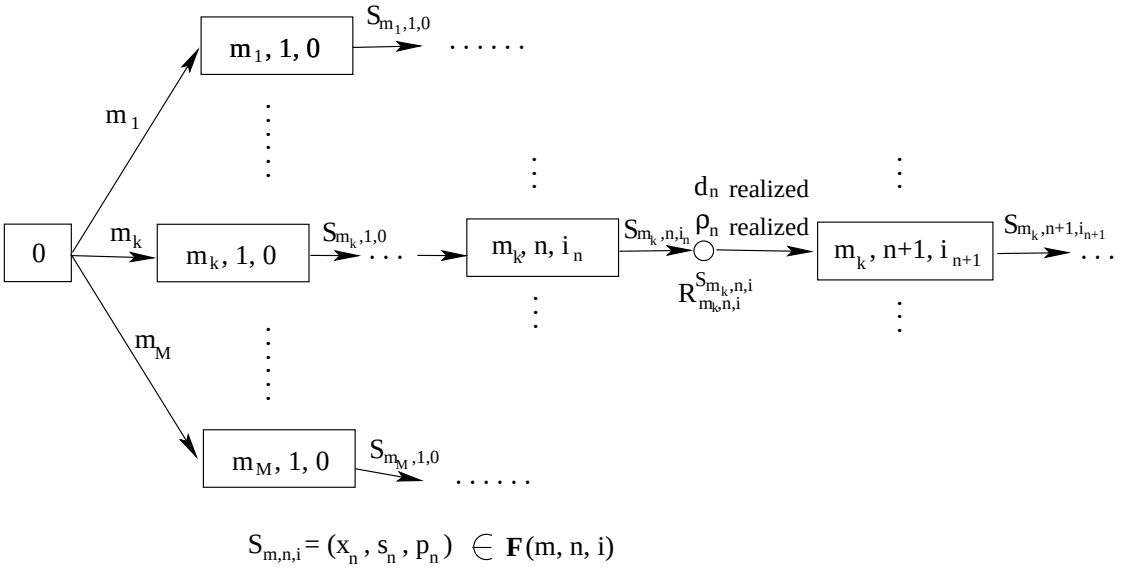

Figure 6.1: The Markov decision model used. $S_{m,n,i}$ is the decision being made at state (m, n, i). $\mathbf{F}(m, n, i)$ is the set of feasible decisions at state (m, n, i) and will be defined later. The demand function, d_n, and the available fraction of the capacity, ρ_n, will be realized after the decision is made. These two realized random variable will then complete the state transition. As ρ_n and d_n realized, the reward, $R^{S_{m,n,i}}_{m,n,i}$, is also generated and accumulated.

After defining the states for the problem, we will define the feasible decisions at each state, the state transition function, the reward function, and finally the functional equation. These important elements of the model are described as follows, and also are illustrated in Figure 6.1.

- At any given state (m, n, i), the set of feasible decisions, $\mathbf{F}(m, n, i)$ is defined by following constraints:

$$\begin{aligned} x_n &\leq m \\ s_n &\leq \min\{i + x_n, \max_{D_j \in \mathbf{D}}\{D_j(p_n)\}\} \\ x_n, s_n &\geq 0, \text{ integer} \\ p_n &\in \mathbf{P}. \end{aligned} \tag{6.2}$$

- The state transition at state (m, n, i), with action (x_n, s_n, p_n), after the realization of ρ_n and $d_n(\cdot)$, is defined by:

$$\begin{aligned} \hat{x}_n &= \min\{x_n, \rho_n m\} \\ \hat{s}_n &= \min\{s_n, i + \hat{x}_n, d_n(p_n)\} \\ i_{n+1} &= i_n + \hat{x}_n - \hat{s}_n. \end{aligned} \tag{6.3}$$

As mentioned in section 6.2.2, we know that $\rho_n \in \mathbf{L}$, $P(\rho_n = l_k) = P_{l_k}$, and each element within **D** is chosen with equal probability. With these definitions and (6.3), we can compute the transition probability, $P^a_{A_1,A_2}$ (the probability of transiting from state A_1 to A_2, if action a is taken), accordingly.

- The reward function at state (m, n, i), with action $S_{m,n,i} = (x_n, s_n, p_n)$, after realizations of ρ_n and $d_n(\cdot)$, is defined by:

$$R^{S_{m,n,i}}_{m,n,i}(\rho_n, d_n(\cdot)) = \hat{s}_n \cdot p_n - c(n, x_n, \hat{x}_n, m) - h(n, i), \tag{6.4}$$

  where $\hat{x}_n$ and $\hat{s}_n$ are as defined in (6.3).

- Functional equation $f(\cdot)$:

  For $n = 0$,

$$f(0) = \max_{m \in \mathbf{M}} \{f(m, 1, 0) - N \cdot C(m)\}. \tag{6.5}$$

  For $n \geq 1$,

$$\begin{aligned} f(m, n, i_n) &= \max_{a \in \mathbf{F}(m,n,i_n)} \mathbf{E}_{\rho_n, d_n(\cdot)} \left\{ R^a_{m,n,i}(\rho_n, d_n(\cdot)) + \gamma f(m, n+1, i_{n+1}) \right\} \\ &= \max_{a \in \mathbf{F}(m,n,i_n)} \sum_{\rho_n \in \mathbf{L}} \sum_{d_n(\cdot) \in \mathbf{D}} \frac{P_{\rho_n}}{|\mathbf{D}|} \left\{ \begin{array}{l} R^a_{m,n,i}(\rho_n, d_n(\cdot)) + \\ \quad \gamma f(m, n+1, i_{n+1}) \end{array} \right\}, \end{aligned} \tag{6.6}$$

  where i_{n+1} can be computed by using (6.3).

It should be noted that in order to drive the model, three important sets of problem data are necessary: the building cost of the production line with different capacity, the set of demand functions, and the manufacturing cost as a function of capacities. The details on the problem data are described later in Section 6.4, when we perform our computational study.

6.2.3 Complexity of the Markov Decision Model

Here we will try to compute an upper bound on the computational effort required in solving functional equations defined above. The required computational effort is measured by the number of *flops*[1] required.

For $n = 0$, the number of flops required is:

$$2M + (M - 1) = 3M - 1.$$

For $n \geq 1$, the number of flops required at each state (m, n, i) is:

$$C_F|\mathbf{L}||\mathbf{D}|(C_T + C_R + 2) + (C_F - 1),$$

[1] *flops* stands for floating-point operations. It is commonly used in providing a measure on the computational complexity.

where C_F represent the size of the feasible decision set $F(m, n, i)$, and C_T and C_R represents the number of flops required to compute state transition and reward function respectively. From equation 6.3, we have $C_T = (2 + 6 + 2) = 10$. From the provided problem data, we can see that the labor cost is linear, the material cost is constant and both costs are stationary. Thus from equation 6.4, we have $C_R = (3 + 3 + 4) = 10$. Therefore for $n \geq 1$, the number of flops required at each state (m, n, i) is:

$$22C_F|\mathbf{L}||\mathbf{D}| + (C_F - 1).$$

For $n \geq 1$, we can compute the range on i for each (m, n) pair: $0 \leq i \leq m(n-1)$. Therefore, for some m_k, the total number of flops to compute $f(m_k, n, i)$, $n = 1, \ldots, N$, $0 \leq i \leq m_k(n-1)$, is:

$$\begin{aligned} & (22C_F|\mathbf{L}||\mathbf{D}| + (C_F - 1)) \sum_{n=1}^{N} (m_k(n-1) + 1) \\ = \; & (22C_F|\mathbf{L}||\mathbf{D}| + (C_F - 1))(m_k N(N-1)/2 + N) \end{aligned}$$

Total number of flops required, can then be computed as:

$$\begin{aligned} & (3M - 1) + \sum_{m_k \in \mathbf{M}} (22C_F|\mathbf{L}||\mathbf{D}| + (C_F - 1))(m_k N(N-1)/2 + N) \quad (6.7) \\ \leq \; & (3M - 1) + (22|\mathbf{L}||\mathbf{D}| + 1)C_F M(m_M N(N-1)/2 + N), \end{aligned}$$

where m_M is assumed to be the largest capacity in $\mathbf{M}$.

The dominating term in equation 6.8 turns out to be $(11C_F|\mathbf{L}||\mathbf{D}|Mm_M N^2)$ and all constants except C_F are well-defined either in the problem data and in the earlier section. To demonstrate how large C_F can be, we look at the extreme case:

$$C_F = m_M D_{\max}|\mathbf{P}|.$$

Substitute it back into the expression of the bound, we have: $(11D_{\max}|\mathbf{P}||\mathbf{L}||\mathbf{D}|Mm_M^2 N^2)$.

From above bound expression, it is obvious that the size of the feasible-action set, C_F, is the key factor that makes this problem hard to solve in practice. This provides the motivation for us to approximate the problem through decomposition along action space.

6.3 Game-Theoretic Model for the Joint Optimization Problem

With the same notations as defined in the previous sections, we can formulate the problem as a game:

- **Players:** each decision module (CI, RM, PS, and SP) is defined as a player. We will use P_{CI}, P_{RM}, P_{PS}, and P_{SP} to represent the player for each of the decision module.

- **Strategy Space:** in the game-theoretic model we proposed, each player must have some probabilistic beliefs about all other players' behaviors. Each realization from such belief on other players' behaviors will create a reduced MDP, where the decision variables are only this player's decision. Therefore the strategy space for each player should be the policy space of this player. However, as defined in equation 6.2, there are interdependencies between players' decision. To be more specific, from equation 6.2, we can see that P_{PS}'s decision relies on P_{CI}'s decision. Similarly, P_{SP}'s decision relies on both P_{PS}'s and P_{RM}'s decisions. Unfortunately, one important requirement for modeling our joint optimization problem as a game is the assumption that all players select their actions simultaneously. Therefore, no matter what kind of beliefs each player has for all other players, it is possible that the combined decisions from all players may be infeasible.

- **Payoff function:** the assignment of payoff values require feasible joint decisions. Therefore, in the case where joint decision is infeasible, the payoff function is not defined.

The feasibility issue mentioned above originates from our attempt to model a constrained problem with an unconstrained model. Thus, we must transform the constrained optimization problem into an unconstrained problem first before putting it in the game-theoretic framework.

In the following section we will describe how to design proper transformation in order to turn our constrained optimization problem into an unconstrained problem that can be modeled by the game-theoretic framework.

6.3.1 Ensuring Feasibility

While implementing a game theoretic method such as fictitious play or sampled fictitious play to solve a complex system optimization problem, one has to ensure feasibility of joint actions by the players. In the case of sampled fictitious play, the concern for feasibility arises from the fact that the algorithm assumes that each player has a finite strategy set that does not depend on actions of other players and therefore, a joint strategy corresponds to a point in a fixed hyper-rectangular subset of the integer lattice. Moreover, players sample their individual actions independently, without knowing what other players have sampled from their respective strategy spaces. This may cause a serious problem if feasibility of a particular action by a player depends on what other players have played. In terms of an optimization problem, the above observations mean that the only allowable constraints are box-constraints with fixed lower and upper bounds on the variables. However, this is rarely the case in most constrained optimization problems including the production systems optimization problem at hand. In particular, the PS player may never decide to produce more than the capacity chosen by the CI player. Similarly, the SP player may never sell more than the minimum of the demand decided by the RM player and the total inventory on hand including the most recent production decided by the PS player. To handle the feasibility issues outlined here, we propose a variable transformation called the *proportional transformation*.

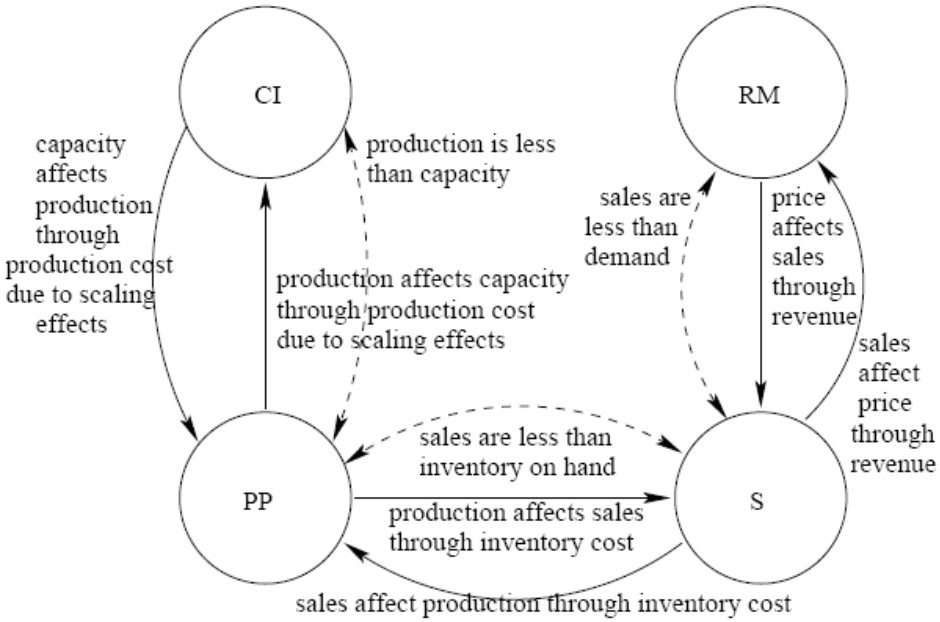

Figure 6.2: Interacting diagram indicating how decision modules affect each other.

The main idea behind this transformation is quite intuitive. Instead of having the PS and the SP players choose the actual production level and the actual sales in a period, we let them choose the fraction of maximum allowable production and the maximum possible sales. Mathematically, instead of letting $x(m, n, i)$ be the decision variable for the PS player, we let $\alpha(m, n, i)$ be the decision variable, where $\alpha(m, n, i)$ is the fraction of realized capacity that is utilized for production. Similarly, we let $\beta(m, n, i)$ be the decision variable for the SP player, where $\beta(m, n, i)$ is the fraction of the minimum of the inventory at hand after production and the realized demand. It is clear from the definition that these two decision variables lie in the interval $[0, 1]$ and no matter what the actual capacity, or inventory or the period is. This helps us transform the optimization problem at hand with complicated side constraints into a problem with box-constraints. Such a problem can be handled by sampled fictitious play after discretizing the interval $[0, 1]$.

One main benefit of the proportional transformation is that it decomposes the decision spaces of various players. In particular, players can choose their own policies without regard to choices made by other players. This can be illustrated as follows. For simplicity, assume that the optimization problem is deterministic, i.e. the machines are reliable and the demand is deterministically set by the price. The joint optimization problem at hand before applying the proportional transformation can then be represented schematically as an interacting diagram shown in Figure 6.2. As illustrated in the figure, a solid line with arrow indicates a particular impact one module has on another module and a dotted line represents that decisions made at two connecting decision modules are mutually constrained.

Obviously, the constrained pairs in Figure 6.2 are the major obstacles in decomposi-

tion, and the purpose of proportional transformation is to break these bonds. Once these bonds are broken, we can then define player from these modules as usual. It is important to note however that even though this representation resolves feasibility issues, the reward of a specific policy employed by a particular player still depends on policies of other players. However, this is relevant only while computing the best replies and not while sampling policies from the empirical distributions.

The proportional transformation is formally defined at each state (m, n, i) for the combined decision $(m, \{\alpha(m,n,i)\}, \{\beta(m,n,i)\}, \{p(m,n,i)\})$ as:

$$\begin{aligned} \tilde{x}(m,n,i) &= \alpha(m,n,i) \cdot m \\ \tilde{s}(m,n,i) &= \beta(m,n,i) \cdot \min\left\{ i + \tilde{x}(m,n,i), \max_{D_j \in \mathbf{D}} \{D_j(p(m,n,i))\} \right\}. \end{aligned} \tag{6.8}$$

Note however that since player CI makes decision on m, during each iteration, when a decision is sampled from player CI's history, it is a specific capacity. This suggests that if we only care about player PS, RM, and SP's best replies against this specific capacity, state variable m is really not necessary and can be removed from the state space. The benefit of doing so is that the computational efforts of computing best replies are reduced by a factor of M (except for player CI). However, if m is removed from the state space, player PS, RM, and SP's best reply are not dependent on m and in the subsequent iterations, it's very likely that the sampled decisions are computed under different capacity than the current sampled capacity from player CI. This constitutes a tradeoff between execution speed and the quality of best replies. While performing the numerical experiments, we tried both approaches. However, in this chapter, we consider only the case where m is removed from the state space. The proportional transformation, after m is removed from the state space, can be written as:

$$\begin{aligned} \tilde{x}(n,i) &= \alpha(n,i) \cdot m \\ \tilde{s}(n,i) &= \beta(n,i) \cdot \min\left\{ i + \tilde{x}(n,i), \max_{D_j \in \mathbf{D}} \{D_j(p(n,i))\} \right\}. \end{aligned} \tag{6.9}$$

The best reply problems for each module is presented in the following sections. In each best reply description, we will describe the version with capacity as state variable and the version without.

6.3.2 Best Reply Problem for the Capital Investment Module

From Figure 6.1, we can see that CI only makes the decision at the beginning of the horizon. In the case where capacity is not part of the state variable, for each $m_k \in \mathbf{M}$ we can compute $f(m_k, 1, 0)$ with other players' policies fixed at $(\{\alpha(n,i)\}, \{\beta(n,i)\}, \{p(n,i)\})$. In this case, CI's problem is just a one-dimensional maximum finding problem that reduces to pure enumeration over all m_k's.

$$m^* = \arg\max_{m_k \in \mathbf{M}} \{f(m_k, 1, 0) - N \cdot C(m_k)\} \tag{6.10}$$

6.3.3 Best Reply Problem for the Production Scheduling Module

Assume that other players' decisions are fixed at $(m, \beta(n,i), p(n,i))$ at each state (n,i). With this given decision and some α, we can compute the transformed point $(\tilde{x}(n,i), \tilde{s}(n,i), p(n,i))$ at each state by using equation 6.9. The state transition and the reward function remain the same. The best reply at each state (n,i) is then:

$$\alpha(n,i) \;=\; \arg\max_{\alpha\in[0,1]} \mathbf{E}_{\rho_n, d_n(\cdot)} \left\{ R^{a(n,i)}_{(m,n,i)}(\rho_n, d_n(\cdot)) + \gamma f(m, n+1, i_{n+1}) \right\}, \quad (6.11)$$

where $a(n,i) = (\tilde{x}(n,i), \tilde{s}(n,i), p(n,i))$.

Note that we need a finite variable domain, therefore $\alpha \in [0,1]$ is actually replaced in implementation with $\{0, \delta, 2\delta, \ldots, 1\}$.

6.3.4 Best Reply Problem for the Revenue Management Module

Assume that other players' decisions are fixed at $(m, \alpha(n,i), \beta(n,i))$ at each state (n,i). With this given decision and some p, we can compute the transformed point $(\tilde{x}(n,i), \tilde{s}(n,i), p)$ at each state by using equation 6.9. The state transition and the reward function remain the same. The best reply at each state (n,i) is then:

$$p(n,i) \;=\; \arg\max_{p\in\mathbf{P}} \mathbf{E}_{\rho_n, d_n(\cdot)} \left\{ R^{a(n,i)}_{(m,n,i)}(\rho_n, d_n(\cdot)) + \gamma f(m, n+1, i_{n+1}) \right\}, \quad (6.12)$$

where $a(n,i) = (\tilde{x}(n,i), \tilde{s}(n,i), p)$.

6.3.5 Best Reply Problem for the Sales Planning Module

Assume that other players' decisions are fixed at $(m, \alpha(n,i), p(n,i))$ at each state (n,i). With this given decision and some β, we can compute the transformed point $(\tilde{x}(n,i), \tilde{s}(n,i), p(n,i))$ at each state by using equation 6.9. The state transition and the reward function remain the same. The best reply at each state (n,i) is then:

$$\beta(n,i) \;=\; \arg\max_{\beta\in[0,1]} \mathbf{E}_{\rho_n, d_n(\cdot)} \left\{ R^{a(n,i)}_{(m,n,i)}(\rho_n, d_n(\cdot)) + \gamma f(m, n+1, i_{n+1}) \right\}, \quad (6.13)$$

where $a(n,i) = (\tilde{x}(n,i), \tilde{s}(n,i), p(n,i))$.

Note that we need a finite variable domain, therefore $\beta \in [0,1]$ is actually replaced in implementation with $\{0, \delta, 2\delta, \ldots, 1\}$.

6.3.6 The Complexity Bound for Solving the Decomposed MDP

We will first find out number of flops required to complete an iteration of SFP. From player CI's best reply expression, number of flops required is the same as $n = 0$ in the global case, i.e. $3M - 1$. For other players, combined flops required at each state is:

$$(C_{\text{PS}} + C_{\text{RM}} + C_{\text{SP}})|\mathbf{L}||\mathbf{D}|(C_T + C_R + 2) + (C_{\text{PS}} - 1) + (C_{\text{RM}} - 1) + (C_{\text{SP}} - 1),$$

where C_{PS}, C_{RM} and C_{SP} represent the number of decisions to be evaluated at each state for player CI, RM, and SP respectively. Term $|\mathbf{L}||\mathbf{D}|(C_T+C_R+2)$ is the effort required in evaluating the expected value of a decision, and terms $(C_{\text{PS}}-1)$, $(C_{\text{RM}}-1)$, and $(C_{\text{SP}}-1)$ are number of comparisons required for player PS, RM and SP.

From the formulations of the best reply problems, we can see that $C_{\text{PS}} = C_{\text{SP}} = (1/\delta + 1)$, and $C_{\text{RM}} = |\mathbf{P}|$. For $n \geq 1$, the upper bound on the number of states is:

$$\sum_{n=1}^{N}(m_M(n-1)+1) = m_M N(N-1)/2 + N.$$

Therefore, total number of flops required for an iteration of SFP is bounded by:

$$\begin{aligned} & (3M-1) + (m_M \frac{N(N-1)}{2} + N)((C_{\text{PS}} + C_{\text{RM}} + C_{\text{SP}})(22|\mathbf{L}||\mathbf{D}|+1) - 3) \\ \leq\ & (3M-1) + (m_M \frac{N(N-1)}{2} + N)(|\mathbf{P}| + \frac{2}{\delta} + 2)(22|\mathbf{L}||\mathbf{D}|+1) \end{aligned} \tag{6.14}$$

The dominating term in above expression is $11(|\mathbf{P}| + 2/\delta)|\mathbf{L}||\mathbf{D}|m_M N^2$. To roughly have an idea about the saving we enjoy with decomposition, we can compute the ratio between the dominating term in the global case and the dominating term here.

$$\begin{aligned} & \frac{11 D_{\max}|\mathbf{P}||\mathbf{L}||\mathbf{D}|M m_M^2 N^2}{11 N_s(|\mathbf{P}| + 2/\delta)|\mathbf{L}||\mathbf{D}|m_M N^2} \\ =\ & \frac{D_{\max}|\mathbf{P}|M m_M}{N_s(|\mathbf{P}| + 2/\delta)}, \end{aligned} \tag{6.15}$$

where N_s is the number of SFP iterations used. After problem data is described in section 6.4, we will compute this ratio and use it as an estimate to possible savings we get from decomposition.

6.4 Vehicle Manufacturing: A Numerical Case Study

In this section, we report detailed results of numerical experiments done using real world data from a major company in the automotive sector.

6.4.1 Problem Data

Recall from section 6.2.2 that the pieces of data required are the plant building costs, stochastic price-demand functions, production costs, inventory costs, and plant reliability data. The general trend in the cost data as plotted in Figure 6.3 was established by discussions with employees of a leading automobile manufacturing corporation. The actual numbers shown in this figure have been purposefully distorted for confidentiality concerns.

- The planning horizon was assumed to be $N = 10$ periods.

- Plant building cost was assumed to be a function of the plant capacity. The cost was amortized over a finite horizon of length $N = 10$, i.e., the horizon used for the optimization problem.

- The price-demand functions were assumed to be exponential, i.e., of the form $D(p) = e^a p^b$. In order to introduce stochasticity, we parameterized demand functions $D_i(\cdot)$ in the set $\mathbf{D}$ by parameters a_i and b_i. In particular, we included three possible demand functions that indicate low demand, normal demand, and high demand. This was achieved by setting $|\mathbf{D}| = 3$, and $(a_i, b_i) \in \{(48.5573, -4.5076), (49.0478, -4.5076), (49.5383, -4.5076)\}$. In each period, the actual realized demand is chosen from one of these three functions with equal probability.

- The variable production cost per vehicle was assumed to decrease with increasing plant capacity due to economies of scale. It was also assumed to be linear in the number of units produced and stationary across time periods.

- The inventory holding cost per vehicle at the end of a period was assumed to be 20 percent of the unit production cost in that period.

- The plant reliability value ρ is assumed to be an element of the set $\mathbf{L} = \{0.6, 0.66, 0.7, 0.74, 0.8\}$. One of these values is selected with equal probability in each period.

- The time value of money was ignored, i.e. the discount factor γ was set to 1.

- α and β were assumed to take values in the set $\{0, 1/300, 2/300, \ldots, 1\}$, i.e., $\epsilon = \delta = 1/300$. To ensure fair comparison between SFP and other alternatives (e.g., a standard MDP solver), we assume that all solution procedures will search within the space of $\mathbf{M} \times \mathbf{A}^N \times \mathbf{B}^N \times \mathbf{P}$.

- 20 iterations of SFP were run on a Pentium 4 (2.8 GHz), 1 GB RAM machine with RedHat Linux operating system.

6.4.2 Experimental Results and Analysis

In our numerical experiments, we looked at the expected values achieved by the policies obtained by both the SFP solver and a standard MDP solver. Also, we looked at computational time required to obtain above policies in both solvers. Although not mentioned earlier, SFP is numerically used as a search algorithm, and a best value and its associated policy will be kept and updated throughout algorithm execution. In our implementation, the best value and associated solution are updated at the end of each best reply evaluation in each iteration.

The comparison results are shown in Table 6.1. Note that for the MDP solver, enumerating all possible capacities cannot be finished in a reasonable amount of time. Therefore, we handpick a capacity which is made to be the optimal capacity by manipulating problem data and try to solve the single-capacity problem. Since the computational effort

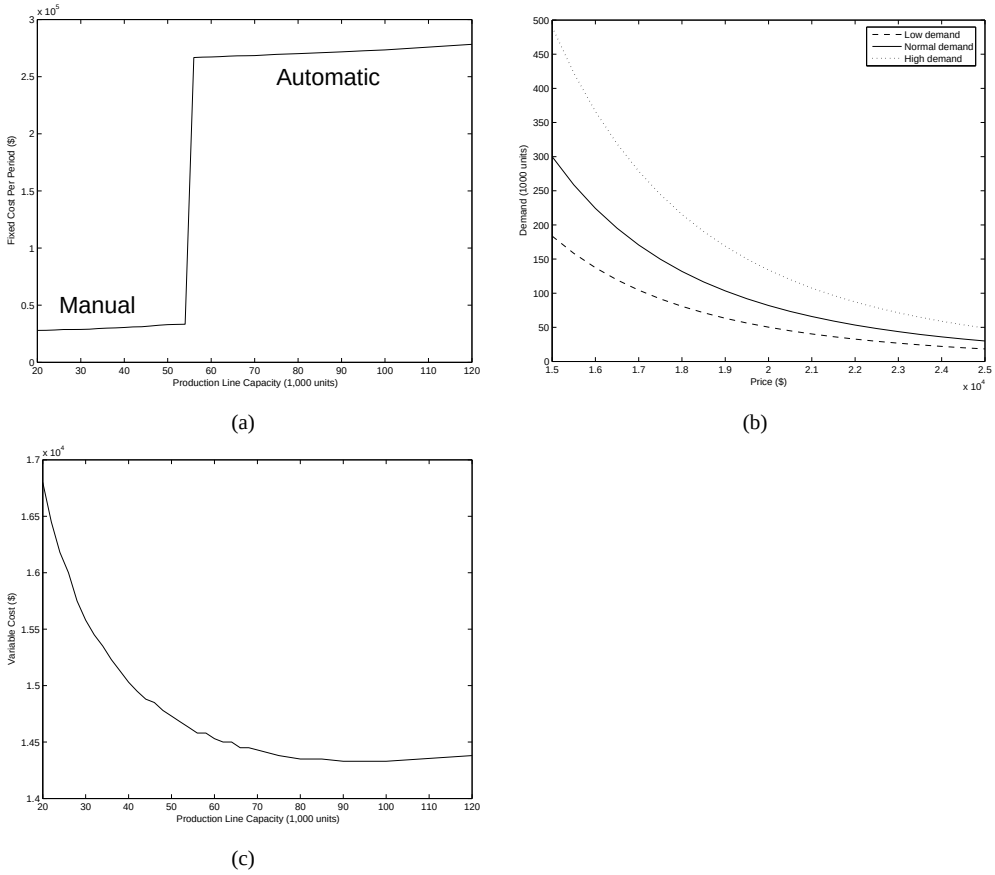

Figure 6.3: Important problem data: (a) Production line building cost, paid by period, as a function of capacity. (b) Demand as a function of price. (c) Variable cost as a function of capacity.

is identical for each capacity, we can estimate the total time required to enumerate all possible capacities. The time required to compute the optimal value for a single capacity is 5,866.3 minutes (or 4.07 days), since we have 33 capacities, the estimated execution time is 193,587.9 minutes (or 134.44 days). SFP solver required 13.1 minutes or was approximately 14,778 times faster than the (estimated) global solver execution time, and the quality of the solution was within 3% of the optimum. The evolution of best values

Algorithm	Execution time	Objective value ratio (versus global optimum)
MDP solver	134.44 days*	1.0
SFP solver	13.1 min.	0.9715

*This execution time is estimated.

Table 6.1: Performances of the MDP solver and the SFP solver

against iterations for the SFP solver is plotted in Figure 6.4. As plotted in Figure 6.4, we can see that the SFP solver makes most improvements during early iterations. In fact, it stops improving after 15^{th} iteration. This empirical finding is why we use 20 iterations as

the stopping criterion for the SFP solver.

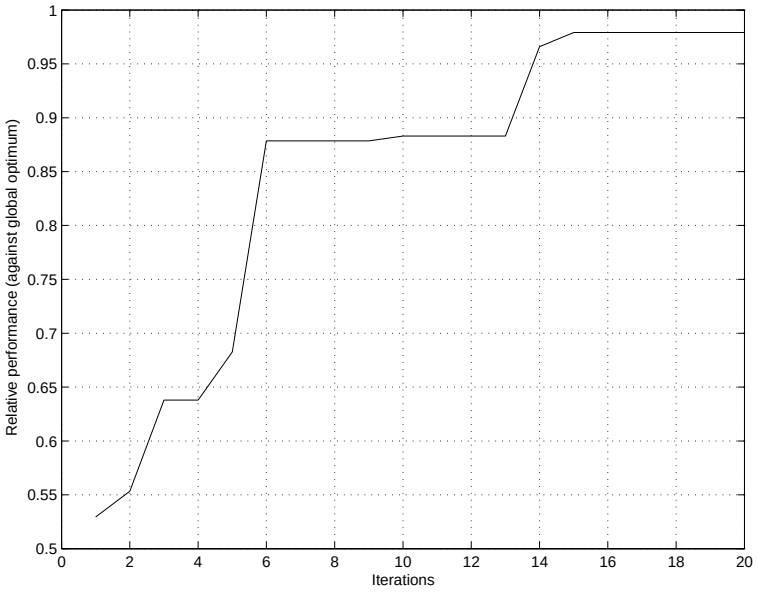

Figure 6.4: Best values plotted against iterations, for the SFP solver.

Notice that since we initiate the SFP solver with some arbitrary initial solution, we can repeatedly restart the SFP solver several times (with different initial solutions) and just keep the best solution in these runs. As an example, if we restart the algorithm 10 times, and randomly generate the initial solution each time, the best objective value can be brought to within 1% of the global optimum. Even in this case, the SFP solver is still about 1,477 times faster than the global solver.

6.4.3 Obtaining Managerial Insights via Optimizations

As mentioned in the introduction, the ultimate goal of this research effort is to take advantage of the speed of the SFP optimization algorithm to develop the understandings on the impacts of key decisions by quickly considering multiple problem scenarios.

As an example, imagine the scenario where we are the production line manager, and we would like to find out the relationship between the reliability of the production line and the associated inventory stocking level. We may accomplish this by solving the integrated problem via the SFP solver for a variety of different reliability levels. Specifically, suppose we consider several different average reliability levels. To reliability level i, we associate the set of service levels $L_i = \{0.20, 0.26, 0.30, 0.34, 0.40\} + 0.05i$. For each reliability level, we approximate an optimal policy by running the SFP solver. With these policies, we can run multiple instances of Monte Carlo simulations on ρ_n and $d_n(\cdot)$, and observe the resulting inventory level in each case. To be more specific, we will run 1,000 instances of Monte Carlo simulations for each reliability level, and compute the average inventory level. Plotting the resulting relationship between mean service level and inventory, we can fit a linear regression equation and use it to predict the average inventory

level for a given reliability. Figure 6.5 illustrates the result of such an analysis, where to speed up execution we set **D**, the collection of demand functions, to be a singleton that includes only the normal demand function. In this case, the computed regression equation is: $I = -20.10r + 20.7924$, where r is the mean reliability level, and I is the average inventory level. Note that the policy used above is selected from a pool of candidate poli-

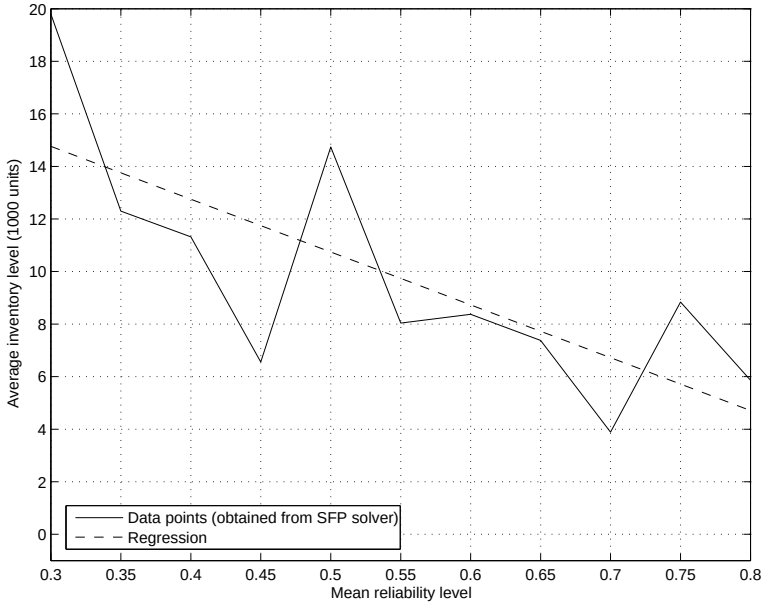

Figure 6.5: Average inventory levels versus mean reliability levels.

cies, all generated by the SFP solver with different initializations. The selection criterion is the objective function value. In other words, we just pick the policy that returns highest expected profit. However, when comparing the average inventory levels of these policies with that of the global optimal policy, we observe that the closeness of objective function values does not imply the closeness of resulting average inventory levels. Furthermore, the policies found by the SFP solver, even with almost identical expected profits, can have very different inventory stocking patterns. This suggests that the inventory stocking level may not be a crucial factor when the expected profit is optimized. As expected, one can see that the inventory level grows almost linearly as the reliability of the production line drops. Also, as reliability level goes over certain level, it becomes optimal to implement a zero-inventory policy.

6.5 Conclusion

In today's competitive environment in manufacturing operations, it is important to make coordinated, near optimal decisions at managerial, strategic and operational levels such as capital investment, revenue management and production planning. The mathematical model of this decision problem is extremely complicated and potentially involves a multitude of exogenous as well as endogenous factors. In this chapter, we presented

a simplified model that captures many of these factors - capital investment, revenue management, production planning, random machine failures, and stochastic demand - yet remains computationally tractable, though still challenging to traditional optimization methods such as dynamic programming. To overcome this computational difficulty, we used the game-theoretic optimization paradigm of Sampled Fictitious Play. SFP has emerged as an effective discrete optimization heuristic for unconstrained problems in the recent past, as demonstrated in Chapter 5. However, to apply it to our manufacturing optimization model, we extended it to handle constraints. This was done by applying a variable transformation to the original dynamic programming formulation to convert it into a finite game in strategic form, making it amenable to SFP. Although illustrated on a specific formulation in this chapter for simplicity and concreteness, we believe that our approach can be generalized to a class of sequential decision problems. In that sense, this approach may be viewed as a heuristic for approximate dynamic programming.

We considered a case study from the automotive manufacturing sector to perform numerical experiments. SFP was able to find near optimal solutions about four orders of magnitude faster than conventional dynamic programming methods when applied to the vehicle manufacturing problem. Since SFP can be parallelized easily, this performance can be further improved. The most important utility of this approach lies in its ability to quickly solve multiple scenarios. The potential of using this tool as a way to develop managerial guidelines is demonstrated in the final section of this chapter. We hope that in the future, this technique can be used to develop data-driven rules of thumb to guide managerial decisions in complex manufacturing operations.

CHAPTER 7

Sampled Fictitious Play: Conclusions and Future Work

The first part of this thesis is devoted to issues related to centralized optimization problems. Chapter 3 focuses on model building process. Chapter 5 focuses on the use of SFP algorithm in general unconstrained black-box optimization problems. Finally, Chapter 6 focuses on the extension of SFP algorithm so that certain class of constrained optimization problems can also be solved with it.

7.1 Summary of Contributions

In Chapter 3, before even going into any particular algorithm, we first discussed important modeling considerations. As model builders, we usually look for completeness and realism. However, as model users, we also want the model to be economic: it should only contain information that is really valuable and absolutely necessary. Before adding any feature to the model, no matter how important it may sound intuitively, we should systematically validate its value. The stochasticity of the model is a good example. Due to the stochastic nature of many real-world problems, it is commonly viewed as a must, and models without it are often viewed as questionable. However, in the specific scenario we studied, we surprisingly found that the value of uncertain information turns out to be zero, implying the redundancy of a stochastic model. This case study provides an example on how valuable simple analysis can be in building models.

Chapter 5 presents a general parallel implementation of the SFP algorithm for solving unconstrained discrete optimization problems. Using pure enumerations in finding best replies is computationally feasible since we can take advantage of the parallel implementation of the algorithm. This capability is shown to be extremely useful in solving the real-world problem of coordinated traffic signal control.

Chapter 6 presents our attempt in extending the SFP algorithm to constrained optimization problems. In particular, we use a joint optimization problem in production systems, modeled as a Markov decision process, for case study. The challenge of this extension lies in the handling of constraints that govern players' interactions, and we proposed a novel feasible space transformation technique to deal with this issue. With

this enhancement, we can solve the problem equipped with real-world data four orders of magnitude faster than the global solver, with satisfactory accuracy (within 3% of the true optimum).

7.2 Future Work

The proposed future work can be classified into two major categories: methodology-related and application-related.

On the methodology development front, extending the SFP algorithm to a more general class of constrained optimization problems remains the most important issue. Other researchers in our research group are also addressing this issue [Ghate *et al.*, 2006] by developing more variants of SFP. However, for a complex real-world problem, given all the available algorithm variations, it remains unclear how we should pick the best one. One interesting idea in addressing this may be to look for the opportunity in borrowing techniques used in other state-of-the-art metaheuristics, like Genetic Algorithms (GA). GA, like the original SFP, is by construction only suitable for unconstrained optimization problems. However, researchers in the GA community have a long history of developing various techniques in dealing with feasibility issues raised when GA is used in constrained optimization problems. With these techniques, GA can be used in many real-world $\mathcal{NP}$-hard problems (e.g., traveling salesman problem). Of course, these techniques are usually highly problem-specific and hard to generalize. However, by reviewing the GA literature on these techniques, we may be able to find inspiration in dealing with specific classes of problems.

One example along this line of thought is the treatment of multidimensional knapsack problems (MKP). Large-scale MKP is an important real-world problem widely studied in the metaheuristics community, GA in particular. In our recent research, we have borrowed the "repair operator" idea in GA, applied it to large MKP test cases, and obtained comparable success.

On the application-related research front, we look to continue our work on traffic-related and production system-related problems.

For our work on CoSIGN, a natural extension is to test CoSIGN on other even larger and more detailed traffic networks. The use of more advanced traffic simulations may also be desirable in modeling more complicated traffic characteristics. All these factors, when combined together, will make an already challenging problem even more so. Of course we can address this issue by throwing in more parallel computing resources, however, this may not be the only way to go. Dell'Olmo and Mirchandani [1996] suggest the use of simplified simulations when the network-wide performance needs to be evaluated frequently. In BESTREPLY, the relative superiority of each player's strategy selections is what we really care about, and a simplified simulation that can accurately provide this relative performance comparison will be good enough, even if in absolute terms it is just an approximation. Of course, in order to take this route, we have to go much deeper into the structure of the problem, and carefully design an approximation scheme. Garcia *et al.* [2000]'s work on dynamic vehicle routing affirmed that using approximate best reply

in SFP is indeed a promising direction. Garcia *et al.* [2000] proposed to solve a dynamic vehicle routing problem by using SFP algorithm. However, the best reply function was constructed by more than just pure enumeration over route alternatives (which explode exponentially in number of nodes). Instead, they proposed to approximate marginal time-dependent link travel times and compute time-dependent shortest paths as best replies. We avoid going into technical details here but the relevant highlights from their work is that by exploiting the problem structure carefully, the use of pure enumeration can be avoided, and they end up requiring only one simulation per iteration.

Another possible future work on the traffic-related application is the combination of both dynamic vehicle routing and coordinated traffic signal control. Much recent research in the field tries to address this issue, however, these attempts have resulted in only limited success, mostly due to the complexity of the problem. With techniques introduced in this thesis, and by applying proper approximation scheme to the best reply computations, we hope to tackle this combined problem.

For our work on production system-related problems, we are interested in the benefit the solver developed in Chapter 3 may have in an industrial setting. When considering all direct and indirect benefits it may bring to the production system, it is estimated that the methodology may bring savings in the scale of hundreds of millions of dollars. It will be a major achievement if such system can be built and deployed.

PART II

Market-Based Approach For Decentralized Resource Allocation Problem

CHAPTER 8

Market-Based Approach: An Introduction

8.1 Motivation

We have already seen in Part I how to optimize complex systems by using SFP algorithms. The use of SFP helps us handle some undesirable properties in optimization problems, e.g., discreteness, ill-structured objective function, and size. However, in some cases, a central optimization may not even be possible due to either or both of following two reasons:

Decentralized control. Authority may be by construction decentralized, such that individual decision makers, or *agents*, have control over respective elements of the overall problem. For example, agents may have discretion over which tasks they perform, or rights over portions of the resources.

Distributed information. Information bearing on possible or preferred allocations may be distributed among the agents. For example, each agent may have its own preferences over task accomplishments, and knowledge of its own capabilities and resources. Such information is generally incomplete, asymmetric, and privately held, so that no central source could presume to obtain it through simple communication protocols.

In these cases, traditional optimization approaches that aim at centralized control cannot be used and we need to focus on designing mechanisms that will encourage independent, self-interested decision makers to act in a way such that the outcome generated by their collective actions is as close to a global optimum as possible. Note that although we are interested in guiding individuals to a global optimum, it doesn't mean that we will try to make individual decision makers collaborate with each other. It is essential for each decision maker to act solely in its own interest.

8.2 Background

8.2.1 Market-Based Resource Allocation

Arguably [Wellman and Wurman, 1998], markets comprise the best-understood class of mechanisms for decentralized resource allocation. In *market-oriented programming* [Wellman, 1993], or *market-based control* [Clearwater, 1995], agents representing end users (those requiring task accomplishments), resource owners, and service providers issue bids representing exchanges or deals they are willing to execute, and the market mediators determine allocations of resources and tasks as a function of these bids. In a well-functioning market, the price system effectively aggregates information about values and capabilities, and directs resources toward their most valued uses as indicated by these prices. As Ygge and Akkermans [1999] put it:

local data + market communication = global control.

Note that in previous studies on market-based approaches, competitive behaviors (meaning that agents take prices as given and neglect their influences on prices) are usually assumed, and as noted by Cheng and Wellman [1998], when certain well-defined conditions are met, classical general equilibrium models can be used to solve general convex-programming problems.

However, in the cases where agents are aware of the influence of their own actions on prices, they may exhibit strategic behaviors, and classical general equilibrium analysis no longer applies. The existent of these strategic behaviors proves to be a major difficulty in designing market mechanisms for decentralized resource allocation problems, because for unbounded agent strategy spaces, it is virtually impossible to evaluate the performance of given market mechanism, let alone choose an optimal one.

Even if we can approximate agents' strategy spaces finitely, predicting agents' behaviors (and identifying associated payoffs for all agents) can still be very hard. This is mostly due to the fact that agents are self-interested and will seek to optimize only their payoff functions. Each agent's optimal decision is a function of other agents' decisions, which are also functions of this agent's decision, ad infinitum. For these scenarios, a solution that is stable in the sense that each agent cannot improve its payoff by deviating unilaterally, will be ideal. As discussed in Chapter 2, such a solution concept is called Nash equilibrium. For each market mechanism, if we can define some collective measure that quantifies overall allocation efficiencies for the NE, it can then be used in evaluating

various market mechanisms.

Finding NEs is a challenging task, especially if each agent's initial preference is characterized by some probability distribution (i.e., the information is incomplete from an individual agent's perspective). To address various issues related to the identification of NEs in practice, we have to perform game-theoretic analysis empirically. Game-theoretic analysis is summarized in the following subsection.

8.2.2 Game-Theoretic Analysis

In order to prepare for the game-theoretic analysis, we will need to specify the payoff matrix that contains payoffs for all agents in each possible joint strategy combination. In our analysis, these payoffs in the payoff matrix are evaluated by running *market games*, where both market mechanism and agent strategies are implemented computationally. In a typical market game, strategies are implemented as software programs (software agents, or just agents) and are initially endowed with random resources and random preferences according to some known distributions. At designated intervals, agents receive information (e.g., prices) from market mechanisms. Based on this information, agents will then perform allowable actions (e.g., bidding). The payoffs for all participating agents will be determined by combined actions over the horizon. From this, we can view each strategy as a mapping from the product of initial information (endowments and preferences) and market information to the actions. Since market information is determined by the interaction of strategies, the actions chosen are ultimately a function of initial information and other agents' actions. In order to capture every detail of agents' interactions, an extensive form game tree must be used. However, to simplify the analysis, we will collapse the extensive form game into a strategic form game by defining payoffs as functions of strategy choices only. To achieve this, we use the probability distribution governing initial information to compute the *expected payoff* for each strategy combination. To evaluate expected payoff computationally, we can draw enough samples from the probability distribution of initial information and execute market simulations for these samples.

8.2.3 Challenges

In this chapter, we motivate the use of markets when decentralization is embedded in the resource allocation problem. Although under some well-defined conditions (e.g., see Cheng and Wellman [1998]), market mechanisms are shown to be ideal devices in guiding resource allocations in a decentralized manner, properly measuring the performance of each market mechanism remains a major challenge. The introduction of Nash equilibrium as a solution concept in market-based resource allocation scenarios aims at addressing this issue. However, setting up market-based resource allocation scenarios for the purpose of identifying Nash equilibria is shown to be a non-trivial task. Various simplifications and techniques are required in order to make game-theoretic analysis possible. More specifically, we must complete following tasks (as noted by MacKie-Mason and Wellman [2006]): (1) choose market mechanism, (2) generate candidate strategies, (3) estimate the resulting "empirical game", (4) solve the empirical game, and (5) analyze the result. This

procedure can be iterative, meaning that the result we get in step (5) can be feedback to step (1) in order to guide the selection of better market mechanism (in terms of allocation efficiency).

This part of thesis will focus on steps (3) to (5). In the following chapters, we will propose some techniques one can use in these steps. And in the concluding chapter, we will use a dynamic task allocation scenario as an example in demonstrating how these steps work in practice.

CHAPTER 9

Market-Based Approach: An Empirical Methodology

9.1 Iterative Mechanism Selection: An Overview

It bears repeating that the motivation of introducing market mechanisms to the decentralized resource allocation problems is our inabilities in controlling these systems centrally. Thus the role of a planner evolves from being a "controller", who seeks optimal control policy, to being a "facilitator", who seeks a set of market mechanisms so that selfish decision makers will be guided to collectively achieve the highest possible allocation efficiency. This chapter will go into details on a series of standard procedures in designing these market mechanisms. In Section 9.2, we introduce a software platform that can be used in simulating market games. In Section 9.3, we highlight some important guidelines in designing agent strategies. In Section 9.4, we discuss issues related to the search for the NE in an estimated empirical game. Finally in Section 9.5, we conclude the chapter and we review some important directions in the study of market-based approaches.

9.2 Simulating Market Games

For all decentralized resource allocation problems we study, there are two major components: 1) agents that represent individual decision makers, and 2) market mechanisms that allow exchange of resources. Due to the decentralized nature of the problem, most agent-specific information, including preferences over tasks, capabilities in performing tasks and resource holdings, are endowed to each agent. Moreover, probability distributions are usually used in describing much of these information to account for uncertainties involved in the problem. This probabilistic representation of the problem makes it very difficult to analytically evaluate the performances of combinations of strategies. To estimate the performances of combinations of strategies, we can define a market game as a collection of agents and market mechanisms, and execute Monte Carlo simulations, in which each agent's related information is generated according to the governing distribution. To support massive simulation efforts, we have developed a software platform that

can be used to provide comprehensive services, including: 1) a general scripting auction engine, AB3D [Lochner and Wellman, 2004], that can be used in defining a wide range of market mechanisms, 2) a general market game engine that can be used in generating market games probabilistically, 3) a set of communication protocols that can be used in designing software agents capable of communicating with components 1) and 2), and 4) a scorer that evaluates performances of all agents after a game ends. In the following paragraphs, we provide more details on these components.

1. **Scripting auction engine**. The idea of designing a flexible software platform for running market game simulations is not new. In fact, AB3D (and also its supporting functions) can be viewed as a redesigned and extended version of the Michigan Internet AuctionBot [Wurman *et al.*, 1998]. Like the AuctionBot but more flexible, AB3D supports a wide range of market mechanisms, specified in a high-level rule-based auction scripting language. The AB3D scripting language exposes parameters characterizing the space of bidding, information revelation, and allocation policies [Wurman *et al.*, 2001]. With proper programming constructs, flow control can also be easily achieved.

2. **Market game engine**. To generate a market game probabilistically, we need to provide both common information and agent-specific information, as described as follows:

   - Common information: this refers to important information agents should know even before the game is actually executed. Most common information is related to the structure of the game, including (but not limited to): i) length of the game, ii) number of agents in the game, and their respective roles, if any (e.g., buyer, seller), and iii) number and type of auctions used in the game.
   - Agent-specific information: in a typical decentralized resource allocation problem, each agent is endowed with information that is only accessible to itself. This information may include task properties (e.g., the value for fulfilling the task, the deadline of the task, and the resource requirement of the task), and initial resource endowment.

   It's not uncommon for the above information to be structured hierarchically (e.g., information can be represented as a tree). To effectively represent and handle such structures, we use XML in describing this information. To support probabilistic game generation, we developed a set of programming constructs, called game description language (GDL), to support basic variable declarations, looping, and random variable generations. A detailed description on GDL is available in Appendix B.

   GDL is general enough to describe a wide class of market games, including TAC classic, a travel shopping game [Wellman *et al.*, 2001], information collection scenarios [Cheng *et al.*, 2004b], job scheduling in reconfigurable production lines [Schvartzman and Wellman, 2006], and dynamic task allocation (in Chapter 11).

3. **Agent interface**. The game system implements a communication interface through which bids, queries, and any other game interactions are transmitted.

4. **Scorer**. For each market game, we must define a procedure to evaluate the performance of each agent on the completion of the game. Scoring typically entails the assembly of transactions to determine final holdings, and for each agent, an allocation of resources to activities maximizing its own objective function. For each agent and the strategy it represents, this score indicates how well it performs in this particular strategy combination for some realization of the agent preferences. It should be noted that scoring mechanism can be highly game-dependent, thus it's up to the developer of the market game to provide the corresponding scorer.

By assembling the above components we have a general environment for executing market games. The interactions of the above components is illustrated in Figure 9.1. For detailed descriptions and a working AB3D market gaming platform, please refer to `http://ai.eecs.umich.edu/AB3D/`.

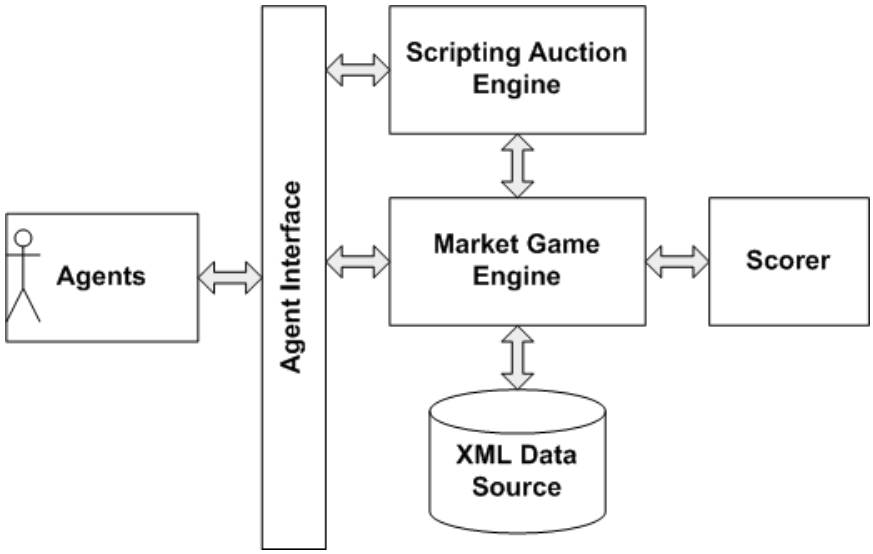

Figure 9.1: General market gaming platform, depicted at functional level.

With this general market gaming platform, we can execute large number of simulations in order to accurately estimate the payoff for each agent strategy in a strategy combination. Note that since it is possible that multiple copies of the same strategy may appear in a strategy combination, when estimating the payoff associated with some strategy, we compute the average payoff for all agents using this strategy, and let the average payoff be the estimated payoff of this strategy.

Also note that when performing game-theoretic analysis, a game with some "strategy ingredient" (a specification on how many of each strategy is used) may be presented in many possible permutations, and these permutations will be viewed as different instances in standard game-theoretic analysis. However, in this thesis, we will assume that market games we studied are symmetric, meaning that the permutation of agents' order will

not be a factor in determining agents' payoffs (e.g., for a game with 4 agents and 2 strategies, A and B, ABAB, AABB, and all permutations having two As and two Bs will be treated as the same game). This simple assumption can greatly reduce the number of strategy combinations we have to consider. Nash's famous result stated that Nash equilibrium exists for every normal form game [Nash, 1950]. For symmetric games, this result holds true as well. However, stronger results can be shown for symmetric games. As a special extension, Nash also showed that symmetric Nash equilibrium exists for finite symmetric games. The existence results in some other special classes of symmetric games are discussed in detail by Cheng *et al.* [2004a].

9.3 Designing Agent Strategies

The definition of agent strategy varies greatly in different contexts. In the context of our market games, an agent strategy is defined as a time-dependent function that takes market information and agent's private information (this may include agent's current resource holdings and agent's preferences) as inputs, and outputs actions that should be taken in the market.

To illustrate the idea, we will use a simple resource allocation problem as an example. Let **R** be the set of resources shared by all agents. For each agent, let **T** be the set of assigned tasks. Let P_j be the current price for resource j, H_j be this agent's holding of resource j, V_i be this agent's valuation on task i, and $M_{i,j}$ be the amount of resource j required for task i. Let **P**, **H**, **V** be the vectors of P_js, H_js and V_is respectively, and **M** be the matrix of $M_{i,j}$s. By definition, **P** is the information obtained from the market, and **H**, **V**, and **M** are agent's private information. It should be noted that some information, e.g., **P** and **H**, may be time-dependent, therefore we add superscript t to indicate price and holding in time period t.

In general, an agent's bids may depend on the whole history of market prices and bids, however, to simplify the construction of the bidding strategy, we assume that each agent's bidding only depends on the current state. Each agent's current state is composed of both market and private information, and agent's bids can be computed by feeding the above information to a bidding function, $\mathcal{F}(\mathbf{P}^t, \mathbf{H}^t, \mathbf{V}, \mathbf{M})$. If prices, task values, and resource requirements are all real numbers, the bidding function is a mapping from $\mathbb{R}^{|\mathbf{R}|^3|\mathbf{T}|^2}$ to $\mathbb{R}^{|\mathbf{R}|}$.

In the following paragraphs, we describe two possible ways of designing and building agent strategies.

Bidding on best package This bidding scheme first solves for the optimal package of resources, given $\mathbf{P}^t$ and $\mathbf{H}^t$. The optimal package includes the amount of additional resources that are required, and how resources should be allocated to tasks. With this optimal package, agent will then place large enough bids so that those required resources can be bought. The problem of finding optimal packages can be

represented mathematically as:

$$\begin{aligned} \max \quad & \sum_{i \in \mathbf{T}} V_i \, x_i - \sum_{j \in \mathbf{R}} P_j^t \, y_j \\ s.t. \quad & \\ & \sum_{i \in \mathbf{T}} M_{i,j} \, x_i \leq H_j^t + y_j, \forall \, j \in \mathbf{R} \\ & x_i \in \{0, 1\}, \forall \, i \in \mathbf{T} \\ & y_j \geq 0, \text{integer}, \forall \, j \in \mathbf{R} \end{aligned} \tag{9.1}$$

where x_i indicates whether task i should be completed, and y_j indicates how many units of additional resource j should be bought from the market (note that no selling is allowed in model (9.1), hence the constraint $y_j \geq 0$). As suggested by model (9.1), the agent will simply place large bids to buy y_j units of resources j. This bidding strategy is common in practice, e.g., a version of this strategy is implemented in Cheng *et al.* [2005b] for a challenging travel shopping game. This strategy may also bear different names, e.g., Greenwald and Boyan [2001] called such problems *completion problems*. This similar strategy is also mentioned in Stone *et al.* [2001].

Bidding on marginal values This bidding scheme first computes the marginal value of each additional unit of available resources; the agent then places bids that match computed marginal values. When computing the marginal value of the resource, we solve model (9.1) repeatedly. In the following paragraph, we use $v(\mathbf{P}^t, \mathbf{H}^t, \mathbf{V}, \mathbf{M})$ to represent the optimal value obtained in model (9.1). With this we can define the marginal value of the n^{th} additional unit of resource j, $m(j, n)$, as:

$$m(j, n) = v(\hat{\mathbf{P}}^t, \mathbf{H}^t + n \, \mathbf{e}_j, \mathbf{V}, \mathbf{M}) - v(\hat{\mathbf{P}}^t, \mathbf{H}^t + (n-1) \, \mathbf{e}_j, \mathbf{V}, \mathbf{M})$$

where $\hat{\mathbf{P}}^t$ is identical to $\mathbf{P}^t$ except $\mathbf{P}_j^t = \infty$, and $\mathbf{e}_j$ is j^{th} unit vector. In words, the above formula says that the marginal value of the n^{th} unit of resource j is the difference between the value of holding exactly n units of additional resource j and the value of holding exactly $(n-1)$ units of additional resource j. The idea of bidding on marginal value has been widely used, for example, see Cheng *et al.* [2005b], Stone *et al.* [2003] and Greenwald and Boyan [2004].

Note that so far we have assumed that $\mathbf{P}^t$ can be directly obtained from the market. However, because of the dynamics of the market mechanisms, current prices usually are not a very good indicator of final prices. This inaccuracy will seriously impact the bids generated by above two schemes. This brings up the need for an accurate prediction of closing prices of auctions. Many possibilities have been investigated in several applications [Stone *et al.*, 2003; Wellman *et al.*, 2004; Zhang *et al.*, 2003; MacKie-Mason *et al.*, 2004; Osepayshvili *et al.*, 2005], and researchers have proposed various ways to improve the quality of price predictions. In this thesis we will assume that price predictions are exogenous and will be provided by a black box.

9.4 Finding Nash Equilibrium in Empirical Games

Given a market scenario, after we have defined players, player strategies, and market mechanisms to use, we can obtain the payoff matrix characterizing this market scenario by executing sufficient number of market game simulations. The next step in the analysis is to compute "solutions" for the market game, i.e., identifying NEs given the payoff matrix.

Significant progress has been made in recent years on the computation of NEs and also associated computational complexity [Conitzer and Sandholm, 2003; Fabrikant *et al.*, 2004; Papadimitriou and Roughgarden, 2005]. In general, the algorithms for computing NEs in a game can be classified into two major categories, the ones that find a sample NE, and the ones that find most (if not all) NEs. Whenever possible, we would prefer methods that can give us as many NEs as possible.

One major issue in NE computation is the exponential growth of the size of the game. Even the simplest n-player game, the one where each player makes a binary decision, requires $n2^n$ values to represent. As demonstrated in Part I, for many practical cases, even storing or loading the game is not possible (e.g., the computational example discussed in Chapter 5 has 54,000 players; even with identical payoff, this implies we have to deal with at least 2^{54000} numbers!). In Part I, we proposed SFP as the algorithm for searching for NEs in large games. SFP is started with no knowledge about the payoff matrix, and a particular payoff value (for a strategy profile) is only evaluated if it is required by some best reply subroutine. This search strategy avoids the need to have a complete payoff matrix before we even begin searching for the NE in the game, thus avoiding this issue. In other words, although the search space is enormous, the search strategy we use selects candidate strategies extremely carefully, with emphasis placed on the most valuable strategy profiles.

Besides this approach, exploiting compact representation of games is also a promising approach in dealing with exponential growth of the game. As discussed by Papadimitriou and Roughgarden [2005], special structures in games, if exploited properly, can assist us in more efficiently searching for NEs. Some particular structures, like symmetry, were well-studied at very early stage of the development of the game theory. As pointed out by some researchers [Papadimitriou and Roughgarden, 2005; Reeves *et al.*, 2005], by simply recognizing the symmetry, a game with n players and k strategies can be represented with only $k\begin{pmatrix} n+k-1 \\ k-1 \end{pmatrix}$ numbers, great reduction compared to nk^n numbers if we don't exploit symmetry. Other notable game structures include graphical games [Kearns *et al.*, 2001], congestion games [Rosenthal, 1973a,b], and local-effect games [Leyton-Brown and Tennenholtz, 2003]. Each of these classes of games describes a particular application domain with certain strong properties. If the scenario studied can be described by any of these games, specialized algorithms that exploit respective structures of these classes of games can greatly improve the efficiency of the solution searching process.

In the following chapters, the primary structure we are exploiting is the symmetry of the game. However, in many cases, this reduction alone may not be sufficient. In those cases, we may want to approximate the solution of the game, by reducing either the

number of players, or the number of strategies. Both ideas are aimed at reducing the size of the game. The size of the game is incrementally reduced until it can be solved properly. As we would expect, the NE found in these reduced games are usually an approximate NE in the original games, i.e., an ϵ-NE. Also, we must note that in the process of game reduction, some NEs may also be eliminated. However, this is the price we have to pay in many cases if we want to solve the game.

All these related issues related to game reduction are discussed in Chapter 10, with particular emphasis on the strategy-reduction technique.

9.5 Conclusion and Related Works

In this chapter we introduced a recently developed set of techniques (under the name "empirical game-theoretic analysis") that can be used for many purposes; in particular, for designing agent strategies, and for designing market mechanisms. These two applications interestingly capture two extremes in the spectrum of the market-based approaches. On the one end, it's individual agents who take environment as given, and try to reason the optimal strategies against other agents (within that particular environment). On the other end, it's the market designer, who tries to select market mechanisms that optimize certain performance measure it cares. These two applications closely relate to each other, since modifications to market mechanisms will change agents' behaviors, and the change in agents' behaviors must be taken into account by the market designer when proposing new mechanisms.

Market mechanisms used in the real world applications usually evolve iteratively. With some market mechanisms initially proposed for certain purposes, agents (participants) then exploit any loophole they can find in order to maximize their own benefits, designer then patches the flaws; this process may repeat for many iterations until the whole system settles down to a stable condition. If there is any change to the environment (e.g., the introduction of new participants, the change to the problem parameters), above adjusting process will repeat again until another stable condition is reached. The merit of the "empirical game-theoretic analysis" is that instead of reacting to what *have* happened, we perform necessary analyses a priori, and propose policies targeted at what *would* happen. Given a decentralized environment, "empirical game-theoretic analysis" provides a way for us to perform computational experiments in order to validate our design. These analyses, if performed properly, can save us from having to make real-time adjustments and could help us avoid making costly mistakes.

There are many recent works on the use of empirical game-theoretic approach on both ends of the spectrum. For the design of market mechanisms, there are works by Vorobeychik *et al.* [2006] and Chapter 11 of this thesis. For the analysis of agents' strategic behavior and efficiency of the game, there are works done by Kiekintveld *et al.* [2006] and Wellman *et al.* [2006]. More details on the use of these techniques can be seen in the example studied in Chapter 11.

CHAPTER 10

Strategy Reduction by Iterated δ-Dominance

10.1 Introduction

As discussed in Section 4.1, finding a NE in a game of realistic size is difficult. Finding all NEs will be even more difficult, and is only possible in fairly small games (e.g., even for 5-player, 5-strategy games, it may take hours, and sometimes days, to solve). However, whenever possible, we would strongly prefer solving for all Nash equilibria.

An immediate thought on how we can solve larger games, as discussed in Section 9.4, is to approximate the game by reducing either the number of players or strategies considered. The idea of reducing the number of players is formalized by Wellman *et al.* [2005a]; the application of this method to a specific market game is described in Wellman *et al.* [2005b]. In this chapter we focus on approaches for reducing the number of strategies.

The idea is directly inspired by the iterative removal of strictly dominated strategies [Luce and Raiffa, 1957; Farquharson, 1969; Moulin, 1979]. A (pure) strategy is strictly dominated if we can find a mixed strategy that performs strictly better than this strategy under all possible combinations of other players' strategies. As a result, these removed strategies cannot be part of any NE. Since the removal of some strategies from an agent's strategy space may result in the removal of other strategies for other players, strict dominance is usually executed iteratively, until no further pruning is possible. One nice property of the process of iterative strict dominance is that any NE in the reduced game is also a NE in the original game.

A weaker version of strict dominance is to allow the pruning of strategies that perform as well as the dominating mixed strategy. These weakly dominated strategies may be part of some NEs in the original game, however, any NE in the reduced game is still a NE in the original game. It should be noted that iterative weak dominance, unlike iterative strict dominance, is *path dependent*, meaning that the set of surviving strategies may depend on the order of eliminations [Gilboa *et al.*, 1990].

An even weaker version of the strict dominance is to allow the dominated strategy to be better than the dominating mixed strategy by a fixed amount δ. This δ-dominated strategy may be part of a NE, and a NE of the reduced game may not necessary be a NE

in the original game, however, it can be viewed as an approximate NE of the original game. Like iterated weak dominance, iterated δ-dominance is path dependent, and furthermore, with every iteration executed, more error will be accumulated. In this chapter, we relate the execution of the δ-dominance to the error bounds on NEs obtained in the reduced game. Also, we propose a simple heuristic for determining the order of strategy elimination. We also explore the benefit this method can bring to the empirical game theoretic analysis.

This chapter is organized as follows. In Section 10.2, we formally define the procedure of iterated δ-dominance, and we discuss the error bounds on NEs in reduced games. In Section 10.3, we go into details on how one would implement iterated δ-dominance in practice, and we provide a simple implementation suggestion. In Section 10.4, we use a challenging empirical game from the trading agent competition community to demonstrate how our procedure can help in solving real games. Finally, in Section 10.5, we conclude our work.

10.2 Iterated δ-Dominance and Equilibrium Approximation

Before we go into details of the procedure, we first define *δ-dominance* for a pure strategy. In the rest of this chapter, we follow the notation defined in Chapter 2.

Definition 10.1 *Let $\boldsymbol{S}_i$ be the finite set of pure strategies for player i, and $\Delta(\boldsymbol{S}_i)$ be the space of mixed strategies for player i. We define strategy $s_i^1 \in \boldsymbol{S}_i$ as δ-dominated if $\exists\ \sigma_i^1 \in \Delta(\boldsymbol{S}_i^1)$, $\boldsymbol{S}_i^1 = \boldsymbol{S}_i \setminus \{s_i^1\}$ such that:*

$$\delta + u_i(\sigma_i^1, s_{-i}) \geq u_i(s_i^1, s_{-i}), \forall s_{-i} \in \boldsymbol{S}_{-i}. \tag{10.1}$$

In other words, s_i^1 is δ-dominated if we can find a mixed strategy (on the set of pure strategies excluding s_i^1) that, when compensated by δ, is at least as good as s_i^1 against all pure opponent strategies. Note that unlike the standard dominance definition, for each pure strategy (s_i^1) we check, we must exclude it from the domain of $\Delta(\cdot)$. This modification is necessary because if we don't exclude s_i^1, it will be δ-dominated by itself.

Because we introduce δ when eliminating strategies, eliminated strategies may in fact be part of some NE. As a result, the NE computed in the reduced game may only be an approximate NE in the original game. In this section, we examine the effect of multiple iterations of δ-dominance has on the quality of obtained NEs, relative to the original game. We first state how error is accumulated with one iteration of δ-dominance.

Proposition 10.2 *Let Γ^n be the original game and let s_i^n be δ-dominated in Γ^n. Let Γ^{n+1} be the game obtained by removing s_i^n from Γ^n. If any unilateral deviation by a player from a mixed strategy can only result in at most ϵ improvement in its payoff, it is called an ϵ-equilibrium. If σ is an ϵ-equilibrium in Γ^{n+1}, then it is a $(\delta + \epsilon)$-equilibrium in Γ^n.*

Proof.
Since s_i^n is δ-dominated in Γ^n, $\exists\ \sigma_i^n \in \Delta(\mathbf{S}_i^n)$, where $\mathbf{S}_i^n = \mathbf{S}_i^{n-1} \setminus \{s_i^n\}$, such that:

$$\delta + u_i(\sigma_i^n, s_{-i}) > u_i(s_i^n, s_{-i}), \forall s_{-i} \in \mathbf{S}_{-i}. \tag{10.2}$$

Also, since σ is ϵ-equilibrium in Γ^{n+1} (which implies $\sigma \in \Delta(\mathbf{S}_i^n)$), we have:

$$\epsilon + u_i(\sigma_i, \sigma_{-i}) \geq u_i(s_i, \sigma_{-i}), \forall s_i \in \mathbf{S}_i^n. \tag{10.3}$$

From (10.2), we can write:

$$\delta + u_i(\sigma_i^n, \sigma_{-i}) > u_i(s_i^n, \sigma_{-i}), \tag{10.4}$$

since (10.2) is true for all $s_{-i} \in \mathbf{S}_{-i}$, arbitrary linear combination on s_{-i} will also satisfy the inequality.
Since σ_i^n and σ_i both belong to $\Delta(\mathbf{S}_i^n)$, and σ_i is part of the ϵ-NE in Γ^{n+1}, from (10.3), we have:

$$\epsilon + u_i(\sigma_i, \sigma_{-i}) \geq u_i(\sigma_i^n, \sigma_{-i}), \tag{10.5}$$

again, since (10.3) is true for any $s_i \in \mathbf{S}_i^n$, arbitrary linear combination on s_i will still satisfy the inequality.
From (10.4) and (10.5), we have:

$$(\delta + \epsilon) + u_i(\sigma_i, \sigma_{-i}) > u_i(s_i^n, \sigma_{-i}), \tag{10.6}$$

from (10.6) we can see that σ is indeed a $(\delta + \epsilon)$-NE in Γ^n. ∎

We are now ready to define a theoretic upper bound on errors after several iterations of δ-dominance.

Proposition 10.3 *Let Γ^n be the game after n iterations of δ-dominance from the original game Γ^0. We assume that one strategy is eliminated with δ_i in each iteration i. Let Γ^0 be the original game. Then an ϵ-NE obtained in Γ^n is a $(\sum_{i=1}^n \delta_i + \epsilon)$-NE in Γ^0.*

Proof.

From Proposition 10.2, we know that the statement is true for $n = 1$. Assume that the statement is true for $n = n_1$, then the ϵ-NE in Γ^{n_1} is $(\sum_{i=1}^{n_1} \delta_i + \epsilon)$-NE in Γ^0.

Now we would like to show that the statement is also true for $n = (n_1 + 1)$.

Note that since Γ^{n_1} is reduced from Γ^0 after n_1 iterations of δ-dominance. The statement for $n = n_1$ should hold for any pair of Γ^p and Γ^q, as long as Γ^p is obtained from n_1 iterations of δ-dominance from Γ^q.

Therefore from above claim, the ϵ-NE in Γ^{n_1+1} is $(\sum_{i=2}^{n_1+1} \delta_i + \epsilon)$-NE in Γ^1. However, from Proposition 10.2 we know that $(\sum_{i=2}^{n_1+1} \delta_i + \epsilon)$-NE in Γ^1 is $(\sum_{i=2}^{n_1+1} \delta_i + \delta_1 + \epsilon)$-NE in Γ^0. Thus the statement is also true for $n = (n_1 + 1)$.

From math induction, the proposition is proved. ∎

10.3 Implementation of Iterated δ-Dominance

Every time we use a δ to dominate certain strategy, we are adding errors to the solution (from Proposition 10.2). Therefore, given a "budget" for errors we would like to endure, we are interested in how to distribute it over several iterations of δ-dominance (one iteration is also possible), so that we can reduce the size of the game most.

If we define the original set of strategies and all its subsets as nodes, then we can pose following two questions: (1) what is the minimal δ that brings us from one node to another node? (2) given a starting node and some δ, for all nodes with distances less than δ from the starting node, which node is smallest in terms of set size?

To answer above two questions, we must first address following fundamental questions:

- When will an arc exist? Obviously, according to the definition of nodes, for an arc to exist between two nodes, it is necessary that one node is a proper subset of another node, and this arc should originate from the node representing superset to the node representing subset.
- What's the definition of arc cost? An arc connecting two nodes represents the action of performing a single iteration of δ-dominance, and the starting node and ending node represent the original set and the set after dominance respectively. From this definition, the arc cost can be naturally defined as the minimal δ required to complete this action.

From these discussions, we can see that the first question we raised earlier can be posed as a shortest path problem in the graph. Similarly, the second question can also be posed as a collection of shortest path problems.

Although shortest path problems are well-studied and can be solved efficiently, the primary difficulty in our case is to come up with arc costs. As we will see later, computing arc costs, although possible, is non-trivial. Since number of nodes and number of arcs grow exponentially with number of original strategies, it quickly becomes intractable to come up with complete arc costs. Therefore, in realistic cases, solving for shortest path can not be performed (again, due to difficulty in acquiring problem data).

In the following sections, we first formulate the arc cost computation problem as a linear program, and use it as a sub-routine in developing a path finding heuristic.

10.3.1 Finding Minimal δ That Dominates Subset of Strategies

Definition 10.1 provides the definition for δ-dominance on a pure strategy. We will now extend it so that we can define δ-dominance on a set of strategies.

Definition 10.4 *Let $\boldsymbol{S}_i$ be the finite set of pure strategies for player i, and $\Delta(\boldsymbol{S}_i)$ be the space of mixed strategies for player i. We define a set of strategies $\boldsymbol{T} \subset \boldsymbol{S}_i$ are δ-dominated if for each $t \in T$, $\exists\ \sigma^1_{t,i} \in \Delta(\boldsymbol{S}^1_i)$, $\boldsymbol{S}^1_i = \boldsymbol{S}_i \setminus T$ such that:*

$$\delta + u_i(\sigma^1_{t,i}\ , s_{-i}) \geq u_i(t, s_{-i}), \forall s_{-i} \in \boldsymbol{S}_{-i}. \tag{10.7}$$

Following Definition 10.4, we can construct an optimization problem that identifies the δ that dominates a set of strategies $\mathbf{T}$. The formulation is listed in Figure 10.1.

$$
\begin{aligned}
\text{LP-A}(\mathbf{S}, \mathbf{T}): \min \quad & \delta \\
\text{s.t.} \quad & \\
& \delta + \sum_{s \in \mathbf{S}_i \setminus \mathbf{T}} x_t(s) \cdot u(s, s_{-i}) \geq u(t, s_{-i}), \forall\, t \in \mathbf{T}, \forall s_{-i} \in \mathbf{S}_{-i} \\
& \sum_{s \in \mathbf{S}_i \setminus \mathbf{T}} x_t(s) = 1, \forall\, t \in \mathbf{T} \\
& 0 \leq x_t(s) \leq 1, \forall\, t \in \mathbf{T}, \forall s \in \mathbf{S}_i \setminus \mathbf{T}
\end{aligned}
$$

Figure 10.1: LP-A(**S**, **T**): formulation for finding δ that dominates **T**, a set of strategies.

10.3.2 A Greedy Heuristic for Forming Domination Path

As mentioned at the beginning of Section 10.3, the major difficulty for finding shortest path in the strategy reduction graph has been the computations of arc costs. Therefore in practice, instead of computing all arc costs (which is computationally prohibitive), we would like to find a simple rule for identifying promising arcs, and compute costs only for these identified arc. Based on computed arc costs, we will then decide strategies that should be pruned.

In this section, we propose a simple iterative greedy heuristic for identifying the order in which strategies should be pruned. At the beginning of each iteration, strictly dominated strategies are first removed, then for each surviving strategy, the δ required to eliminate it is computed using LP-A($\cdot$). The heuristic is greedy because it prunes the strategy with least δ in each iteration. This simple greedy heuristic is described in Figure 10.2. Two input parameters are required: **S** is the initial set of strategies, and Ω is our "budget" for errors.

A simple variant that prunes k strategies in one iteration can be extended from Algorithm 10.2. We use each strategy's associated δ to determined k strategies with least δs. We then group them into a set **K**, and use LP-A(**S**, **K**) to find the real $\delta_{\mathbf{K}}$ that can prune them within one iteration. The general heuristic is described in Figure 10.3. Of course, if we actually compute δ for all subsets with size k, **K** may not be the one with least δ. However, in order to identify such set, exponential number of enumerations is required, and this is impractical. Also note that after set **K** is identified, the real error subtracted from Ω will not be $\sum_{i \in \mathbf{K}} \delta(k)$, instead, it will be $\delta_{\mathbf{K}}$ computed using LP-A($\cdot$). This is exactly why we introduce GREEDY-K: eliminating multiple strategies at once may incur less error compared to eliminating them one by one. In the next section, we will introduce a way to compute a tighter bound on error once we obtain a reduced game.

ALGORITHM 10.2: GREEDY($\mathbf{S}$,Ω)

$n \leftarrow 1,\ \mathbf{S}^n \leftarrow \mathbf{S}$
while $\Omega > 0$ **do**
  for $s \in \mathbf{S}_i^n$ **do**
    $\delta(s) \leftarrow$ LP-A$(\mathbf{S}^n, \{s\})$
  end for
  $t \leftarrow \arg\min_{s \in \mathbf{S}_i^n} \delta(s)$
  $d \leftarrow \delta(t)$
  if $\Omega \geq d$ **then**
    $\Omega \leftarrow \Omega - d$
    $\mathbf{S}_i^{n+1} \leftarrow \mathbf{S}_i^n \setminus \{t\},\ \mathbf{S}^{n+1} \leftarrow (\mathbf{S}_i^{n+1}, \mathbf{S}_{-i})$
    $n \leftarrow n + 1$
  else
    $\Omega \leftarrow 0$
  end if
end while
return $\mathbf{S}^n$

Figure 10.2: Simple greedy heuristic, one strategy (the one with least δ) is pruned in each iteration until Ω is all used up.

10.3.3 Computing Tighter Error Bounds

We can reduce several players' strategy spaces by running Algorithm 10.3 sequentially. Let Γ be the original game, and let Γ' be the reduced game. Let $\{\mathbf{S}_i\}$ and $\{\mathbf{S}'_i\}$ be the set of all players' strategy spaces for Γ and Γ' respectively. For each player i, let Ω_i be the accumulated error actually used in GREEDY-K. The total error generated by these reductions, according to Proposition 10.4, is then $\sum_i \Omega_i$. Given that both $\{\mathbf{S}_i\}$ and $\{\mathbf{S}'_i\}$ are known to us, we are interested in finding a tighter bound on the error.

Let set $\mathbf{M}$ be the set of all NEs in Γ'. Then for each $\sigma \in \mathbf{M}$, it is an ϵ_σ-NE in Γ. This ϵ_σ, by definition, is the maximal gain any player can get by unilaterally deviating to the original strategy space. The overall error bound is the maximum of all NEs' error bounds, i.e.,

$$\epsilon = \max_{\sigma \in \mathbf{M}} \max_{i \in \mathcal{N}} \max_{t \in \mathbf{T}_i} \left\{ u_i(t, \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i}) \right\}, \tag{10.8}$$

where set $\mathbf{T}_i$ is defined as $\mathbf{S}_i \setminus \mathbf{S}'_i$. To compute ϵ with (10.8), we must first find all NEs for Γ'. However, computing all NEs, as mentioned at the beginning of the chapter, is not easy, and in many cases, not possible. Therefore, we would like to find a way to compute ϵ without having to find all NEs a priori. If this is not possible, at least we would like to find a way to compute a bound (as tight as possible) for ϵ.

Since σ is a NE in Γ', $u_i(\sigma_i, \sigma_{-i}) \geq u_i(x_i, \sigma_{-i})$, for all $x_i \in \Delta(\mathbf{S}'_i)$. For each $i \in \mathcal{N}$ and $t \in \mathbf{T}_i$ pair, we associate it to a mixed strategy x_i^t, and we can obtain an upper bound

ALGORITHM 10.3: GREEDY-K($\mathbf{S}$,Ω,k)

$n \leftarrow 1,\ \mathbf{S}^n \leftarrow \mathbf{S}$
while $\Omega > 0$ **do**
  for $s \in \mathbf{S}_i^n$ **do**
    $\delta(s) \leftarrow$ LP-A($\mathbf{S}^n, \{s\}$)
  end for
  $\mathbf{K} \leftarrow \{\}$
  for $j = 1$ to k **do**
    $t_j \leftarrow \arg\min_{s \in \mathbf{S}_i^n \setminus \mathbf{K}} \delta(s)$
    $\mathbf{K} \leftarrow \{\mathbf{K}, t_j\}$
  end for
  $\delta_{\mathbf{K}} \leftarrow$ LP-A($\mathbf{S}^n, \mathbf{K}$)
  if $\Omega \geq \delta_{\mathbf{K}}$ **then**
    $\Omega \leftarrow \Omega - \delta_{\mathbf{K}}$
    $\mathbf{S}_i^{n+1} \leftarrow \mathbf{S}_i^n \setminus \mathbf{K},\ \mathbf{S}^{n+1} \leftarrow (\mathbf{S}_i^{n+1}, \mathbf{S}_{-i})$
  else
    if $\Omega \geq t_1$ **then**
      $\Omega \leftarrow \Omega - \delta(t_1)$
      $\mathbf{S}_i^{n+1} \leftarrow \mathbf{S}_i^n \setminus \{t_1\},\ \mathbf{S}^{n+1} \leftarrow (\mathbf{S}_i^{n+1}, \mathbf{S}_{-i})$
    else
      $\Omega \leftarrow 0$
    end if
  end if
  $n \leftarrow n + 1$
end while
return $\mathbf{S}^n$

Figure 10.3: Generalized greedy heuristic, which is similar to Algorithm 10.2, but prunes k strategies in each iteration.

on (10.8):

$$\begin{aligned} & \max_{i \in \mathcal{N}} \max_{t \in \mathbf{T}_i} \max_{\sigma \in \mathbf{M}} \left\{ u_i(t, \sigma_{-i}) - u_i(x_i^t, \sigma_{-i}) \right\} \\ \geq\ & \max_{\sigma \in \mathbf{M}} \max_{i \in \mathcal{N}} \max_{t \in \mathbf{T}_i} \left\{ u_i(t, \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i}) \right\} = \epsilon \end{aligned} \tag{10.9}$$

Also note that since $\max_{s_{-i} \in \mathbf{S}'_{-i}} [u_i(t, s_{-i}) - u_i(x_i^t, s_{-i})] \geq \max_{\sigma \in \mathbf{M}} [u_i(t, \sigma_{-i}) - u_i(x_i^t, \sigma_{-i})]$, we can further relax the bound on ϵ, and totally remove set $\mathbf{M}$ from consideration:

$$\begin{aligned} \bar{\epsilon} &= \max_{i \in \mathcal{N}} \max_{t \in \mathbf{T}_i} \max_{s_{-i} \in \mathbf{S}'_{-i}} \left\{ u_i(t, s_{-i}) - u_i(x_i^t, s_{-i}) \right\} \\ &\geq \max_{\sigma \in \mathbf{M}} \max_{i \in \mathcal{N}} \max_{t \in \mathbf{T}_i} \left\{ u_i(t, \sigma_{-i}) - u_i(x_i^t, \sigma_{-i}) \right\} \\ &\geq \max_{\sigma \in \mathbf{M}} \max_{i \in \mathcal{N}} \max_{t \in \mathbf{T}_i} \left\{ u_i(t, \sigma_{-i}) - u_i(\sigma_i, \sigma_{-i}) \right\} = \epsilon \end{aligned} \tag{10.10}$$

According to (10.10), we can find $\bar{\epsilon}$ by solving the following optimization problem:

$$\begin{aligned} \min \quad & \bar{\epsilon} \qquad (10.11) \\ \text{s.t.} \quad & \\ & \bar{\epsilon} \geq u_i(t, s_{-i}) - \sum_{s_i \in \mathbf{S}'_i} x_i^t(s_i) \cdot u_i(s_i, s_{-i}), \forall\, i \in \mathcal{N}, t \in \mathbf{T}_i, s_{-i} \in \mathbf{S}'_{-i} \\ & \sum_{s_i \in \mathbf{S}'_i} x_i^t(s_i) = 1, \forall\, i \in \mathcal{N}, t \in \mathbf{T}_i \\ & 0 \leq x_i^t(s_i) \leq 1, \forall\, i \in \mathcal{N}, t \in \mathbf{T}_i, s_i \in \mathbf{S}'_i, \end{aligned}$$

Note that this formulation is very similar to LP-A(**S**, **T**) in Figure 10.1, which is constructed according to Definition 10.4. The major difference is that LP-A($\cdot$) is defined for a particular player i, but (10.11) considers all players at once.

10.3.4 δ-Dominance for Symmetric Games

So far in this chapter, we have assumed that the procedure of δ-dominance is used to prune one strategy (or a set of strategies) from an agent's strategy space. However, for a symmetric game, this assumption forces us to miss the opportunity for pruning strategies from more than one player. In this section, we show that if we are given a symmetric game, and it takes δ_s to prune strategy s from one player's strategy space, then the accumulated error for pruning s from all players' strategy spaces is still δ_s.

Proposition 10.5 *Suppose we are given a symmetric N-player game, Γ, and each player's strategy space, $\mathbf{S}_i$, is by definition identical. Let δ_s be required to prune s from player i's strategy space, and let Γ' be the reduced game with s pruned from all players' strategy spaces. Then an ϵ-NE in Γ' is a $(\delta_s + \epsilon)$-equilibrium in Γ.*

Proof.
From Definition 10.1, we know that $\exists\ \hat{\sigma}_i \in \Delta(\mathbf{S}_i)$, such that:

$$\delta_s + u_i(\hat{\sigma}_i\ , s_{-i}) \geq u_i(s, s_{-i}), \forall s_{-i} \in \mathbf{S}_{-i}.$$

Let σ be an ϵ-NE in Γ'. Then by multiplying each $\sigma_{-1}(s_{-1})$ to the corresponding inequality above, we have:

$$\delta_s + u_i(\hat{\sigma}_i\ , \sigma_{-i}) \geq u_i(s, \sigma_{-i})$$

Since σ is an ϵ-NE in Γ', we know that $\epsilon + u_i(\sigma_i\ , \sigma_{-i}) \geq u_i(\hat{\sigma}_i\ , \sigma_{-i})$. Therefore, we have:

$$(\delta_s + \epsilon) + u_i(\sigma_i\ , \sigma_{-i}) \geq (\delta_s + \epsilon) + u_i(\hat{\sigma}_i\ , \sigma_{-i}) \geq u_i(s, \sigma_{-i})$$

From definition, σ is $(\delta_s + \epsilon)$-equilibrium in Γ. ∎

From Proposition 10.5, we know that for a symmetric game, once we identify δ_s for strategy s, we can eliminate s from all players' strategy space, without incurring additional errors (in other words, the total error we are adding to the equilibrium in the

reduced game, with s removed from all players' strategy spaces, is at most δ_s). According to Proposition 10.5, we can modify Algorithms 10.2 and 10.3 respectively. For Algorithm 10.2, we should modify line 10, so that $\{t\}$ is pruned from all players' strategy spaces within the same iteration. For Algorithm 10.3, we should modify line 14 and line 18 similarly.

Exploiting symmetry is also beneficial in solving the optimization problem (10.11). By exploiting symmetry, we can reduce the size of the problem by considering only one i, instead of all players in $\mathcal{N}$, since the inequalities will be identical for all players. Also, for s_{-i}, we only need to create inequalities for opponent strategy profiles with unique strategy ingredients, as mentioned in Section 9.2.

10.4 Numerical Experiments

Following theoretical results from previous sections, we will now show how iterated δ-dominance can be used as a tool in empirical strategic analysis.

As discussed earlier, the attempt to solve for all NEs quickly gets out of hand even if we only consider games with moderate sizes. By using δ-dominance, we would like to more aggressively reduce players' strategy spaces so that we can solve the reduced game with some sacrifice to the quality of the solution. In this section, we use a reduced two-player game [Wellman *et al.*, 2005b] as an example, and demonstrate how strategy pruning can help in solving real games. We are also interested in seeing the difference between GREEDY and GREEDY-K empirically. In our experiments, we compared GREEDY against GREEDY-K with $k = 2$. Since GREEDY can be viewed as GREEDY-K with $k = 1$, in the following discussion, we use GREEDY-1 and GREEDY-2 to represent these two cases.

10.4.1 A Brief Description on the Game

The game studied by Wellman *et al.* [2005b] is a travel-shopping game [Wellman *et al.*, 2003b] with eight players. Due to the exponential growth on the number of strategy profiles in number of players, Wellman *et al.* [2005b] proposed to approximate the original game through hierarchical reduction methods. From their definition, the two-player reduction from the original game is obtained by creating two 4-player groups, and let strategy selection in each group be homogeneous. To be explicit, we assume that the game is symmetric, and then we let player 1 through 4 play a chosen strategy, and player 5 though 8 play another chosen strategy. This is analogous to letting a leading player in each group make decisions for all members in the group, which can then be thought of as a two-player game.

It should be noted that in order to accurately estimate the expected payoff value for each strategy profile, on average we will need to execute over 20 simulations (per profile). Given that the number of strategies in this game is 40, it is possible to evaluate all possible profiles (total number of profiles for the 2-player reduction game is 840), how-

ever, Wellman *et al.* [2005b] choose to skip some of the less promising profiles in order to make best use of limited computation time.

Due to this reason, when analyzing the game, we will skip any strategy if its inclusion will result in some profiles having undefined payoffs (due to the lack of simulations). For a partially explored payoff matrix, if such principle is followed, we should be able to identify multiple subsets of strategies that are maximal in the sense that the inclusion of any additional strategy will result in some unexplored profiles. In the following analysis, we will only look at the largest such set (with 27 strategies).

10.4.2 Comparison of GREEDY-1 and GREEDY-2

In this section, we will start with the 27-strategy set, and apply GREEDY-1 and GREEDY-2 on it. By testing both heuristics on this real case, we would like to answer following two questions: (1) How much better is GREEDY-2 compared to GREEDY-1 in terms of efficiency in pruning strategies? (2) Given a path of strategy pruning, a tight bound can be found by using formulation in (10.11), how tight is it compared to the accumulated error? One related question is, how tight is the bound obtained by (10.11), when compared to the real equilibrium error?

To answer the first question, we execute both GREEDY-1 and GREEDY-2with $\Omega = 200$, and we track the progresses of both heuristics. The comparison can be seen in Figure 10.4, where the evolutions of number of strategies versus accumulated δ are plotted for both GREEDY-1 and GREEDY-2. As demonstrated by Figure 10.4, we can see that given the same δ consumption, GREEDY-2 eliminates more strategies than GREEDY-1. This results shows that, for a strategy pair (A, B), where δ_A and δ_B (δs required to dominate A and B respectively) are the smallest two among standing δs, it is usually the case that $\delta_{AB} \leq \delta_A + \delta_B$ (δ required to dominate A and B in the same iteration is smaller than the sum of δ_A and δ_B). Of course, since $\delta_{AB} \geq \delta_A$(or δ_B), when trying to identify next two strategies to be eliminated, it is usually wise to choose two strategies with similar δs. It should be noted that although for this specific numeric case, the orders in which the strategies are pruned are almost identical for both GREEDY-1 and GREEDY-2, in general they can be arbitrarily different.

Next thing we are interested in is the error bounds with different tightness. The loosest bound is the accumulated error used by the greedy heuristic. A tighter bound can be computed by using (10.11), suppose we have already identified a set of δ-dominated strategies (either through GREEDY-K, or other heuristics). The tightest bound can be found by looking at the symmetric NEs computed in the reduced game, and for each such NE, evaluating its ϵ if any player is allowed to deviate to the strategy in the original game. Although this bound sounds like an exact bound, it is not, since we are computing ϵ only for the "symmetric NE". The issue with computing ϵ for each symmetric NE is again that we have to solve all the NEs for games with various sizes. In many cases, GAMBIT, the software tool we use, cannot finish it even given several days of computation time. To help GAMBIT solve these games, we can perturb the payoff matrix slightly, and hope this slight perturbation can help us avoiding possible numeric difficulties that stop us from solving the game.

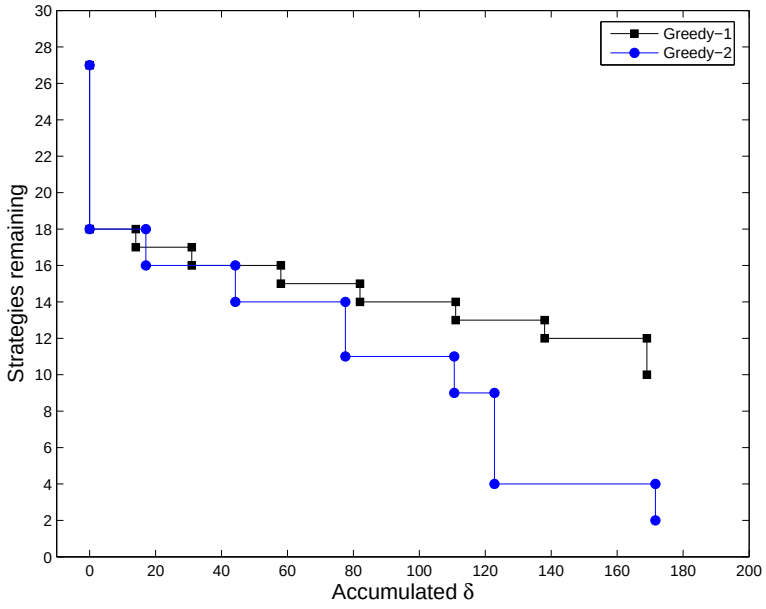

Figure 10.4: Evolutions of number of remaining strategies versus accumulated δ.

The perturbation approach is used since the algorithm GAMBIT used in searching for NEs in a two-player game is Lemke-Howson algorithm [Lemke and Howson, 1964], a pivotal algorithm very similar to the Simplex method. Due to the same reason as in the Simplex method, Lemke-Howson algorithm would suffer from the numerical difficulties if the problem is degenerate. Researchers in the linear programming community have long suggested the use of random perturbation in resolving degeneracy and this has been mentioned in the work by Lemke and Howson [1964]. Of course, perturbing the payoff matrix may introduce some errors to the NEs found, however, if this method can indeed help us solve the game we cannot solve before, it should be worthwhile (since our purpose lies in obtaining an idea on the tightness of various bounds).

In our experiments, we use the following procedure to repeatedly try to solve a game until it can be solved within a predetermined amount of time:

1. Given a game Γ, we randomly perturb its payoff matrix by adding a value randomly drawn from $U[0, P]$[1] to each $u_i(s)$, for all $i \in \mathcal{N}$ and all $s \in \mathcal{S}$ (it should be noted that in this step, we always apply perturbations to the original payoff matrix).

2. Solve the game with Lemke-Howson algorithm, wait for T seconds, if the game is not solved, terminate the solver and go to Step 1; otherwise end the process.

The implementation of this process indeed resolved the numerical difficulties we have had earlier. In our implementation, we let $T = 25$, and the maximal amount of time spent in solving a game is 135.7 minutes (325 instances are generated) for games with 18 strategies. With the 2-player game solved at different sizes, we can now provide a complete summary on the behavior of the GREEDY heuristic. Different bounds for the

[1]U[a,b] is a uniform random variable in $[a, b]$

reduced games generated by GREEDY-1 and GREEDY-2 are summarized in Table 10.1. These relationships are also plotted in Figure 10.5. Note that for the 18-strategy game, since only strictly dominated strategies are eliminated, the NEs found in it shouldn't contain any error ($\epsilon_{\max}$ should be 0), however, since we randomly perturb it in order to solve for the NEs, minor errors are incurred.

$\lvert\mathcal{S}\rvert$	GREEDY-1	GREEDY-2	Tighter bound	$\epsilon_{\max}$
18	0	0	0	0.93
17	14	-	14.67	0
16	31	17.09	17.09	0
15	58	-	27.11	0
14	82	44.2	27.11	1.06
13	111	-	28.43	18.02
12	138	77.59	28.43	20.37
11	-	77.59	28.43	18.67
10	169	-	30.12	0
9	-	110.58	32.99	0
7	-	110.58	12.18	0
6	-	110.58	12.12	0
5	-	110.58	3.74	0
4	-	122.76	3.74	0
2	-	171.6	48.84	20.18

Table 10.1: Summary of various error bounds at each strategy level.

10.5 Conclusion

The explosion of strategy space we encountered in real world can be handled by either reducing the number of agents, or as stated in this chapter, by reducing the number of each agent's strategies. By combining these two types of reduction methods, we are able to treat a fairly large empirical game, with 8 agents, and each agent with 40 strategies. Any attempt to directly solve such game without exploiting symmetry and reasonable reductions is hopeless. After applying various reduction techniques already investigated in the literature, the game is reduced to a 2-player, 27-strategy game. However, to enable the search for all NEs, we must slash some additional strategies systematically. The methodology mentioned in this chapter provides a way to achieve this.

While computing all NEs is empirically infeasible even for a 2-player game with over 14 strategies, we can apply the random perturbation technique frequently mentioned in the literature and approximately solve the game. By comparing the bounds we computed to the real error, we can see that the tighter bound we suggested in Section 10.3.3 indeed provides a much closer bound on NE errors. This implies that once we obtain a list of pruned strategies (which can be determined either by the greedy heuristic suggested here, or any other approach), a much tighter bound can be found.

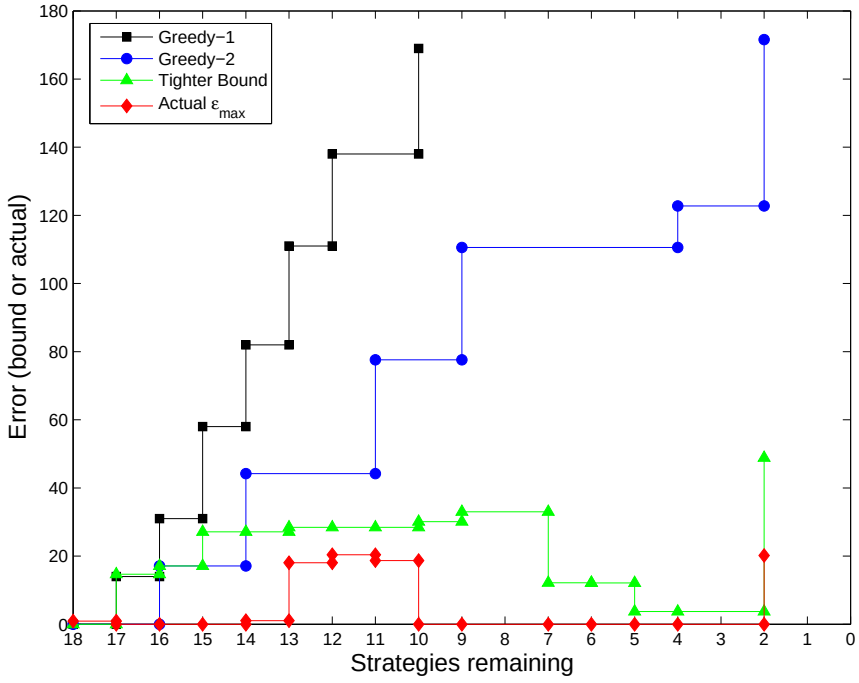

Figure 10.5: Error bounds at each strategy level.

CHAPTER 11

Task Allocation for Dynamic Information Processing Environments: A Motivational Example

Designing market mechanisms for a complex environment is difficult. Firstly, the designer has an infinite design space; secondly, even if the design space can be restricted, it is not clear how to properly evaluate each design, since the value of each design inevitably involves how each agent would react to it.

In Chapter 9, we have described a collection of techniques that can be used in addressing these issues. The ultimate goal of these tools is to scientifically analyze a given scenario and propose a reasonable solution. Depending on who is using these tools, the so-called "solution" may have different meanings. For participating agents, a "solution" may be a suggestion about optimal strategy (in game-theoretic settings, a NE). For the market designer, a "solution" is the market mechanism that optimizes certain performance criterion, e.g., social welfare. In this chapter, we take the market designer's position, and we use a resource allocation problem in generic dynamic information processing environments to demonstrate how important design decisions can be made by using the tools suggested in Chapter 9.

This chapter is organized as follows. Section 11.1 presents a brief introduction and provides motivation. Section 11.2 describes the scenario we are interested in and also its corresponding abstract model. In Section 11.3, we describe agent strategies we designed for the scenario. Section 11.4 presents the setup of the computational experiment and also related analyses. Finally in Section 11.5, we conclude the section with our remarks on the methodology and the scenario.

11.1 Introduction

One important problem resistive of centralized control techniques is managing the allocation of information-processing resources within a dynamic, knowledge-intensive environment. Such resources (e.g., analysts, computational facilities, sensors and other data collection assets) are typically distributed geographically, may be owned by different

organizations (private and public), and may be subject to inter-operability constraints. In practice, this leads to great inefficiencies, and an actual level of information processing well below potential aggregate capacity. Advances in networking and inter-operation standards promise to facilitate flexible allocation, but realizing the potential gains will require a suitable global planning methodology for the task allocation problem.

Our work evaluates the potential of applying market-based approaches to dynamic task allocation problems in information-processing environments. In general, a task allocation problem involves multiple independent decision makers (i.e., agents), where each agent is assigned certain number of tasks and each task may have different resource requirements and value associated with it. The problem is dynamic, meaning that besides initially assigned tasks, agents may be given new tasks dynamically. The problem is also decentralized, since this task-related information is by default known only to each agent and each agent makes its decision independently (based on this information), aiming at optimizing its own objective. As described in Chapter 8, these difficulties (decentralized control and distributed information) are exactly the ones that can be best handled by the market-based approach, and these characteristics motivate the study of market-base approaches in this scenario.

11.2 Task Allocation Scenario

In the remainder of this chapter, we describe a generic task allocation problem, and our investigation of a market game scenario addressing a particular configuration of this generic problem. The model is specified abstractly, with no particular interpretation applied to tasks or resources. Intuitively, the tasks correspond to information-gathering or processing assignments, and the resources to factors (e.g., human labor or expertise, computation cycles, sensor operations, communication activities) that contribute to achieving the tasks. The model generalizes a scenario we developed originally for the information-collection domain [Cheng *et al.*, 2004b], incorporating the extension to include dynamic tasks and task dependency.

In Figure 11.1 we can see a high-level graph illustrating this scenario. On the left-hand side are agents, each endowed with certain number of tasks (which can be assigned at the beginning of the planning horizon, or can be arriving dynamically later) where details on these tasks are assumed to be known only to the agent owning these tasks. On the right-hand side are resources, categorized by resource types (e.g., computing capacity, capital, and human resources) and time spans in which these resources are to be consumed. In a centralized setting, these resources are allocated by a central planner. In a decentralized setting, as in our case, the rights to use these resources should be exchanged through a set of pre-defined market mechanisms. Details on the problem are described in the next section.

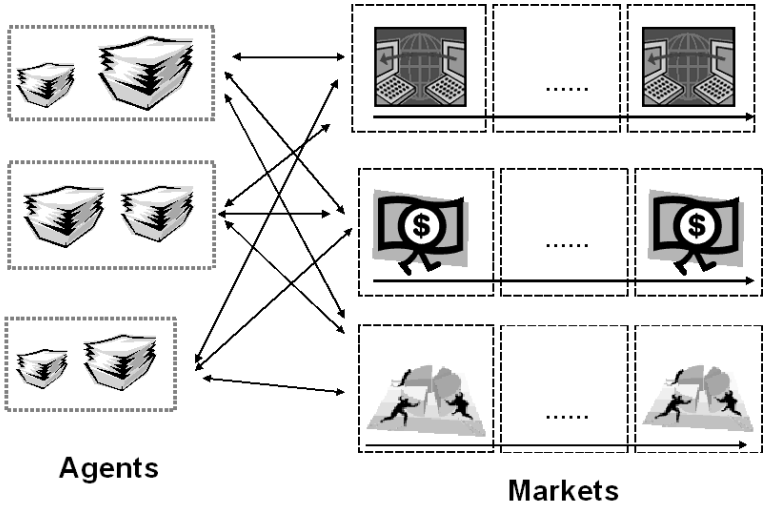

Figure 11.1: A high-level illustration on task allocation problem in a decentralized setting. Agents on the left-hand side are assigned certain tasks independently, and required resources must be obtained through the corresponding exchanges.

11.2.1 Dynamic Task Allocation Problem

In the dynamic task allocation problem, each of N agents may accrue value by performing its assigned *tasks*. Agent i is initially assigned a set of $\mathbf{T}_i$ tasks (we refer to tasks assigned initially as *static tasks*). Agents operate over a *planning horizon* of H discrete time periods, after which the scenario terminates. During each of these H time periods, dynamic tasks may arrive randomly according to a fixed distribution. To finish a task we need to determine a period which is not later than the supplied deadline, and we obtain the rights to use required *resources* in that specific period. Also, the completion of a task may require some task to be finished first. In general a task may depend on multiple tasks, however, in the case we study, we assume each task depends on at most one task.

In Section 9.3 we have introduced a simple resource allocation scenario for the purpose of introducing several frequently used agent design principles. The scenario studied here is more complicated because some tasks arrive dynamically, all tasks are defined with deadlines, and a task may depend on some other task. Before we go into the detail of the problem, we first introduce important parameters that characterize a task in the scenario studied here:

ID A uniquely identifying number.

Priority The value of the task, in monetary units.

Duration The length of the task, in number of time units.

Deadline A time slot index indicating the latest time period when this task should be finished.

Required resources A collection of types and quantities of the resources necessary in finishing the task.

Dependencies The ID of the task this task depends on (it is possible for a task to be independent).

A task may be started in any time period after the completion of the task it depends on (if any), and must be completed at or before the given deadline (if we choose to finish it). Also, the agent must possess the required resources for the duration of task execution.

The major challenges of this problem are the dynamic arrival of tasks and the allocation of resources in a decentralized manner. Since some tasks are assigned dynamically, we must incorporate those tasks in the state space describing the problem, thus quickly exploding the state space. Added to this difficulty is the decentralized way of allocating resources; in most cases, this implies that the exact resources that will be assigned to this agent will depend on other agents' actions. In this chapter, we are interested in designing market mechanisms and agent strategies that are capable of handling these two challenges.

11.2.2 Market Structure

In this section, we would like to propose a design for the market mechanisms to be used in our scenario. We can build the market incrementally by starting with the case where each agent is only assigned static tasks. In this scenario, the most promising candidates are ascending auctions and sealed-bid auctions. As argued by many authors [Cramton, 1998; Ausubel and Milgrom, 2002], in cases where collusion is not a major concern, ascending auctions are more favorable since they are more practical and are able to provide more information to the participants through iterative bidding process. Cheng *et al.* [2004b] demonstrate the use of simultaneous ascending auctions (SAAs) in solving static task allocation problems. In this setting, for the rights of using each type of resource i in each time period j, an ascending auction is established. Usually, an ascending auction closes if there is no bidding activity for a while. The planning horizon will not begin until all auctions close. To simplify the implementation, we assume that each ascending auction will open for a fixed amount of time (which is sufficiently long for agents to finish reasonable number of bidding iterations), after which it will close, and all agents can begin planning their own tasks.

When bidding in each SAA, agents may offer to buy various quantities at various prices. The auction enforces a "beat-the-quote" (BTQ) rule, which dictates that admissible bids must offer to increase or maintain the number of units the agent would be winning at the currently prevailing price, or *price quote* . This BTQ rule is sufficient to ensure that prices only increase, hence the term "ascending auction". At the end of the designated bidding interval, the auction closes, allocates its available units to the top bidders (breaking ties arbitrarily), and charges all winners with the price offered by the lowest winning bid. At closing, all bids (fulfilled or not) are removed from the auctions' order books. If there are unsold units (which implies that the closing price for this auction would be zero), these units are removed from the system.

In the case where agents are assigned both static and dynamic tasks, using only SAAs is no longer sufficient. This is because all resources an agent obtained were based on the requirement of only static tasks (plus some expectation on the arrival of dynamic tasks); when a dynamic task indeed arrives after the planning horizon begins, the holdings of resources in most cases may not meet the requirements of the newly arrived tasks. Therefore, we should provide some market mechanisms for resource exchange after the planning horizon begins, so that agents can exchange resources if incoming dynamic tasks change their plans. Unlike first phase of bidding, in which agents are bidding for resources owned by an auctioneer (through SAAs), the second phase of bidding involves the exchange of resources among agents, thus each agent may be a buyer and a seller at the same time. The most popular market mechanism for this kind of purpose is continuous double auctions (CDAs) [Friedman and Rust, 1993]. This type of auction is both "double" and "continuous" since all participating agents can be both buyers and sellers at the same time, and auctions are cleared continuously as soon as a match is found.

Implementation-wise, this two-phase bidding process can be seen in Figure 11.2. For each (resource-type, time-period) pair, an ascending auction is set up in the first phase (i.e., the preparation phase), SAAs operate until the indicated time line, close, convert to CDAs, and reopen at the beginning of the horizon. A CDA accepts buy or sell offers from any agent, and whenever a buy bid is received that is compatible with an existing sell bid (or vice versa), the offers transact immediately, transferring the corresponding quantity of the goods (right to use the resource in the time period), as well as money balances. Offers that do not match existing bids are retained in the auction's order book, until they subsequently transact with new bids, or are replaced or removed by the original bidder.

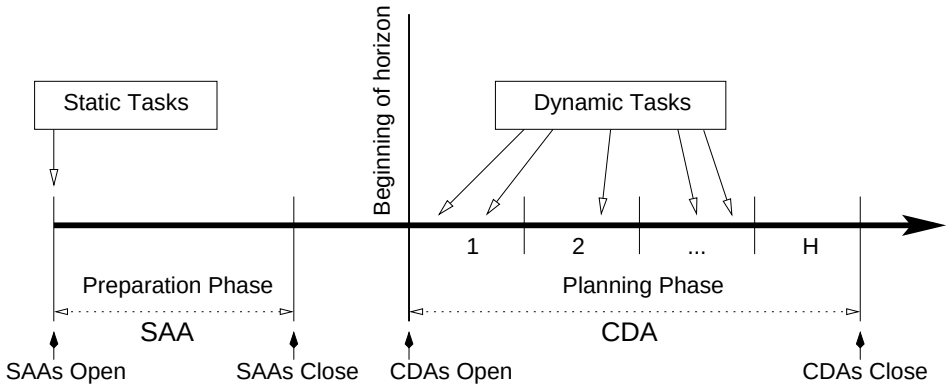

Figure 11.2: Two-phase markets. SAAs are used for the "preparation phase" where each agent drafts its initial plan. After the "planning phase" begins, all SAAs are converted to CDA. The planning is "online", therefore agents will receive dynamic task information, market updates, and have to submit task commitments as time progresses.

Note that the problem is *online*, meaning that the agents must commit decisions sequentially. In particular, they must determine their use of resources in period j before time period j begins. As a result, all auctions related to time period j should close right before time period j. Also, if an agent wants to commit the execution of certain task in time period j, they must make the commitment before time period j. It is agent's

responsibility to ensure that all requirements (including both resources and dependency requirements) for the task are met before submitting the commitment. In our case study, we assume that commitment cannot be retracted once submitted. For any commitment fails to exercise, this agent will be penalized.

As described in Section 9.2, we implement these markets using the AB3D market game system. The specification of this two-phase auction in AB3D auction scripting language is presented in Figure 11.3. The script begins with a series of assignment (`set`) statements, initializing parameters controlling the auction's bidding and clearing policies.[1] Together, these specify a form of ascending auction. The rest of the script comprises a set of rules (employing the `when` construct), specifying the flow of control by defining actions to be taken conditional on parenthesized conditions becoming true. In this case, all conditions are temporal predicates, in one case also contingent on receipt of a valid bid. Times are specified in milliseconds (e.g., 120000 represents two minutes), and built-in variables such as `time`, `gameStartTime`, and `lastQuoteTime` represent time points maintained by the auction state and exposed to the script interpretation engine. Note in particular that two minutes after game start, the third rule is executed, clearing the ascending the auction and modifying the auction parameters in order to set up a CDA policy for the remainder of operation. The CDA closes at the start of the period corresponding to the slot index of its associated resource.

11.3 Agent Strategy

There are several important components in designing agent strategies for this scenario: (1) collecting latest information, (2) deciding resource-bidding strategies for both SAAs and CDAs, and (3) finding task committing policies. No matter how we design the agent strategies, these components must be included. Based on these requirements, we put in place the following skeleton for designing agent strategies:

1. **Setup:** Obtain problem-related information from the game server (e.g., number of planning periods, number of resources, and length of market phases).

2. **Update:** Update transactions and price-quote information from the open auctions. Update dynamic task arrivals from the system.

3. **Compute Commitment Bundle:** As already discussed, the commitment for executing a task in period j must be submitted no later than the beginning of the committed period. However, we don't want to submit the commitment too early, either, because when possible, we always want to make our decisions based on latest information. Therefore, a task commitment plan will only be computed if the time remaining in the current period is below a specified threshold. For the same reason, we will send in commitments incrementally. Only commitments that are scheduled in the next period will be sent in.

[1] Several of these are described in a paper about the AB3D scripting language [Lochner and Wellman, 2004]; further documentation is available at `http://ai.eecs.umich.edu/AB3D/`.

```
defAuction twoPhase {
  set auction_bid_language pq
  set sellerIDs SELLER_ID
  set buyerIDs BUYER_ID
  set bid_btq 1
  set bid_btq_strict 0
  set bid_btq_delta 1
  set matching_fn uniform
  set pricing_k 0
  set bid_dominance_buy none
  set bid_dominance_sell none
  when (time = gameStartTime + 5000)
    {quote}
  when (time = lastQuoteTime + 20000)
    {quote}
  when (time = gameStartTime + 120000)
    {clear;
     set sellerIDs BUYER_ID;
     set matching_fn earliest;
     set bid_btq 0;
     flushBids; quote}
  when (time ≥ gameStartTime + 120000
        and validBid)
    {clear; quote}
  when (time = gameStartTime + 120000 + slotIndex * 120000)
    {close}
}
```

Figure 11.3: AB3D specification of a resource auction. The third and fourth rules (`when` clauses) trigger the change from ascending auction to CDA aftermarket.

4. **Compute Bids:** Compute and submit bids for active auctions.

5. **Repeat:** If end of horizon not yet reached, go to Step 2.

The first two items are simple bookkeeping steps, where the agent apprises itself of the current game state. The substance of the agent's strategy lies in how it determines its task commitment plan, and how it bids in resource auctions based on that plan. We defined two specific strategy variations. The first is a straightforward greedy strategy, which is easy to compute but appeals to numerous and unrealistic simplifying assumptions. The second employs the marginal-value based strategy as described in Section 9.3, which improves decision quality at the expense of more complex computation.

11.3.1 Greedy Strategy

In the simplest strategy we consider, the agent computes the *potential gain* each task brings in, calculated by summing the value of this task and the potential gains of all

its dependent tasks[2]. The agent then sorts tasks according to the potential gains, starting from the greatest. For each task considered, it determines the desirable committing period that minimizes anticipated cost of resources. A committing period is feasible if it is before or at the task's deadline, and after the scheduled commitment of task on which it depends (note that if the task we considered does not depend on any task, the earliest time we can commit it is the first period). The anticipated cost of a particular resource at a given time slot is its current ask price (the part of the price quote indicating the price to buy) at auction, or zero if the agent currently holds the resource and has not committed it to a previously considered task.

To determine the offer price for a particular unit of resource, the agent calculates its value for the task using the resource, subtracting the ask prices of resources that it needs to procure. It then divides this surplus among the resources, offering at the prevailing ask price plus the associated fraction of surplus. Agents place minimal buy offers even for resources that they have no current use for, anticipating the possibility of dynamic task arrivals or resale opportunities.

The bidding policy for buy offers is basically the same during the first (SAA) and second (CDA) phases. In the preparation phase it recalculates bids upon receiving new price quotes, adjusting if necessary to satisfy the BTQ rule. In the planning phase, it reconsiders bids continuously, and does not need to perform any BTQ adjustments.

In the preparation phase, the agent cannot sell any resources. In the planning phase, it offers to sell any resources from its holdings that it does not project to use, based on the task planning process described above. It places sell offers at a predetermined reserve price.

Finally, the agent determines its commitment bundles based on the most recent calculated task performance decisions, just prior to the start of each period. Once a task is committed, the resource holdings will be adjusted accordingly to reflect the usage of resources.

11.3.2 Marginal-Value Bidding Strategy

Marginal-value bidding strategy introduced here is very similar to the one described in Section 9.3, except that we also have to include the task committing periods in the completion problems we solve. The modified completion problem is presented in (11.1), with following notation. **Q** and **R** are the set of periods and the set of resources respectively. T_s and T_d represent the set of static tasks and the set of dynamic tasks. For each task t, $M_{t,r}$ is the amount of resource type r required, D_t is the deadline, v_t is the value, and E_t is the parent task it depends on. Let $E_t = 0$ if task t doesn't have a parent. The auction is identified by the pair (r, q), with $r \in \mathbf{R}$, and $q \in \mathbf{Q}$. For each pair (r, q), let $H_{r,q}$ be agent's holding for goods in (r, q), and $P_{r,q}$ be the current asking price. The decision variables are $b_{r,q}$ and $x_{t,q}$. $b_{r,q}$ represents units of (r, q) bought. $x_{t,q}$ is binary and equals

[2]If we define nodes as tasks, and use the task dependencies to connect these nodes to a tree, the *potential gain* at a node is just the sum of this task's value and all child nodes' potential gain.

1 if task t is scheduled in period p, 0 otherwise.

$$\max \quad \sum_{t \in T_s \bigcup T_d} v_t \cdot \sum_{q \in \mathbf{Q}} x_{t,q} - \sum_{r \in \mathbf{R}, q \in \mathbf{Q}} b_{r,q} \cdot P_{r,q} \tag{11.1}$$

s.t.

$$\sum_{q \in \mathbf{Q}} x_{t,q} \leq 1 \ , \forall \ t \in T_s \bigcup T_d$$

$$\sum_{q \in \mathbf{Q}} x_{t,q} \cdot q \leq D_t \ , \forall \ t \in T_s \bigcup T_d$$

$$\sum_{t \in T_s \bigcup T_d} x_{t,q} \cdot M_{t,r} \leq H_{r,q} + b_{r,q} \ , \forall \ r \in \mathbf{R}, q \in \mathbf{Q}$$

$$\sum_{i=1}^{q} x_{t,i} \leq \sum_{i=0}^{q-1} x_{E_t,i} \ , \forall \ t \in T_s \bigcup T_d \ , q \in \mathbf{Q}$$

In this strategy, agent's bidding behavior is divided into two phases, just as indicated in Figure 11.2. In both preparing phase and planning phase, the agent bids according to the marginal values computed using (11.1) as subproblem. The marginal values are computed similarly as described in Section 9.3: The $(+n)$ marginal value of goods (r, q) = the value with EXACTLY $(H_{r,q} + n)$ goods (r, q) − the value with EXACTLY $(H_{r,q} + n - 1)$ goods (r, q), where $H_{r,q}$ indicates the current holding of goods (r, q). Similarly, one can compute the $(-n)$ marginal value.

The difference between these two phases is that in CDA, since at most one unit of resource is guaranteed at the current asking price, we have no idea about the cost of two or more units of resources. As a result, when solving (11.1) in the planning phase, $b_{r,q}$ is assumed to take only binary value. A consequence of this is that during the planning phase, we only need to compute the $(+1)$ and (-1) marginal values. Another important difference between these two periods is that task commitments, as represented by $x_{t,q}$, are only submitted during the planning phase. As a result, in some later time periods in the planning phase, some tasks may already been committed to some earlier periods; therefore, when solving (11.1) with committed tasks, we have to fix the corresponding $x_{t,q}$ to 1 or 0 depending on whether task t has been committed or not.

If marginal values are used as bidding prices, it's possible that for some goods, the buying price (i.e. $(+1)$ marginal value) might be higher than the selling price (i.e. (-1) marginal value)[3]. Since the type of auction used in the second phase is CDA, such bids with buying price higher than selling price will be transacted immediately. This type of behavior is undesirable from both agent's and market designer's point of view. For the agent, failing to put bids into action implies that it is giving up chance of getting more value. And for the market designer, with a collection of agents failing to participate in

[3]It might not be straightforward why the marginal values for a single goods may be increasing, given that agent's value is obtained through the fulfillments of tasks. The reason for this to happen in our case is the dependency of tasks. To illustrate this, suppose we only have one type of resource, and two tasks, task A and task B. Let task B depend on task A and have higher value. Suppose both task A and task B request one unit of resource, then the marginal value of the second unit will be higher than that of the first unit, because task A (which has lesser value) must be finished first.

the trading every once in a while implies the deterioration of the market efficiency. To counter this issue, we will introduce shading into agent's bidding process.

When trying to come up with the bid for each individual auction, we will check if we have buying price higher than selling price. If this is not the case, the bid will be submitted without modification. However, if the buying price is higher than the selling price, we will shade down the buying price and increase the selling price so that modified selling price is higher than the buying price. Since there are many possible ways of designing shading schemes and there is no compelling argument which method should prevail, we will just use the simplest scheme, described as follows. In Figure 11.4, we assume that we are about to submit two bids, one for buying, $(b_i, +1)$, and one for selling, $(s_i, -1)$, where b_i, s_i stand for the original buy and sell princes, and $b_i^{'}$, $s_i^{'}$ stands for the modified prices. The tuple (p, q) is the bid string where p is price and q is quantity. Positive quantity stands for buying while negative quantity stands for selling. δ and β are the user-specified constants. In our experiments, we use $\delta = 0.5$ and $\beta = 1$.

```
if b_i ≤ s_i
   submit (b_i, +1), (s_i, −1)
else
   s_i' = δ(b_i − s_i)
   b_i' = s_i' − β
   submit (b_i', +1), (s_i', −1)
end if
```

Figure 11.4: Simple shading procedure for the marginal value strategy.

The commitment package will only be computed when it is within 20 seconds to the next period (remember, the length of a period is 120 seconds). This limitation is used here because we believe that the agent will make better commitment decision with more information.

11.4 Numerical Experiments

11.4.1 Setup

Numerical data used in our experiment is defined as followed:

- Time periods: 5 (2 minutes per period)
- Number of agents: 4
- Number of resources: 4
- Capacity of each resource: 4
- Number of static tasks: 6

- Number of dynamic tasks: uniformly distributed between 4 and 8.
- Task attributes:
  - ID: a unique sequential number that identifies the task.
  - Arrival time: for static tasks, this attribute is meaningless; for dynamic tasks, it's distributed uniformly between 2 and 5.
  - Value: for each static task, the value uniformly distributed between 100 and 1000; for each dynamic task, the value uniformly distributed between 100 and 1200.
  - Deadline: uniformly distributed between 2 and 5.
  - Resource requirement: each resource is required with probability 0.5.
  - Task dependency:
    - For static tasks: it will depend on task between 1 and (ID-1) with probability 0.5.
    - For dynamic tasks: a static task will be required with probability 0.5.
  - Duration: fixed to one period for simplicity.

The market environment is provided by *AB3D*, a configurable market environment. The auction services and the communication protocols are also provided by *AB3D*.

11.4.2 Dynamic Task Allocation Scenario in GDL

In Section 9.2, the GDL is introduced as a general language for describing market games. For completeness, we include important components of this scenario, written in GDL, at the end of Appendix B. For detailed lists, please refer to Figure B.1, B.2 and B.3.

11.4.3 Analysis and Discussion

In this section, we will present the result of our computational experiments. The design of the experiments aims at answering following issues:

- How good is the marginal value bidder compared to the simple bidder?
- What is the value of second-phase auction (i.e., CDAs used in the planning phase)?
- What is the benefits of shading for the marginal value bidders? Does shading actually result in more transactions?

To answer the above three questions, we must first introduce a variety of agent strategies based on Section 11.3.1 and 11.3.2. Greedy strategy introduced in Section 11.3.1 is included in the strategy portfolio without additional modification. However, for the

marginal-value strategy described in Section 11.3.2, we will create three variants out of this basic framework. The first variant, *MARG (w/ shading)*, is the most complete version, with all the features as described in Section 11.3.2. The second variant, *MARG (w/o shading)*, is the marginal value strategy without the shading procedure described in Figure 11.4, i.e., all bids generated from the marginal value computation routine are submitted without modification. The third and the final variant, *MARG (w/o CDA)*, is the most crippled strategy version, since it skips bidding in the planning phase altogether. It still updates dynamic task arrivals, and the commitments are also computed dynamically, however, it assumes that bidding in the planning phase is not allowed.

With these agents strategies, we must also define relative scale of the market's performance against centralized planning. However, since the exact global solution is extremely hard to obtain due to the dynamic nature of the problem, we will come up with upper and lower bounds on the performance ratio (between market mechanism and global planning) instead.

- **Percentage versus static global sum:** The static global sum is computed by collecting holdings and task-related information from all agents, and assumes that all dynamic tasks are treated as static tasks, meaning that they are known to the planner a priori. This measure will serve as the lower bound of the percentage versus real expected global sum. This measure is very similar to the computation of the value of perfect information in Section 3.5.3. In this measure, we remove the stochasticity and also the decentralization from the problem. The mathematical formulation of this problem is just (11.1), with $b_{r,q}$ fixed to 0, and T_s and T_d replaced by the set that contains all agents' static and dynamic tasks.

- **Percentage versus rolling-horizon global sum:** The rolling-horizon global sum is computed by assuming that the solver knows the final combined holdings from all agents. Also, all agents' static tasks and "revealed" dynamic tasks up to current period are assumed to be available. The rolling-horizon global solver will compute the commitment plan period by period, with appropriate dynamic task information (all dynamic tasks arriving beyond current period are treated as non-existent to the solver). Note that as in the individual agent's case, the global solver only commits tasks that are due in the next period.

The results of the experiments, in terms of above measures, are summarized in Table 11.1.

Strategy	(%) v.s. static global sum	(%) v.s. rolling-horizon global sum	Number of transactions
Greedy	50.19	64.1	78.2
MARG (w/o shading)	70.35	86.07	72.9
MARG (w/ shading)	77.41	93.09	82.8
MARG (w/o CDA)	71.41	85.99	53.6

Table 11.1: Performance comparison.

With the results in Table 11.1, we can then answer the three questions raised earlier in the section.

- From the result we can see that all versions of marginal-value-based agent strategies outperform simple bidder. We are not claiming that marginal value bidder is the best strategy for our problem. However, given that marginal values can be easily obtained, in most cases it can be used as the first reasonable strategy.

- The value of the after market can be shown by comparing the performance between MARG (w/o CDA) and MARG (w/ shading). In terms of the percentage versus both static and rolling-horizon global sum, MARG (w/ shading) performs better than MARG (w/o CDA) by around 8%. This 8% can be viewed as the benefit one can get by reacting adaptively to the dynamic events.

- From Table 11.1 we can see the the introduction of bid shading causes 13% increase in the number of transactions and around 8% increase in the system utility. This can be explained by the complementarity in the resource requirements and the dependency among tasks. The dependency among tasks results in sometimes increasing marginal values. And the complementarity in the resource requirements implies that if the agent constantly fails to send in bids because of self-transaction, even if it does manages to get some of the goods it bids, it might turn out to be worthless because other required resources cannot be purchased.

11.5 Conclusion

The main objective of this chapter is to explore the use of market mechanisms in a highly decentralized scenario. As already discussed in previous chapters, market-based approaches can be used in resolving difficulties that are originated from the decentralized nature of the problem. As discussed in Wellman *et al.* [2003a], the choice of market mechanisms and agents' strategic behaviors may result in solution inefficiencies. By controlling our computational experiments properly, we can identify various possible reasons that contribute to the efficiencies or the inefficiencies of the market-based approach.

As demonstrated in this chapter, for the dynamic task allocation problem studied, it is important to create an after-market that allows agents to adjust their respective resource holdings according to the latest received dynamic tasks. However, even with after-market created, if an agent does not carefully check its bidding behavior in the specific market mechanism, even a small glitch may cause significant loss in efficiencies.

CHAPTER 12

Market-Based Approach: Conclusions and Future Work

The second part of this thesis is devoted to issues related to the use of market mechanisms in decentralized resource allocation problems. Chapter 9 focuses on various issues related to the use market games in evaluating market mechanisms. Chapter 10 focuses on aggressive strategy pruning technique that is useful in game-theoretic analysis. Finally, Chapter 11 focuses on the analysis of the use of markets in solving decentralized resource allocation problems.

12.1 Summary of Contributions

In Chapter 9, we gave an overview on a collection of tools one can use in analyzing the performance of market mechanisms in market games. The first important tool discussed is the Game Definition Language, which is part of the AB3D market gaming platform. This language allows us to use AB3D as a standard platform for defining and executing market game simulations, thus eliminating one of the most time consuming parts of setting up numerical experiments. Next in Chapter 10, we introduced an aggressive strategy pruning technique. Although weaker than the usual strategy dominance concept, it is shown to significantly reduce the size of the game without introducing significant errors to the solution. With the help of this strategy pruning scheme, we can quickly obtain a reduced game, solve it, and obtain a tight error bound on the solution. The development of tools of this kind (also see Wellman *et al.* [2005a]) can be used in helping researchers analyzing large games empirically (for example, see Wellman *et al.* [2006] and Kiekintveld *et al.* [2006]).

Finally in Chapter 11, we studied a challenging dynamic task allocation problem. By modeling this problem as a market game, we can answer many qualitative and quantitative questions empirically by executing numerical experiments.

12.2 Future Work

The proposed future work follows the two trends studied in the second part of this thesis. On the study of game-reduction techniques, we are interested in carrying out a more extensive study on the effectiveness of the strategy-pruning technique on a wide variety of games, by using the game generator, GAMUT [Nudelman *et al.*, 2004]. We believe this is the first step towards developing a class of more specialized game-reduction techniques. Ultimately, we are interested in exploiting game structures other than symmetries.

For the market-based approach, we are interested in continuing the study of the dynamic task allocation problem. Our study of the particular scenario in Chapter 11 only answers some qualitative and quantitative questions. To make the scenario more realistic, we can introduce a wider variety of strategies, and perform a more extensive search in terms of classes of mechanisms (one such example is studied in Vorobeychik *et al.* [2006]). Ultimately, by building better game-estimators and game-solvers, which can all be executed automatically, we are interested in building a set of tools that can greatly reduce the labor required in testing and discovering market-based solutions, and discovering insights in market and strategy designs.

APPENDIX A

Adaptive Signal Re-timing

Adaptive signal re-timing in INTEGRATION-UM is an online cycle time and phase-split optimization heuristic, as described in Wunderlich [1994]. The underlying theory for this approach is based on Webster and Cobbe's model [Webster and Cobbe, 1958]. Underlying analysis will not be explained in detail here; instead, the implementation of the algorithm as embedded in INTEGRATION-UM is presented.

The automatic signal re-timing algorithm determines signal timing plans based on current flows on the approaches[1] leading to the signalized intersections. (In this appendix we use the term "flow" to represent the volume of traffic on a link or approach.) The re-timing algorithm in INTEGRATION-UM is invoked repeatedly at user-specified intervals, and proceeds in three steps:

1. **Estimating link flows**: for each signalized intersection, the equivalent flow for each link is estimated by combing average incoming flow and average size of the standing queue. The following formula is used for this purpose:

$$v^a = f^a + 4q^a, \tag{A.1}$$

where v^a is the estimated flow on link a, f^a is the exponentially smoothed average flow on link a, and q^a is the exponentially smoothed average size of the standing queue on link a.

Both average incoming flow (f^a) and average size of the standing queue (q^a) of link a are obtained by periodically performing the following exponential smoothing updates:

$$f^a := 0.75f^a + 0.25f^a_{\text{in}} \tag{A.2}$$

$$q^a := 0.9q^a + 0.1\hat{q}^a, \tag{A.3}$$

where f^a_{in} is the number of vehicles flowing into link a during the interval between smoothing updates, and $\hat{q}^a$ is the size of standing queue on link a during the same interval.

[1] If a signal timing plan is used at more than one intersection within the traffic network, the approach is defined as the set of links coming into these controlled intersections during the same phase.

2. **Computing critical values**: based on the above flow data, the procedure will compute a measure (i.e., critical value) that represents the relative congestion of each link. By using this measure, the procedure then computes cycle length and the allocation of green times.

   For each link a leading to the intersections controlled by the signal timing plan, a critical value (measure of congestion) y^a is computed as the ratio between estimated link flow and link's saturation flow:

$$y^a = \frac{v^a}{s^a}, \tag{A.4}$$

   where s^a is link a's saturation flow rate (as defined in the network topology definition).

   Let the set A_p consist of all the links that have the right of way during phase p of the signal under consideration. The critical value for phase p is then the maximal y^a of all links in A_p:

$$y_p = \max\left\{\max_{a \in A_p}\{y^a\}, y_{\text{min}}\right\}, \tag{A.5}$$

   where y_{min} is a predefined minimal critical value.

   The combined critical value for the signal timing plan, denoted by Y, is then the sum of values of y_p over all its phases:

$$Y = \sum_p y_p. \tag{A.6}$$

3. **Computing cycle time and green time for each phase**: the new cycle time for each signal timing plan, C_o, is computed from its corresponding critical value, Y, and the sum of lost time (i.e., yellow time) for all phases, L. For $Y \leq 0.95$,

$$C_o = \max\{\min\{\frac{(1.5L+5)}{(1-Y)}, C_{\max}\}, C_{\min}\} \tag{A.7}$$

   Otherwise, $C_o = C_{\text{max}}$. $C_{\min}$ and C_{max} are the specified minimal and maximal cycle times, respectively.

   After C_o is obtained, the length of green time for all phases can be computed accordingly. g_p, the length of green time assigned to phase p, is determined by

$$g_p = \frac{y_p}{Y}(C_o - L). \tag{A.8}$$

APPENDIX B

Game Definition Language for Market Games

As discussed in Section 9.2, one of the important functionalities of the market gaming platform is to generate common and agent-specific information. According to the previous discussions, these information may be hierarchical and probabilistic. Therefore, in order to effectively generate these information, it must be easy to specify hierarchical structures and random variables.

Gam Definition Language (GDL) is mainly designed to meet these two requirements. On top of these two requirements, GDL is also designed to make information generation more efficient. Also, GDL must be sophisticated enough to generate some complicated features (e.g., the generation of random sequence and execution of simple arithmetics). To meet all these design goals, and without complicating game design too much, we choose to build GDL based on XML.

One of the major benefits of XML is its inherited ability in representing information hierarchies. Based on XML, we can then realize most of the above mentioned design goals by embedding commands in the XML tags, as illustrated as follows:

1. **Tree structure generation.** This type of task will generate results in a hierarchical manner. The parser will export any tag it encounters, starting for the root node, unless it belongs to specify to execute one of the following command. Usable commands include “for”, “var”, “seeds”, “distribution”, “declare”, “endowment”, which will be explained later.

2. **Pattern-based value generation.** This type of task will generate the text content for a single element. A Java-style pattern tag is defined, and user can use this tag to perform simple arithmetic operations (+ - * /), string composition, and random sequence generation. This task can be nested, i.e., we can have pattern-generated value as the argument as a parent pattern.

B.1 Basic Parsing Rules

When a XML element is parsed, its content will be evaluated according to the following rules:

- If the content is a number, it will be exported directly.
- If the above condition fails, the parser will try to search the global variable hash by using the content as key.
- If the above conditions fail, the parse will try to search the local variable hash by using the content as key.
- If the above conditions fail, the parse will try to evaluate the content as an arithmetic expression and return the value.
- A value -9999 will be exported if all of the above conditions fail.

For convenience, some commonly used values are inserted into a global hash by the parser and can be accessed by the following key:

GAME_ID Game ID.

GAME_PATH The path to game-specific data.

AUCTION_PATH The path to auction-related data.

GAME_START_TIME Game starting time, in milliseconds.

GAME_END_TIME Game ending time, in milliseconds.

SEED The global random seed, it's initialized to use the game ID.

INDEX The index (absolute order) of auction or agent (depending on which file is parsed). Only available in the file that specifies agent's preference and the file that specifies auctions used in the game.

AGENTID Current agent's ID. Only available in the file that specifies agent's preference.

Besides these common keys, all parameters specified in the main game definition files will also be hashed by their tag names and can be used.

B.2 Command Syntax and Examples

There are two ways of using GDL's programming constructs. If we want to export the tag in the final output, we can insert a parameter called *template* inside the tag, and use one of the following commands as the value. Or, if we just want to execute the command without outputting the tag, we can use a pair of special tags called `<CMD>...</CMD>`, and include the command in the parameter "template" as well. The commands that are supported by GDL are explained in detail as follows:

for Used to repeatedly export all the children nodes until the condition on loop variable is false. “for” can be used in an ordinary tag, where the tag, as well as all the children tags will be exported. Or it can be used in special tag `<CMD></CMD>`, where the command will be executed, but the tag will not be exported (as mentioned earlier). Required parameters:

- var: The name of the loop variable.
- from: The starting value of the variable.
- to: The ending value of the variable.

Here is an example for an ordinary tag:

```
<someTAG from="1" to="2" var="X" template="for">
  <ChildrenTags1>
    <ChildrenTags1.1>...</ChildrenTags1.1>
    <ChildrenTags1.2>...</ChildrenTags1.2>
  </ChildrenTags1>
</someTAG>
```

The parsed result will be:

```
<someTAG>
  <ChildrenTags1>
    <ChildrenTags1.1>...</ChildrenTags1.1>
    <ChildrenTags1.2>...</ChildrenTags1.2>
  </ChildrenTags1>
  <ChildrenTags1>
    <ChildrenTags1.1>...</ChildrenTags1.1>
    <ChildrenTags1.2>...</ChildrenTags1.2>
  </ChildrenTags1>
</someTAG>
```

var Used to set the tag’s content as the value of the variable specified by parameter `var`. For example, we can set a tag’s content to the value of the variable in the “for” loop. Required parameters:

- var: The name of the variable whose value will be exported as content.

Example:

```
<someTAG from="1" to="2" var="X" template="for">
  <ChildrenTags1 var="X" template="var" />
  <ChildrenTags2>
    <CMD from="1" to="2" var="Y" template="for">
      <ChildrenTags2.1 var="Y" template="var" />
    </CMD>
  </ChildrenTags2>
</someTAG>
```

The parsed result will be:

```
<someTAG>
  <ChildrenTags1>1</ChildrenTags1>
  <ChildrenTags2>
    <ChildrenTags2.1>1</ChildrenTags2.1>
    <ChildrenTags2.1>2</ChildrenTags2.1>
  </ChildrenTags2>
  <ChildrenTags1>2</ChildrenTags1>
  <ChildrenTags2>
    <ChildrenTags2.1>1</ChildrenTags2.1>
    <ChildrenTags2.1>2</ChildrenTags2.1>
  </ChildrenTags2>
</someTAG>
```

seeds Used to specify the random seed that will be used in all the subsequent parsings. Required parameters:

- type: The type of the seed.

Two types of seeds are available.

1. Use GAME_ID only.
2. Use GAME_ID and INDEX.

Example:

```
<CMD type="2" template="seeds" />
```

distribution Used to generate a value according to the distribution specified. Required parameters:

- distribution: The name of the distribution.

Currently there is only one distribution implemented now (more distribution can be included as needed).

- **UNIFORM** It takes two parameters, lower bound and higher bound. Note that this distribution is actually the discrete uniform distribution.

Example:

```
<deadline distribution="UNIFORM" template="distribution">
  <params>
    <param index="1">1</param>
    <param index="2">10</param>
  </params>
</deadline>
```

Parsed result:

```
<deadline>x</deadline>
```

Where *x* is some discrete-uniform random number drawn between 1 and 10.

declare Used to declare a new entry in the local variable hash. We have to specify two children tags under "declare": `<NAME></NAME>` and `<VALUE></VALUE>`. Text enclosed by `<NAME></NAME>` is the name of the next entry in local variable hash. Text (or number) enclosed by `<VALUE></VALUE>` is the value of this variable. Note that the text (or number) enclose by `NAME` and `VALUE` can also be generated by GDL commands. Also note that if a hash entry with the same name already exists, its value will be overwritten.

Example:

```
<CMD template="declare">
  <NAME>newVariableEntry</NAME>
  <VALUE distribution="UNIFORM" template="distribution">
    <params>
      <param index="1">10</param>
      <param index="2">20</param>
    </params>
  </VALUE>
</CMD>
```

Above example will insert `newVariableEntry` into local variable hash, with value randomly drawn from a discrete uniform distribution between 10 and 20.

endowment Originally it is used to generate agent-specific endowment information. But now it is used to generate any information that is game-specific and cannot be generated by using above mechanism. This command will invoke an user-supplied Java class (with predefined interface) and anything that is outputted will be included under the current tag. Required parameters:

- type: The name of the user-defined Java class.

Example:

```
<someTAG type="some.user.class" template="endowment" />
```

Parsed result:

```
<someTAG>
  whatever outputted by the object
</someTAG>
```

pattern If we include a tag named `<pattern></pattern>` anywhere, it will be handled in a special way. The idea for using such tag is to introduce a way such that we can compose a composite expression or string by inserting various parameters into a pattern. Required parameters:

- format: The Java-style formatting string.
- type: Can be *string* or *value*, explained as follows.

There are two modes for "pattern":

1. **string** The composite string, after parameters inserted will be as is, without further post processing.
   Example:

```
<DoSomePattern>
  <pattern format="SOMETHING-\{0\}-\{1\}" type="string">
    <arg index="0">X</arg>
    <arg index="1">Y</arg>
  </pattern>
</DoSomePattern>
```

   Parsed result, suppose X=1, Y=2:

```
<DoSomePattern>SOMETHING-1-2</DoSomePattern>
```

2. **value** The composite string should be an arithmetic expression, its result will be calculated and outputted.
   Example:

```
<DoSomePattern>
  <pattern format="\{0\}*6+\{1\}" type="value">
    <arg index="0">X</arg>
    <arg index="1">Y</arg>
  </pattern>
</DoSomePattern>
```

   Parsed result, suppose X=1, Y=2:

```
<DoSomePattern>8</DoSomePattern>
```

B.3 The Partial GDL Listings for the Dynamic Task Allocation Problem

In the following three figures, we demonstrate how to use GDL in representing a problem containing random variables. In particular, Figure B.3 presents how to generate random number of tasks, and assign random deadlines to tasks generated.

```
<game>
  <gameLen>840000</gameLen>
  <totalAgents>5</totalAgents>
  <phaseOneEndTime>120000</phaseOneEndTime>
  <numResources>4</numResources>
  <numSlots>5</numSlots>
  <msPerSlot>120000</msPerSlot>
  <whenDynamicTaskComeIn>2</whenDynamicTaskComeIn>
  <numStaticTasks>6</numStaticTasks>
  <minDynamicTaskNumber>4</minDynamicTaskNumber>
  <maxDynamicTaskNumber>8</maxDynamicTaskNumber>
  <minStaticTaskValue>100</minStaticTaskValue>
  <maxStaticTaskValue>1000</maxStaticTaskValue>
  <minDynamicTaskValue>100</minDynamicTaskValue>
  <maxDynamicTaskValue>1200</maxDynamicTaskValue>
  <resourcesCap>4</resourcesCap>
</game>
```

Figure B.1: This is the main game file that defines important game parameters mentioned in Section 11.4.1.

```
<getGameParams>
  <agent var="AGENTID" template="var"/>
  <taskPreferences>
  <CMD type="2" template="seeds"/>
  <list from="1" to="numStaticTasks" var="X" template="for">
    <taskPrefTuple>
      <task var="X" template="var"/>
      <value distribution="UNIFORM" template="distribution">
        <params>
          <param index="1">minStaticTaskValue</param>
          <param index="2">maxStaticTaskValue</param>
        </params>
      </value>
      <deadline distribution="UNIFORM" template="distribution">
        <params>
          <param index="1">2</param><param index="2">numSlots</param>
        </params>
      </deadline>
      <requiredResources>
        <CMD from="1" to="numResources" var="Y" template="for">
          <resourceTuple>
            <type var="Y" template="var"/>
            <quantity distribution="UNIFORM" template="distribution">
              <params>
                <param index="1">0</param><param index="2">1</param>
              </params>
            </quantity>
          </resourceTuple>
        </CMD>
      </requiredResources>
      <requiredTasks>
        <taskTuple>
          <task distribution="UNIFORM" template="distribution">
            <params>
              <param index="1">1</param>
              <param index="2">
                <pattern format="0-1" type="value"><arg index="0">X</arg></pattern>
              </param>
            </params>
          </task>
          <required distribution="UNIFORM" template="distribution">
            <params>
              <param index="1">0</param><param index="2">1</param>
            </params>
          </required>
        </taskTuple>
      </requiredTasks>
    </taskPrefTuple>
  </list>
  </taskPreferences>
</getGameParams>
```

Figure B.2: This figure lists the GDL used in defining agent's preference.

```
<getEvents>
  <agent var="AGENTID" template="var"/>
  <CMD template="declare">
    <NAME>numDynamicTasks</NAME>
    <VALUE distribution="UNIFORM" template="distribution">
      <params>
        <param index="1">minDynamicTaskNumber</param>
        <param index="2">maxDynamicTaskNumber</param>
      </params>
    </VALUE>
  </CMD>
  <taskPreferences>
    <CMD type="2" template="seeds"/>
    <list from="1" to="numDynamicTasks" var="X" template="for">
      <taskPrefTuple>
        <task>
          <pattern format="{0}+{1}" type="value">
            <arg index="0">numStaticTasks</arg><arg index="1">X</arg>
          </pattern>
        </task>
        <CMD template="declare">
          <NAME>taskArrivalSlot</NAME>
          <VALUE distribution="UNIFORM" template="distribution">
            <params>
              <param index="1">whenDynamicTaskComeIn</param>
              <param index="2"><pattern format="{0}-{1}" type="value">
                <arg index="0">numSlots</arg><arg index="1">1</arg>
              </pattern></param>
            </params>
          </VALUE>
        </CMD>
        <time>
          <pattern format="{0}+{1}+{2}*{3}" type="value">
            <arg index="0">GAME_START_TIME</arg>
            <arg index="1">phaseOneEndTime</arg>
            <arg index="2">taskArrivalSlot</arg>
            <arg index="3">msPerSlot</arg>
          </pattern>
        </time>
        <value distribution="UNIFORM" template="distribution">
          <params>
            <param index="1">minDynamicTaskValue</param>
            <param index="2">maxDynamicTaskValue</param>
          </params>
        </value>
        <deadline distribution="UNIFORM" template="distribution">
          <params>
            <param index="1">taskArrivalSlot</param>
            <param index="2">numSlots</param>
          </params>
        </deadline>
        ...
      </taskPrefTuple>
    </list>
  </taskPreferences>
</getEvents>
```

Figure B.3: This figure lists the GDL used in defining dynamically arriving tasks. Note that the section that defines task's parameter is identical to the fragment in Figure B.2, therefore it is neglected here.

BIBLIOGRAPHY

Richard E. Allsop. SIGSET: A computer program for calculating traffic capacity of signal-controlled road junctions. *Traffic Engineering & Control*, 13:58–60, 1971.

Richard E. Allsop. SIGCAP: A computer program for assessing the traffic capacity of signal-controlled road junctions. *Traffic Engineering & Control*, 17:338–341, 1976.

Steve Alpern and Dennis J. Snower. A search model of optimal pricing and production. Discussion paper 224, Center for Economic Policy Research. London, United Kingdom, 1988.

Kalidas Ashok and Moshe E. Ben-Akiva. Alternative approaches for real-time estimation and prediction of time-dependent origin-destination flows. *Transportation Science*, 34(1):21–36, 2000.

Kalidas Ashok and Moshe E. Ben-Akiva. Estimation and prediction of time-dependent origin-destination flows a stockhastic mapping to path flows and link flows. *Transportation Science*, 36(2):184–198, 2002.

Lawrence M. Ausubel and Paul R. Milgrom. Ascending auctions with package bidding. *Frontiers of Theoretical Economics*, 1(1), 2002.

F. Boillot, J.M. Blosseville, J.B. Lesort, V. Motyka, M. Papageorgiou, and S. Sellam. Optimal signal control of urban traffic networks. In *6th International Conference on Road Traffic Monitoring and Control*, pages 75–79, London, England, 1992. IEE.

George W. Brown. Iterative solution of games by fictitious play. In *Activity Analysis of Production and Allocation*, pages 374–376. John Wiley, New York, 1951.

Apostolos N. Burnetas and Craig E. Smith. Adaptive ordering and pricing for perishable products. *Operations Research*, 48(3):436–443, 2000.

Xin Chen and David Simchi-Levi. Coordinating inventory control and pricing strategies with random demand and fixed ordering cost: The finite horizon case. *Operations Research*, 52(6):887–896, 2004.

Xin Chen and David Simchi-Levi. Coordinating inventory control and pricing strategies with random demand and fixed ordering cost: The infinite horizon case. *Mathematics of Operations Research*, 29(3):698–723, 2004.

John Q. Cheng and Michael P. Wellman. The WALRAS algorithm: A convergent distributed implementation of general-equilibrium outcomes. *Computational Economics*, 12:1–24, 1998.

Shih-Fen Cheng and Michael P. Wellman. Iterated weaker-than-weak dominance. In *Twentieth International Joint Conference on Artificial Intelligence*, to appear, Hyderabad, India, 2007.

Shih-Fen Cheng, Daniel M. Reeves, Yevgeniy Vorobeychik, and Michael P. Wellman. Notes on equilibria in symmetric games. In *AAMAS-04, Workshop on Game Theoretic and Decision Theoretic Agents*, New York City, NY, August 2004.

Shih-Fen Cheng, Michael P. Wellman, and Dennis G. Perry. Market-based resource allocation for information-collection scenarios. In Koichi Kurumatani, Shu-Heng Chen, and Azuma Ohuchi, editors, *Multiagent for Mass User Support (MAMUS-03)*, volume 3012 of *Lecture Notes in Computer Science*, pages 33–47. Springer, Jan 2004.

Shih-Fen Cheng, Archis Ghate, Stephen Baumert, Daniel Reaume, Dushyant Sharma, and Robert L. Smith. A fast algorithm for joint optimization of capital investment, revenue management and production scheduling in manufacturing systems. IOE Technical Report 05-05, 2005.

Shih-Fen Cheng, Evan Leung, Kevin M. Lochner, Kevin O'Malley, Daniel M. Reeves, Julian L. Schvartzman, and Michael P. Wellman. Walverine: A Walrasian trading agent. *Decision Support Systems*, 39:169–184, 2005.

Shih-Fen Cheng, Blake Nicholson, Marina A. Epelman, Daniel Reaume, and Robert L. Smith. A dynamic programming approach to the end-state problem. Working paper, 2006.

Shih-Fen Cheng, Marina A. Epelman, and Robert L. Smith. CoSIGN: A parallel algorithm for coordinated traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, to appear, 2007.

Kwok-Yuen Cheung, Chi-Wai Hui, Haruo Sakamoto, Kentaro Hirata, and Lionel O'Young. Short-term site-wide maintenance scheduling. *Computers and Chemical Engineering*, 28:91–102, 2004.

Scott Clearwater, editor. *Market-Based Control: A Paradigm for Distributed Resource Allocation*. World Scientific, 1995.

Vincent Conitzer and Tuomas Sandholm. Complexity results about Nash equilibria. In *Eighteenth International Joint Conference on Artificial Intelligence*, pages 765–771, 2003.

Peter Cramton. Ascending auctions. *European Economic Review*, 42(3-5):745–756, 1998.

Paolo Dell'Olmo and Pitu B. Mirchandani. REALBAND: An approach for real-time coordination of traffic flows on a network. *Transportation Research Record*, 1494:106–116, 1995.

Paolo Dell'Olmo and Pitu B. Mirchandani. A model for real-time traffic coordination using simulation based optimization. In L. Bianco and P. Toth, editors, *Advanced Methods in Transportation Analysis*, pages 525–546. Springer, 1996.

Johann Dréo, Alain Pétrowski, Patrick Siarry, and Eric Taillard. *Metaheuristics for Hard Optimization: Methods and Case Studies*. Springer, 2006.

Alex Fabrikant, Christos Papadimitriou, and Kunal Talwar. The complexity of pure Nash equilibria. In *Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC-04)*, pages 604–612, 2004.

Robin Farquharson. *Theory of Voting*. Yale University Press, New Haven, 1969.

Awi Federgruen and Aliza Heching. Combined pricing and inventory control under uncertainty. *Operations Research*, 47(3):454–475, 1999.

Youyi Feng and Youhua Chen. Joint pricing and inventory control with setup costs and demand uncertainty. Working paper, 2003.

Youyi Feng and Youhua Frank Chen. Optimality and optimization of a joint pricing and inventory-control policy for a periodic-review system. Working paper, 2004.

Daniel Friedman and John Rust, editors. *The Double Auction Market*. Addison-Wesley, 1993.

Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, 1991.

Alfredo Garcia, Daniel Reaume, and Robert L. Smith. Fictitious play for finding system optimal routings in dynamic traffic networks. *Transportation Research B*, 34(2):146–157, February 2000.

Nathan H. Gartner, Farhad J. Pooran, and Christina M. Andrews. Implementation of the opac adaptive control strategy in a traffic signal network. In *Proceedings of the IEEE Intelligent Transportation Systems Conference*, pages 195–200. 2001.

Nathan H. Gartner. OPAC: A demand-responsive strategy for traffic signal control. *Transportation Research Record*, 906:75–81, 1983.

Archis Ghate, Marina Epelman, and Robert L. Smith. Sampled fictitious play for black-box stochastic sequential decision problems. Technical Report 06-02, University of Michigan, 2006.

Itzhak Gilboa, Ehud Kalai, and Eitan Zemel. On the order of eliminating dominated strategies. *Operations Research Letters*, 9(2):85–89, 1990.

Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

Amy Greenwald and Justin Boyan. Bidding algorithms for simultaneous auctions. In *Third ACM Conference on Electronic Commerce*, pages 115–124, New York, NY, USA, 2001. ACM Press.

Amy Greenwald and Justin Boyan. Bidding under uncertainty: theory and experiments. In *Twentieth Conference on Uncertainty in Artificial Intelligence*, pages 209–216. AUAI Press, 2004.

Michael R. Hagerty, James M. Carman, and Gary J. Russell. Estimating elasticities with PIMS data: Methodological issues and substantive implications. *Journal of Marketing Research*, 25(1):1–9, 1988.

Jean-Jacques Henry and Jean-Loup Farges. PRODYN. In *6th IFAC/IFIP/IFORS Symposium on Control, Computers and Communications in Transportation*, pages 253–255, 1989.

Jean-Jacques Henry, Jean-Loup Farges, and J. Tuffal. The PRODYN real time traffic algorithm. In *4th IFAC/IFIP/IFORS Conference on Control in Transportation Systems*, pages 305–310, 1983.

Tsin Hing Heung, Tin Kin Ho, and Yu Fai Fung. Coordinated road-junction traffic control by dynamic programming. *IEEE Transactions on Intelligent Transportation Systems*, 6(3):341–350, September 2005.

P. B. Hunt, D. I. Robertson, R. D. Bretherton, and R. I. Winton. SCOOT - a traffic responsive method for coordinating signals. In *Laboratory Report no. LP 1014*. Transportation and Road Research, Crowthorne, Berkshire, England, 1981.

Michael Kearns, Michael L. Littman, and Satinder Singh. Graphical models for game theory. In *Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 253–260, 2001.

Christopher Kiekintveld, Michael P. Wellman, , and Satinder Singh. Empirical game-theoretic analysis of chaturanga. In *AAMAS-06 Workshop on Game-Theoretic and Decision-Theoretic Agents*. 2006.

Theodore J. Lambert and Hua Wang. Fictitious play approach to a mobile unit situation awareness problem. Technical report, University of Michigan, 2003.

Theodore J. Lambert, Marina A. Epelman, and Robert L. Smith. A fictitious play approach to large-scale optimization. *Operations Research*, 53(3):477–489, May-June 2005.

C. E. Lemke and J. T. Howson. Equilibrium points of bimatrix games. *Journal of the Society for Industrial and Applied Mathematics*, 12(2):413–423, 1964.

Kevin Leyton-Brown and Moshe Tennenholtz. Local-effect games. In *Eighteenth International Joint Conference on Artificial Intelligence*, pages 772–780, 2003.

Wei-Hua Lin and Chenghong Wang. An enhanced 0-1 mixed-integer LP formulation for traffic signal control. *IEEE Transactions on Intelligent Transportation Systems*, 5(4):238–245, December 2004.

John D. C. Little, M. D. Kelson, and Nathan H. Gartner. MAXBAND: A program for setting signals on arteries and triangular networks. *Transportation Research Record*, 795, 1981.

John D. C. Little. The synchronization of traffic signals by mixed-integer linear programming. *Operations Research*, 14(4):568–594, 1966.

Kevin M. Lochner and Michael P. Wellman. Rule-based specification of auction mechanisms. In *Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 818–825, 2004.

R Duncan Luce and Howard Raiffa. *Games and Decisions*. Wiley, New York, 1957.

Jeffrey K. MacKie-Mason and Michael Wellman. Automated markets and trading agents. In Leigh Tesfatsion and Kenneth L. Judd, editors, *Handbook of Computational Economics Vol. 2: Agent-Based Computational Economics*, Handbooks in Economics Series. North-Holland, 2006.

Jeffrey K. MacKie-Mason, Anna Osepayshvili, Daniel M. Reeves, and Michael P. Wellman. Price prediction strategies for market-based scheduling. In *Fourteenth International Conference on Automated Planning and Scheduling*, pages 244–252, 2004.

V. Mauro and D. DiTaranto. UTOPIA. In *6th IFAC/IFIP/IFORS Symposium on Control, Computers and Communications in Transportation*, pages 245–252, 1989.

Richard D. McKelvey and Andrew McLennan. Computation of equilibria in finite games. In *Handbook of Computational Economics*, volume 1. Elsevier, 1996.

A. J. Miller. A computer control system for traffic networks. In *Second International Symposium on the Theory of Road Traffic Flow*, pages 200–220, 1965.

Pitu B. Mirchandani and Larry Head. A real-time traffic signal control system: Architecture, algorithms, and analysis. *Transportation Research C*, 9(6):415–432, 2001.

Pitu B. Mirchandani and Fei-Yue Wang. RHODES to intelligent transportation systems. *IEEE Intelligent Systems*, 20(1):10–15, 2005.

Dov Monderer and Lloyd S. Shapley. Fictitious play property for games with identical interests. *Journal of Economic Theory*, 68(1):258–265, 1996.

Herve Moulin. Dominance solvable voting schemes. *Econometrica*, 47(6):1337–1352, November 1979.

John F. Nash. Equilibrium points in n-person games. In *Proceeding of the National Academy of Sciences*, volume 36, pages 48–49, 1950.

Eugene Nudelman, Jennifer Wortman, Kevin Leyton-Brown, and Yoav Shoham. Run the GAMUT: A comprehensive approach to evaluating game-theoretic algorithms. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 880 – 887, 2004.

Anna Osepayshvili, Michael P. Wellman, Daniel M. Reeves, and Jeffrey K. MacKie-Mason. Self-confirming price prediction for bidding in simultaneous ascending auctions. In *Twenty-First Conference on Uncertainty in Artificial Intelligence*, pages 441–449, July 2005.

Christos H. Papadimitriou and Tim Roughgarden. Computing equilibria in multi-player games. In *Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–91, 2005.

Nicholas C. Petruzzi and Maqbool Dada. Dynamic pricing and inventory control with learning. *Naval Research Logistics*, 49(3):303–325, 2002.

Daniel Reeves, Michael Wellman, Jeffrey MacKie-Mason, and Anna Osepayshvili. Exploring bidding strategies for market-based scheduling. *Decision Support Systems*, 39(1):67–85, 2005.

Dennis I. Robertson. TRANSYT method for area traffic control. *Traffic Engineering & Control*, 10:276–281, 1969.

Robert W. Rosenthal. A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, 2:65–67, 1973.

Robert W. Rosenthal. The network equilibrium problem in integers. *Networks*, 3:53–59, 1973.

L. Julian Schvartzman and Michael P. Wellman. Market-based allocation with indivisible bids. *To appear in Production and Operations Management*, 2006.

Suvrajeet Sen and Larry K. Head. Controlled optimization of phases at an intersection. *Transportation Science*, 31:5–17, 1997.

Arthur G. Sims. The Sydney coordinated adaptive traffic system. In *Urban Transport Division of ASCE Proceedings*, pages 12–27, New York, NY, 1979.

Peter Stone, Michael L. Littman, Satinder Singh, and Michael Kearns. ATTac-2000: An adaptive autonomous bidding agent. *Journal of Artificial Intelligence Research*, 15:189–206, 2001.

Peter Stone, Robert E. Schapire, Michael L. Littman, Ja'nos A. Csirik, and David McAllester. Decision-theoretic bidding based on learned density models in simultaneous, interacting auctions. *Journal of Artificial Intelligence Research*, 19:209–242, 2003.

Saroja Subrahmanyan and Robert W. Shoemaker. Developing optimal pricing and inventory policies for retailers who face uncertain demand. *Journal of Retailing*, 72(1):7–30, 1996.

Jayashankar M. Swaminathan and Sridhar R. Tayur. Tactical planning models for supply chain management. In A. G. de Kok and S. C. Graves, editors, *Supply Chain Management: Design, Coordination and Operation*, volume 11 of *Handbooks in Operations Research and Management Science*, chapter 8. Elsevier, Amsterdam, 2003.

M. Van Aerde, J. Voss, and G. McKinnon. *INTEGRATION Simulation Model User's Guide*. Queen's University, 1989.

John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, second edition, 1947.

Yevgeniy Vorobeychik, Christopher Kiekintveld, and Michael P. Wellman. Empirical mechanism design: Methods, with application to a supply chain scenario. In *Seventh ACM Conference on Electronic Commerce*, pages 306–315, Ann Arbor, 2006.

F. V. Webster and B. M. Cobbe. *Traffic Signals*. Road Research Technical Report 39, Her Majesty's Stationery Office, London, 1958.

Michael P. Wellman and Peter R. Wurman. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, 24:115–125, 1998.

Michael P. Wellman, Peter R. Wurman, Kevin OMalley, Roshan Bangera, Shou de Lin, Daniel Reeves, , and William E. Walsh. Designing the market game for a trading agent competition. *IEEE Internet Computing*, 5(2):43–51, 2001.

Michael P. Wellman, Shih-Fen Cheng, Daniel M. Reeves, and Kevin M. Lochner. Trading agents competing: Performance, progress, and market effectiveness. *IEEE Intelligent Systems*, 18(6):48–53, 2003.

Michael P. Wellman, Amy Greenwald, Peter Stone, and Peter R. Wurman. The 2001 Trading Agent Competition. *Electronic Markets*, 13:4–12, 2003.

Michael P. Wellman, Daniel M. Reeves, Kevin M. Lochner, and Yevgeniy Vorobeychik. Price prediction in a trading agent competition. *Journal of Artificial Intelligence Research*, 21:19–36, 2004.

Michael P. Wellman, Daniel M. Reeves, Kevin M. Lochner, Shih-Fen Cheng, and Rahul Suri. Approximate strategic reasoning through hierarchical reduction of large symmetric games. In *Twentieth National Conference on Artificial Intelligence*, pages 502–508, Pittsburgh, 2005.

Michael P. Wellman, Daniel M. Reeves, Kevin M. Lochner, and Rahul Suri. Searching for Walverine 2005. In *IJCAI-05 Workshop on Trading Agent Design and Analysis*, Edinburgh, 2005.

Michael P. Wellman, Patrick R. Jordan, Christopher Kiekintveld, Jason Miller, and Daniel M. Reeves. Empirical game-theoretic analysis of the TAC market games. In *AAMAS-06 Workshop on Game-Theoretic and Decision-Theoretic Agents*. 2006.

Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

Karl E. Wunderlich and Robert L. Smith. Large scale traffic modeling for route guidance evaluation: A case study. IVHS Program Technical Report 92-08, The Univerisity of Michigan, 1992.

Karl E. Wunderlich, David E. Kaufman, and Robert L. Smith. Link travel time prediction for decentralized route guidance architectures. *IEEE Transactions on Intelligent Transportation Systems*, 1(1):4–14, March 2000.

Karl E. Wunderlich. *Link Travel Time Prediction for Dynamic Route Guidance in Vehicular Traffic Networks*. PhD thesis, University of Michigan, 1994.

Peter R. Wurman, Michael P. Wellman, , and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Second International Conference on Autonomous Agents*, pages 301–308, Minneapolis, 1998.

Peter R. Wurman, Michael P. Wellman, and William E. Walsh. A parametrization of the auction design space. *Games and Economic Behavior*, 35:304–338, 2001.

Sam Yagar and Bin Han. A procedure for real-time signal control that considers transit interference and priority. *Transportation Research B*, 28(4):315–331, 1994.

Fredrik Ygge and Hans Akkermans. Decentralized markets versus central control: A comparative study. *Journal of Artificial Intelligence Research*, 11:301–333, 1999.

Li Zhang, Peter B. Luh, and Krishnan Kasiviswanathan. Energy clearing price prediction and confidence interval estimation with cascaded neural networks. *IEEE Transactions on Power Systems*, 18(1):99–105, February 2003.