Solution Methods for Multiprocessor Network Scheduling Problems, with application to railroad operations

by

James D. Christodouleas

A.B. Woodrow Wilson School for Public Policy and International Affairs, Princeton University (1992)

Submitted to the Sloan School of Management
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

©Charles Stark Draper Laboratory, Inc.

Permission to copy all or part of this work is granted, provided that the copies are not made or distributed for resale, and that the copyright notice and this notice are retained.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author . . . . . .
Sloan School of Management
May 23, 1997

Certified by . . . . . .
John N. Tsitsiklis
Professor, Laboratory for Information and Decision Systems.
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Approved by . . . . . .
Michael J. Ricard
Senior Member, Technical Staff, C.S. Draper Laboratory, Inc.
Draper Technical Supervisor

Accepted by . . . . . .
Thomas L. Magnanti
George Eastman Professor of Management Science,
Professor of Electrical Engineering and Computer Science
Co-director, Operations Research Center

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MAY 30 1997

ARCHIVES

LIBRARIES

Solution Methods for Multiprocessor Network Scheduling Problems, with application to railroad operations

by

James D. Christodouleas

Submitted to the Sloan School of Management
on May 23, 1997, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis explores decomposable methods for a generalization of job shop scheduling which we call the Multiprocessor Network Scheduling Problem (MNSP), with a particular application to what we call the Railroad Network Scheduling Problem (RNSP). In Chapter 1, we introduce key concepts of railroad scheduling, as well as formally define MNSP and RNSP, and introduce the reader to some of the "state of the art" in scheduling for railroad applications.

Chapter 2 discusses what we see as the next step in such research, which is to integrate various scheduling models into a hierarchical decision tool which utilizes these models in an abstract and generic manner. We present a decomposable formulation of MNSP and describe its application to RNSP, as well as detailed analysis of this decomposition and a heuristic approach to finding feasible solutions to MNSP and RNSP. This heuristic is based on a simple averaging technique for calculating fixed points known by some as the *Method of Successive Approximations (MSA)*. MSA empirically finds feasible, though sub-optimal, schedules, and the quality of the schedules it finds can be affected through appropriate use of scheduling "targets."

Chapters 3 and 4 discuss two approaches to generating such targets. The first is based on a large-scale but low-fidelity integer programming formulation of RNSP, which is solved to optimality but requires significant time, memory, and computational resources. The second (Chapter 4) is based on the novel coordination of heuristic control policies within a dynamic programming framework. Computational results demonstrate that it is capable of generating schedules within a few percentage points of optimality, yet consume less than 1% of the time and memory of the IP.

Chapter 5 presents a case-study of the application of approximate dynamic programming (ADP) and heuristics to a Yard Switching and Sequencing Problem (YSSP), which is itself a particular instance of MNSP. Several variations of ADP and heuristic coordination are presented, along with general conclusions and insights.

The thesis concludes in Chapter 6 with a review of the discussion and summary of contributions.

Thesis Supervisor: John N. Tsitsiklis
Title: Professor, Laboratory for Information and Decision Systems,
Department of Electrical Engineering and Computer Science

DRAPER LABORATORY

154-97-JHS
25 June 1997

To: Moses Carr, MIT Library

From: John Sweeney, Director of Education

Subject: Copyright for June 1997 Thesis of James D. Christodouleas entitled "Solution Methods for Multiprocessor Network Scheduling, with Applications to Railroad Operations"

The thesis noted above, title page attached, was inadvertently processed with a non-standard copyright notice. It is recommended that this note be appended to the thesis:

Permission is hereby granted by the Charles Stark Draper Laboratory, Incorporated to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

JHS/lm

Attachment

c: Student File

The Charles Stark Draper Laboratory, Inc.
555 Technology Square, Cambridge, Massachusetts 02139-3563
Telephone: 617 258-1000

Acknowledgments

I would like to acknowledge the contributions of Professor Dimitris Bertsimas and Efthalia Chrysikou, who, as consulting professor and research assistant, respectively, were involved in the initial stages of the work described in Chapter 3 in academic year 1994-1995. In particular, I wish to thank Professor Bertsimas for both encouraging us to consider the Bertsimas/Stock Air Traffic Flow model in the rail scheduling domain, and for suggesting Everett's Method to us in the decomposition strategy of Chapter 3. I would also like to acknowledge consultation with Sarah Stock, who, with Dimitris Berstimas, was the author of the initial application in the air transportation domain of the IP model presented in that same chapter, and provided valuable insights into the formulation.

Andrew Armacost was not only a constant source of ideas and insights about rail operations as an office-mate at Draper between 1993 and 1995, but it was experimental work he had done that made me realize that MSA (see Chapter 2) was a viable alternative to a gradient based approach to CAD coordination. Another Draper office-mate, Tim Kniker, offered tremendous knowledge of railroad operations and was extremely valuable to me throughout the course of my research as both colleague and friend.

This thesis would not have been possible without the extensive financial and technical support of Dr. Milton B. Adams, Dr. Stephan E. Kolitz and Dr. Michael J. Ricard at C.S. Draper Laboratory, Inc. Dr. Ricard was also gracious enough to serve as a fourth reader in my thesis committee, and corrected drafts of this document with tremendous care and precision. He was joined by Professors Cynthia Barnhart and Thomas L. Magnanti, who provided helpful support and guidance as members of that same committee.

I would like to acknowledge an even greater debt to the generous work and impressive talent of Professor John N. Tsitsiklis, who, as thesis supervisor and academic advisor, patiently and forgivingly guided me through four years of academic study, thesis research, and a long and meticulous career search.

Finally, on a personal note, I want to thank Ms. Anne Keliher. Her profound and sincere friendship has made these four years a beautiful time of my life which I'll always treasure.

Contents

1 Introduction 13

1.1 Overview of Thesis . . . 13

1.2 Key Ideas in Railroad Operations . . . 15

1.2.1 Topology of Classification Terminal . . . 16

1.2.2 Classification Process . . . 17

1.2.3 Linehaul Operations . . . 18

1.2.4 Network Operations . . . 19

1.3 Multiprocessor Network Scheduling Problem MNSP . . . 20

1.3.1 Application to Rail Network Scheduling Problem (RNSP) . . . 22

1.3.2 Application to Linehaul Meet-Pass Planning . . . 22

1.3.3 Application to Yard Switching and Sequencing . . . 23

1.3.4 Application to Yard Dispatching . . . 27

1.4 Rail Operations Literature Review . . . 31

1.4.1 Freight Routing, Network Service Design, Blocking . . . 31

1.4.2 Linehaul Decision Modeling . . . 33

1.4.3 Yard Switching and Sequencing . . . 36

1.5 Scheduling and ADP Literature Review . . . 37

1.5.1 Job Shop Scheduling . . . 37

1.5.2 Approximate Dynamic Programming . . . 39

1.5.3 Multiclass Queuing Networks and Rail Operations . . . 41

2 Decomposable Solution of RNSP with Node Planning Tools 44

2.1 Overview . . . 44

2.2 Decomposition of MNSP . . . 45

2.2.1 Definition of Terms . . . 45

2.2.2 Problem Decomposition . . . 46
2.2.3 Combinatorial Abstraction . . . 49
2.3 Normal Gradient Descent over y and λ . . . 52
2.4 Method of Successive Approximations . . . 54
2.4.1 Feasible Schedule as Fixed Point . . . 54
2.4.2 Importance of Objective Function . . . 57
2.4.3 Target Times . . . 59
2.5 Computational Experiments . . . 60
2.6 Summary and Conclusions . . . 62

3 Three IP Models for Railroad Scheduling and Sequencing 63
3.1 Introduction and Overview . . . 63
3.1.1 Chapter Summary and Statement of Intent . . . 63
3.1.2 Chapter Outline . . . 65
3.2 Generic Bertsimas/Stock IP Model (GBS) . . . 65
3.3 First Application: Line Meet-Pass Model (LMPM) . . . 69
3.3.1 Example of linehaul dispatching problem . . . 69
3.3.2 Formulation of Model . . . 70
3.3.3 Segments as Signal Blocks . . . 74
3.3.4 Extensions and Features of Formulation . . . 75
3.3.5 Limitations of LMPM . . . 76
3.4 Second Application: Yard Switching and Sequencing . . . 77
3.4.1 YSSM Formulation . . . 77
3.4.2 Limitations of YSSM . . . 78
3.5 Third Application: Yard Dispatching Model . . . 78
3.6 Solving the IP Models . . . 79
3.6.1 Bertsimas/Stock Polyhedron . . . 80
3.6.2 Branching Strategy . . . 82
3.6.3 Lagrange Decomposition and Everett's Method . . . 83
3.6.4 Updated Branching Strategy . . . 86
3.7 Computational Experiments with Applications . . . 87
3.7.1 Experimental Environment . . . 87

3.7.2 Line Meet-Pass Model . . . 88
3.7.3 Yard Switching And Sequencing Model . . . 91
3.7.4 Yard Dispatching Model . . . 92
3.7.5 Progress of Everett's Method . . . 93
3.7.6 Speeding up the solution . . . 94
3.8 Summary and Conclusions . . . 97

4 Heuristic Scheduling **98**
4.1 Introduction and Motivation . . . 98
4.1.1 Approximate DP with Heuristic Delimited Control Space . . . 98
4.1.2 Chapter Outline . . . 101
4.2 Heuristic Scheduling Methods . . . 101
4.2.1 Scheduling Heuristics . . . 105
4.2.2 Comments about Individual Heuristics . . . 108
4.3 Approximation of Cost-to-Go . . . 111
4.3.1 Features . . . 112
4.3.2 Approximation Architecture . . . 115
4.4 "Learning" to make decisions . . . 115
4.4.1 Updating parameter vector w: Online Learning . . . 116
4.4.2 Updating parameter vector w: Off-Line Learning . . . 118
4.4.3 Rollout, Approximate Rollout, and Limited Lookahead . . . 120
4.4.4 Optimization in Policy Space . . . 123
4.5 Implementation Issues . . . 125
4.5.1 Data Conditioning, Scaling, and Reduction . . . 125
4.5.2 Network-Wide Scheduling . . . 125

4 ADP Case Study: Hump Yard Switching and Sequencing **108**
4.1 Case Study Environment and Solution Approaches . . . 108
4.1.1 Experimental Environment . . . 108
4.1.2 Solution Strategies . . . 109
4.1.3 Utilized Features . . . 111
4.2 Summary of Case Study Conclusions . . . 112

4.2.1 Performance of Heuristics . . . 114
4.2.2 Performance of e-ADP, xtd-ADP, κ-ADP . . . 115
4.2.3 Performance of Rollouts . . . 116
4.2.4 Performance of Approximate Rollout . . . 121
4.2.5 Mean vs. Min based control policies . . . 121
4.2.6 Value of Limited Lookahead . . . 122
4.2.7 Value of Constraining Polynomial Architecture . . . 126
4.2.8 Fitting Approximation Models . . . 126
4.2.9 Training Neural Networks . . . 130
4.3 Comparison of all solution methods . . . 130
4.4 Conclusions and Insights . . . 134
4.4.1 Information Paradox . . . 134
4.4.2 Stochastic vs. Deterministic Problems . . . 135
4.4.3 Comparisons with Exact Optimization . . . 136

5 Conclusion and Summary of Contribution **127**
5.1 Summary of Research . . . 127
5.2 Statement Of Contributions and Conclusions . . . 130

Bibliography **132**

List of Figures

1-1 Overview of Classification Terminal 19
1-2 Example of linehaul track topology 19
1-3 Multiprocessor Network Scheduling Problem 20
1-4 YSSM of Classification Terminal 23
1-5 Path of Inbound Trains 24
1-6 Paths of Normal Blocks 25
1-7 Path of Outbound Trains 25
1-8 Inbound Bringing Last Cuts 28
1-9 Two Outbound and Four Inbound 28

2-1 Two Trains waiting for a segment. 46
2-2 Simple example of decision space $\mathcal{P}$. 47
2-3 Illustrative Scheduling Scenario 47
2-4 One Train, Two Line Network 53
2-5 Price Critical Planning Scenario 57
2-6 Line Planning Scenario 59

3-1 Single Train Traversing 7 Segments 66
3-2 Bertsimas Stock Decision Variables 66
3-3 Simple Planning Problem and Data 70
3-4 Right-Right-Left Meetpoint Configuration 72
3-5 Right-bound traffic takes siding 72
3-6 Progress Everett's Method 95

4-1 Illustration of Overlapping Hump Job Sets . . . 109
4-2 Illustration of Min Slack Limitation . . . 111
4-3 Illustration of MRP Heuristic . . . 111
4-4 Illustration of Approx DP Trajectory . . . 116
4-5 Dangers of Cost-to-Go Approximation . . . 124

4-1 Heuristic Relative Error . . . 114
4-2 Heuristic Relative Error . . . 115
4-3 Histogram of Heuristic ROG . . . 116
4-4 Best Heuristics vs *e*-ADP . . . 117
4-5 κ-ADP vs. *e*-ADP, *xtd*-ADP. *a*-H . . . 118
4-6 Histogram of κ-ADP, *e*-ADP, *xtd*-ADP, *a*-H . . . 118
4-7 Rlout vs. Best Heuristic . . . 119
4-8 Rlout vs. *e*-ADP . . . 120
4-9 Rlout vs. IP . . . 120
4-10 Histogram of Rollouts . . . 121
4-11 Neural Networks minus *e*-ADP . . . 122
4-12 Polynomial ADP with no lookahead: min minus mean Control . . . 123
4-13 Polynomial ADP with Lookahead: Min minus Mean Control . . . 123
4-14 Neural Net ADP's: Min minus Mean Control . . . 124
4-15 Lookahead Penalty: NN vs Polynomial: ROG . . . 125
4-16 LKHD-2 vs. LKHD-4 . . . 125
4-17 Value of Constraints . . . 126
4-18 Neural Network with Lookahead minus LKHD-4 . . . 127
4-19 Polynomial Coefficient Plots . . . 128
4-20 Predictive Power: Linear and Neural Network Architectures . . . 129
4-21 Predictive Power: Quadratic and Cubic Architectures . . . 129
4-22 Gradient Descent Record on MD Features . . . 131
4-23 Gradient Descent Record on MRP Features . . . 131
4-24 Gradient Descent Record on WMS Features . . . 132
4-25 Histogram of *r*-H, average over *h*, and *a*-H ROG . . . 132
4-26 Information Paradox . . . 135

4-27 Linear ARLT vs. Heuristics 137
4-28 Quadratic ARLT vs. Heuristics 137
4-29 Neural Net with No Lookahead vs. Heuristics 138
4-30 Neural Network with Lookahead vs. Heuristics 138

List of Tables

1.1 YSSM Topology 24
1.2 Topology of Yard Dispatching Model 30

2.1 Two Yard Scenario Data 57
2.2 Single Fixed Point when $\hat{\lambda_{1B}} = 10$. 58
2.3 Convergence of MSA under scenarios with varied congestion 61
2.4 Comparison of MSA with and without Targets 62

3.1 Bounds from LP relaxation, BS Polyhedron 81
3.2 Combined Network-Simplex vs. Simplex Method 82
3.3 Initial Everett's Price Update Method 85
3.4 Extended Everett's Price Update Method 86
3.5 Machines on which computational experiments were run 88
3.6 Multipliers to Generate Machine (b) CPU Times 88
3.7 LMPM Computational Experiments 89
3.8 LMPM CPU Times as Machine (b) times 89
3.9 YSSM Computational Experiments 91
3.10 YSSM CPU Times as Machine (b) times 92
3.11 YDM Computational Experiments 92
3.12 YDM CPU Times as Machine (b) times 93
3.13 YSSM Computational Experiments–Simplified IP and Branching Rule 96
3.14 Simplified YSSM CPU vs. Standard YSSM, Machine (b) times 96
3.15 Feasible Schedule Heuristic 96

4.1 "Expert" led Optimization in Policy Space 124

4.1 Polynomial Approximation Errors 127
4.2 Neural Network Approximation Errors 130
4.3 ROG Averages and Spreads for all approaches 133

Chapter 1

Introduction

Many of the most important problems in operations research involve the scheduling of tasks over a network of processors. In fact, one very elementary version of such problems, the min-makespan job shop scheduling problem, is one of the most notoriously challenging problems in the O.R. and computer science communities. In this thesis, we address scheduling issues within the railroad industry from the perspective of optimization based scheduling. We introduce a generalization of the job shop scheduling problem, which we call the *multiprocessor network scheduling problem* (MNSP), as well as a discrete-time special case of MNSP, which we call MNSP*. We formulate several important railroad scheduling problems as examples of MNSP and MNSP*, then attack these problems with various methodological strategies. These problems are the *rail network scheduling problem* (RNSP), the *line meet-pass problem*, the *switching and sequencing problem*, and the *yard dispatching problem*. We present, in this chapter, formal models to solve the last three problems, which we call LMPM, YSSM, and YDM, respectively.

1.1 Overview of Thesis

In Chapter 2, we discuss a generic decomposition strategy for RNSP, which is formulated first for MNSP. This decomposition strategy has two notable features. First, it utilizes single-processor scheduling models (what we call *node planners*) as its subproblems. Second, it coordinates these node planners to generate feasible, but not necessarily optimal, schedules. However, the quality of these schedules can be affected by feeding the node planners an appropriate set of scheduling

"targets."

The first feature is extremely useful, as in many examples of MNSP, the single-processor scheduling problem is much easier than the network scheduling problem. In the case of railroads, for example, while there exists to date no successful solution to RNSP, there has been promising work on scheduling tools for individual lines or yards, which are the "processors" of RNSP. In this case, the node planners correspond to what are called *computer aided dispatching* (CAD) tools, and this thesis presents the first systematic effort to leverage progress being made in the CAD domain to the problem of generating network-wide schedules. In job shop scheduling, the node planners correspond to single-machine scheduling problems, for which there exist many effective solution techniques despite the fact that such problems are often strongly $\mathcal{NP}$-hard.

The second feature implies the need for tools that will generate appropriate schedule targets, and this need motivates the work of Chapters 3 and 4. In both chapters, we utilize models of rail networks formulated as examples of networks of MNSP*, propose generic solutions for MNSP*, then implement these solutions for our railroad models. Chapter 3 presents an integer programming formulation of MNSP*, which is then applied to the solution of YSSM, a network of LSSM's, and a network of YDM's. That chapter demonstrates the potential efficacy of an IP model to the solution of such problems while at the same time highlighting important disadvantages as well. The principal one is that the computation speed of the IP models is extremely non-robust in problem size and schedule congestion.

Motivated by such problems, Chapter 4 explores a battery of scheduling heuristics for MNSP* and generates an approximate dynamic programming (ADP) formulation in which the control decision at any time-period is to select one heuristic from this battery to control the node for the duration of the time-period. The control decisions are made by simulating each control for one time-period, then using a feature based pricing function to compare the products of each heuristic. Chapter 4 presents a detailed discussion of scheduling heuristics, features, and other issues relevant to the implementation of this ADP formulation to MNSP*, YSSM and LMPM.

Chapter 4 presents a case study in the application of heuristic-based ADP to MNSP*. We generated 100 random variations of YSSM and utilized forty two different ADP solution approaches to solve it. The strengths and weaknesses of each of these are discussed.

Finally, Chapter 5 summarizes the thesis and enumerates the conclusions and major contributions of the work.

1.2 Key Ideas in Railroad Operations

Unfortunately for the reader new to railroad operations, there is a great deal of terminology which is loosely defined throughout the industry. For example, the term "classification yard" in practice refers to both the entire classification terminal (as defined in this thesis), as well as the part of the terminal where classification occurs. For this reason, we will introduce the following terminology, and use it throughout this thesis.

Cut: A group of cars with common origin and destination. Such cars can be assembled into a contiguous group at origin that need never be taken apart until they reach their destination.

Block: A group of cuts at a terminal which share as a common intermediate destination a classification terminal. These will typically share the same track in the *classification yard*, which is the part of a *classification terminal* where classification takes place.

Freight-Routing Plan: For every origin-destination pair (O, D), the decision of how to route freight from O to D. It is typically determined at the strategic level.

Blocking Plan: At every terminal and for every destination, the decision of which block to assign freight bound for that destination. Also typically determined at the strategic level. We note that a blocking plan specifies the freight-routing plan, but the reverse is not true.

Terminal: Any location in a railroad network where cuts can originate or to which they can be destined.

Classification Terminal: Any terminal where cuts can change blocks.

Switching Terminal: Any terminal where cuts can pass through and change trains, but without changing blocks. Thus, for any terminal, its designation as either switching terminal, classification terminal, or plain terminal is specified by the blocking plan.

Hump Yard: Any terminal with a "hump" (see Section 1.2.2). In practice, a hump yard is always a classification terminal, but the reverse is not true.

Flat Yard: Any terminal without a hump. In practice, a flat yard is often a switching yard. For this thesis, however, flat and hump yards will designate physical distinctions, while switching and classification terminals will designate operational distinctions.

Line: Any part of the network outside a terminal under the control of a single dispatching authority.

Planning Train: A group of blocks traveling together. A planning train has an itinerary, a consist of blocks, and a frequency.

Train Plan: A plan for the movement of blocks. The train plan defines all the planning trains.

Train: A specific *instance* of a planning train within any planning period. For example, if the train plan specifies that there be a daily Chicago - St. Louis planning train, and the planning horizon is 5 days, then this horizon has 5 trains for the single Chicago - St. Louis planning train. Two very important ingredients of actual trains, crews and locomotives, are not explicitly modeled in this thesis.

Station: Any location in a network where the time of arrival of a train is important. Terminals are obviously stations, as are crew change points and boundaries between linehaul dispatcher regions.

Trip Plan: For each (O, D) pair, specifies the set of planning trains that will carry freight from O to D. Determined by the blocking plan and train plan.

Car-Train Assignment: For every car (shipment), specifies which *trains* will carry it from its origin to its destination. Car-train assignment and trip plan are often used synonymously in practice, but we choose to distinguish them here.

Schedule: For every train, specifies its departure and arrival time from each station in its itinerary.

Terminal Train Master The planner responsible for making dispatching decisions from a terminal.

Linehaul Dispatcher The planner responsible for making meet-pass and other movement authorizations across a line.

Yard Master The planner responsible for the coordination of crews and engines within a classification terminal.

Some of the most important strategic decisions to be made by a railroad include the definitions of its blocking plan and train plan. For example, it is important to determine how cars on an outbound train are grouped. If an outbound train from Chicago is headed to St. Louis, and St. Louis is a flat yard, it will make a significant difference if all cars heading on from St. Louis to Memphis, say, appear in a contiguous block within that St. Louis train, (minimizing the amount of reshuffling to be done in St. Louis), rather than sequenced in whatever random order they arrived at the Chicago yard (which would be the scheme that would minimize work in Chicago). Thus there are important trade-offs to be made regarding how much "sorting" to do at each yard. However, these are longer-term decisions, and for the purposes of our work, are considered inputs to our scheduling models.

Given these inputs, one must then generate a car-train assignment and schedule. Ideally, one would optimize over both simultaneously, since fixing a schedule limits the space of available car-train assignments, and fixing a car-train assignment limits the space of feasible schedules. In this thesis, however, we take the position that the car-train-assignment should be done first, off-line, and we therefore concern ourselves only with the task of generating a quality schedule that satisfies this assignment.

1.2.1 Topology of Classification Terminal

A classification terminal typically comprises three major blocks of track, illustrated in Figure 1-1.

Receiving Yard: Where inbound trains arrive, and are inspected.

Classification Yard: Where cars are separated into distinct groups, or "blocks."

Departure Yard: Where blocks from the classification yard are placed together and combined with locomotives to form outbound trains.

Although there are many variations on this theme (some classification yards may have one large, combined receiving/departure yard, for example), these three areas are the key ingredients of a classification terminal.

1.2.2 Classification Process

In a major classification terminal, trains arrive carrying blocks and either pass directly through the yard with minimal handling, or must be disassembled, which means that their cuts (or cars) are separated apart. Before they are disassembled, they are inspected by a yard crew for mechanical reliability, which can be a time-consuming process, and the locomotives powering the inbound train are removed and sent to a mechanical shop for inspection, routine maintenance, and any necessary repair.

The largest classification terminals are typically hump yards, where the car-separation process is called "humping." In such yards, the inbound train is pushed by an auxiliary locomotive (the "hump engine") up an incline towards the crest. As each car passes over the crest (or "hump"), the car is released from the train and is allowed to accelerate down a gradient toward the area of a classification terminal where the classification tracks are. This area within the classification terminal is called the "classification yard." Alternatively, because of the physical topology described above, it is also often called "the bowl."

Because of the slope of the track from the hump into the bowl, cars recently released from the train accelerate quickly and hence separate from the cars behind. This separation allows the yard crew to move the switches on the track that send different cars onto different tracks in the bowl. See Armacost [4] for a more detailed description and model of the classification process.

Most real terminals, however, are not hump yards, but rather flat yards. Here, disassembling and reclassifying trains is much more difficult, as blocks of cars must be pulled from trains one at a time with locomotives. They do not have the benefit of a gradient down which they can allow cars to flow and naturally separate. Major reclassification is done mostly at hump yards, with hopefully only minor reclassification to be done at the many, smaller, flat yards, although there are examples

of large flat yards where major reclassification is done. Conrail's largest yard is in fact a flat yard.

Whatever the type of yard, cuts flow into the classification yard onto specific tracks where they make up the block to which they are assigned. Once completed, the blocks are pulled out of the classification yard, through the assembly leads into the departure yard (by the "pull engine" or "assembly engine"), and combined to form outbound trains, which are then also inspected and finally released. This pulling process is what is known as "assembly," and along with inbound and outbound inspection, is the most time-consuming process in the yard.

There are two decision-making authorities within the yard. The yardmaster determines when and in what order to inspect and then begin disassembling inbound trains, and when and in what order to begin pulling blocks from the classification yard to assemble and then inspect outbound trains. The terminal train master is responsible for scheduling the final dispatching of the fully assembled and inspected trains and handing over authority to the appropriate linehaul dispatcher.

In this sense, the terminal train master has *de facto* authority to modify the car-train assignment, simply by deciding whether to release a train before all its intended consist has been assembled. In our work, we are particularly interested in the decisions of the yardmaster, and in the sequencing of outbound departures by the terminal train master. Namely, can we manage yard operations in such a way that all cars make their intended trains, and do so in some optimal way? We call this problem the *Yard Switching and Sequencing Problem.*

Alternatively, we might wish to simply concentrate on the decisions of the terminal train master, without explicitly modeling the yard master. In this case, we wish to find a set of departure times for the terminal which are consistent with the arrival times of inbound trains, and under which there exists, with "high probability," a switching and sequencing plan that has every car making its intended outbound train, without actually specifying that plan. We call this problem the *Yard Dispatching Problem.*

1.2.3 Linehaul Operations

A linehaul dispatcher region, or line, typically comprises a long stretch of single or double (and sometimes triple or more) main-line track, as well as sidings and spurs, where trains can meet, pass, and overtake each other on the main line (Figure 1-2). The decisions of the dispatcher are to determine which trains take which sidings and when (or, where and how trains should meet, pass, and overtake each other). The linehaul dispatcher is often called the "shift master" or "trick

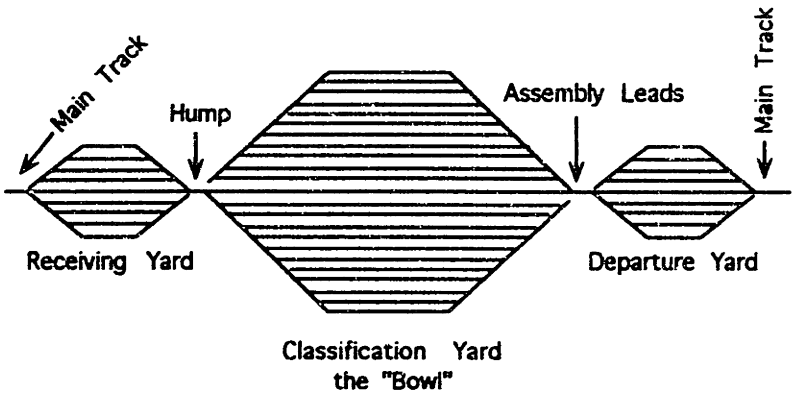


Figure 1-1: Overview of Classification Terminal (graphic from [4])

Figure 1-2: Example of linehaul track topology

master."

1.2.4 Network Operations

We now define the Rail Network Scheduling Problem (RNSP) as follows. We are given a set of trains with fixed itineraries through a network of lines and terminals. We are also given for each train its release date, starting locations, and intended consist, as well as the location and release date of every car not yet assembled to a train. Given this information, we must determine a set of train arrival times for each node in our network that:

1. is feasible under the operating conditions of the railroad,
2. allows every train to carry its intended consist,
3. optimizes an objective function which is separable by train and is a function only of train arrival times.

As we said before, we do not model the scheduling of crews and locomotives. Such scheduling is of course very important. According to Martland [52], approximately one fourth of all delays in terminals are caused by crew shortages and crew rest requirements. Good planning of crews,

therefore, is crucial to train reliability. In this thesis, however, we take the position that a prerequisite to the effective planning of crews and locomotives is the generation of "good" schedules. Therefore, although we do not explicitly model these elements in our work, by establishing feasible schedules, we make it easier to schedule crews and locomotives and thus minimize the chances that crew or locomotive shortages will cause schedule failures. In this sense, schedule reliability is a self-reinforcing phenomenon. In fact, according to Martland, *et. al.*, railroad task forces for two decades have "identified the development and implementation of feasible operating plans as the critical step toward improving rail freight service reliability." [54, 50]

1.3 Multiprocessor Network Scheduling Problem MNSP

We now formally introduce our generalization of the job shop scheduling problem mentioned earlier, which we call the *Multiprocessor Network Scheduling Problem* (MNSP). We have a network of

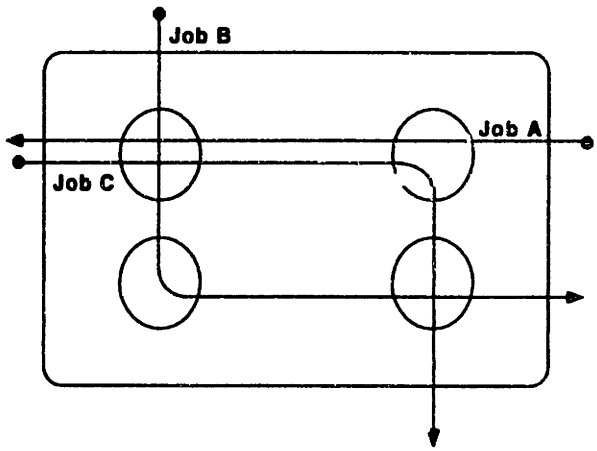


Figure 1-3: Multiprocessor Network Scheduling Problem

processors, or *nodes*, and a set $\mathcal{F}$ of jobs. Each job has a distinct itinerary through the network. This is the canonical job shop scheduling problem. In addition to this, however, each node may, at any given time, be in one of a set $\{s_1, s_2, \ldots, s_k\}$ of states which determine the speed and types of jobs the node may process. This state may be a control decision, or may be given as exogenous data. Moreover, we specify finite buffers preceding each node, and state that nodes may handle multiple jobs simultaneously. In addition to this, we allow for a set $\mathcal{R}$ of precedence constraints $r = ((f', j'); (f, j); s)$, which state that a job f may not start at a node j until s time-units after job

f' has completed processing at node j'. Finally, the amount of time job f must spend on node j is not fixed, as is usual in the canonical job shop scheduling problem, but is a function of the amount of "competing" traffic at the node at the time job f arrives. We might call this function $T_{f,j}(\vec{x}_j)$, where $\vec{x}_j$ is a vector of arrival times of jobs at node j.

Similarly, we define MNSP* to be identical to MNSP, except that $T_{f,j}$ is independent of $\vec{x}_j$, and time is discrete.

Both problems involve the following data, to which we will refer in Chapters 3 and 4.

Data for MNSP:

$\mathcal{F}$: set of jobs $f \in \{1, \cdots |\mathcal{F}|\}$

$\mathcal{T}$: set of time periods $t \in \{1, \cdots |\mathcal{T}|\}$

$\mathcal{J}$: set of processors $j \in \{1, \cdots |\mathcal{J}|\}$

$\mathcal{R}$: set of precedence relationships $r = ((f_r, j_r); (f'_r, j'_r); s_r)$

N_f: number of processors in job f's path

$P(f,k)$: k^{th} processor in job f's path.

$\kappa_f(j)$: The "index" or element number of processor j in f's path; satisfies the relationship $j = P(f, \kappa_f(j))$.

$\nu_f(j)$: The *next* processor in f's path after processor j, i.e. $\nu_f(j) = P(f, \kappa_f(j) + 1)$.

$\pi_f(j)$: The *previous* processor in f's path before processor j, i.e. $\pi_f(j) = P(f, \kappa_f(j) - 1)$.

$C_j(t)$: capacity of processor j at time t

r_f: release time of job f from origin $P(f,1)$.

d_f: scheduled due time of job f at destination $P(f, N_f)$

s_r: needed buffer time between jobs f, f', having some precedence relationship $r \in \mathcal{R}$, $r = ((f_r, j_r); (f'_r, j'_r); s_r)$.

l_{fj}: unimpeded processing time of job f on processor j

T_f^j: range of feasible times t for job f to be at processor j

c_{ft}: cost corresponding to the arrival of job f at its destination at time t

Note that origins and destinations are also modeled as "processors," even though they could very well be dummy-processors which do nothing (such as a global source or sink). Note also that the data for MNSP* will be identical, except that t will take only discrete values; data such as d_f, r_f, s_r, and $l_{f,j}$ must naturally be integral, and T_f^j is interpreted as a set of discrete time periods rather than range of times.

1.3.1 Application to Rail Network Scheduling Problem (RNSP)

Imagine a railroad network, and consider every train to be a job and every terminal, port, intermodal facility, and linehaul dispatching region to be a node. Every train, of course, has a distinct itinerary through this network of nodes. The states a node may be in might correspond to the time of day or other operating conditions. For example, some terminals have night shifts which are smaller than the day crews, and can therefore process trains at a much slower rate. The buffers preceding each node in this case are zero; each train immediately enters processing in one node when it finishes processing in a preceding one. The "processing" of each train by a node corresponds to the physical traversal of the node by the train as well as any other operational work that must be done, such as inspections, maintenance, adding of cars, etc. Of course, each node may process more than one train at a time, and the amount of time it takes a train to traverse a node is obviously a function of the traffic at the node. Crowded lines, for example, will take more time to traverse than empty ones. Finally, the precedence constraints correspond to relationships between trains. For example, we might state that a certain train may not pass a given crew change point until another train carrying its new crew has arrived.

1.3.2 Application to Linehaul Meet-Pass Planning

For a linehaul dispatcher region, we specify three types of processors: mainline track, track with sidings, and spurs. The processing time for each processor is the free running time of each train across the corresponding section of track. The state of the processor, for single track, designates the current direction of permitted traffic across it at any time. Precedence constraints may correspond to restrictions of train movements due to crew changes or linehaul car switches. We call a specification of MNSP* in the linehaul domain the *Line Meet-Pass Model* (LMPM).

The way we model track with sidings is extremely important. We call them meetpoints and represent them as segments with minimum travel time equal to zero. In other words, sidings are reduced to zero-dimension entities along the track where at any time, a certain number of trains may "disappear," allowing other traffic to flow past them. The number of trains that may be on a siding at any one time depends on the physical length of the siding. This representation of the track topology is similar to that in [43] and [57].

Since on any segment of track there can be only two orientations, we divide our set of trains $\mathcal{F}$ into two sets: those traveling in one direction, which we denote with $\rightarrow$, and those traveling in

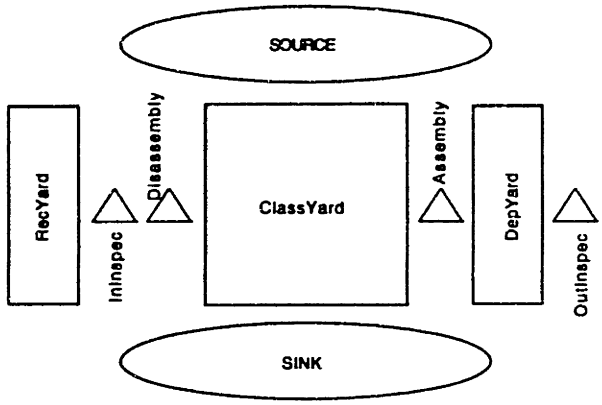


Figure 1-4: YSSM of Classification Terminal

the opposite direction, denoted with $\leftarrow$ (call these two sets $\mathcal{F}^{\rightarrow}$ and $\mathcal{F}^{\leftarrow}$). The data for LMPM is therefore:

Data

All the data from MNSP*, where "jobs" are trains and "processors" track segments or meetpoints.

$\mathcal{M}$: set of meetpoints m ($\mathcal{M} \subset \mathcal{J}$)

1.3.3 Application to Yard Switching and Sequencing

Within a classification terminal, there are three types of jobs: inbound trains, outbound trains, and blocks, and each of these is made up of distinct cuts. Our representation of a classification terminal looks something like that of Figure 1-4.

Processors in this application are not segments of track as in LMPM, but either physical areas or operational processes of the terminal. In YSSM, the classification terminal looks something like a flowshop, with inbound trains, blocks, and outbound trains all flowing in the same direction through the terminal. We therefore have no need for the notion of direction which was so important in LMPM, or any other processor "state variables" s_k.

Each terminal has as major areas those listed in Table 1.1, which may have associated with it a duration, in units of 15 minutes, and a capacity. This data is for illustrative purposes only, but roughly corresponds to operating statistics for Conrail's Selkirk yard, as reported in [4]. A terminal with two inbound inspection crews instead of one would have a capacity of 2 for processor 3 in Table 1.1. Similarly, the capacity of processor 5 depends on the number of classification tracks. The capacities of processor 2 and processor 7, however, are a bit more complicated. Since trains being

Processor ID	Name	Duration	Capacity
1	Source	0	∞
2	RecYard	0	20
3	InInspec	8	1
4	Disassembly	3	1
5	ClassYard	0	70
6	Assembly	2	1
7	DepYard	0	20
8	OutInsp	8	1
9	Sink	0	∞

Table 1.1: YSSM Topology

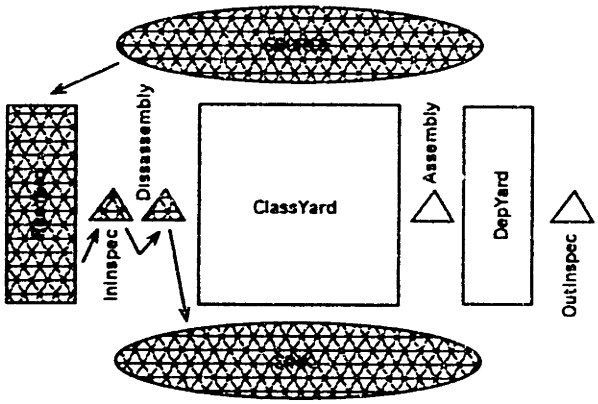


Figure 1-5: Path of Inbound Trains

inspected or disassembled normally still take physical space in the Receiving Yard, for example,

$$\begin{pmatrix} \text{Capacity} \\ \text{processor 2} \end{pmatrix} = \begin{pmatrix} \text{\# tracks in} \\ \text{receiving yard} \end{pmatrix} - \begin{pmatrix} \text{\# disassembly} \\ \text{processes} \end{pmatrix} - \begin{pmatrix} \text{\# outbound} \\ \text{inspection crews} \end{pmatrix}.$$

Similarly,

$$\begin{pmatrix} \text{Capacity} \\ \text{processor 7} \end{pmatrix} = \begin{pmatrix} \text{\# tracks in} \\ \text{departure yard} \end{pmatrix} - \begin{pmatrix} \text{\# assembly} \\ \text{engines} \end{pmatrix} - \begin{pmatrix} \text{\# outbound} \\ \text{inspection crews} \end{pmatrix}.$$

In a real-time setting, the true capacity depends also on the lengths of each train and track, but we make this simplification here.

The three principal types of trains (Inbound Trains, Blocks, and Outbound Trains) each have distinct paths through this network, demonstrated in Figures 1-5, 1-6, and 1-7, respectively. Inbound trains start at the Source (processor 1), then travel to the Receiving Yard (processor 2), through

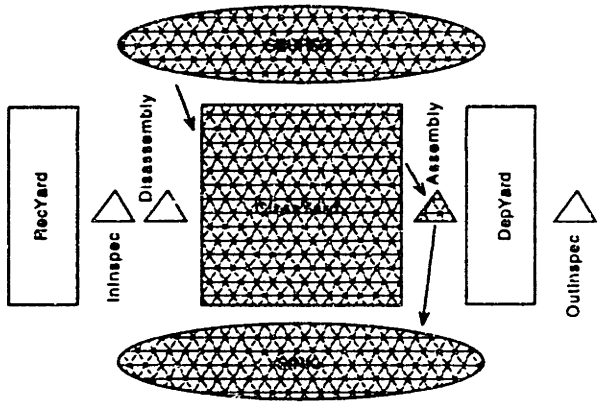


Figure 1-6: Paths of Normal Blocks

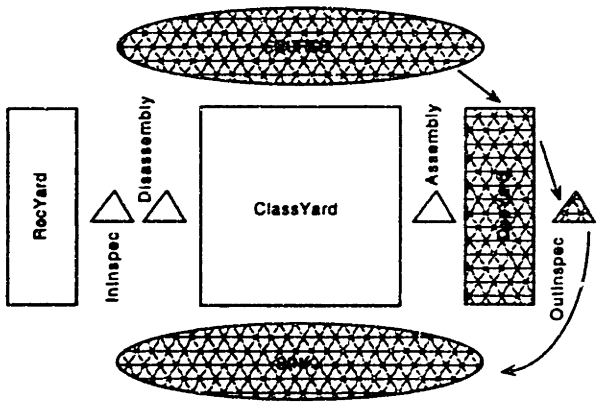


Figure 1-7: Path of Outbound Trains

Inbound Inspection (processor 3), then Disassembly (processor 4), before departing to the Sink (processor 9). There are also "blocks" which really correspond to cuts whose final destination is this terminal, which we call Dest Blocks, and these travel from Source to ClassYard to the Sink, never entering the Assembly Leads. These blocks will normally have objective coefficients attached to their arrival at the Sink, since they correspond to the delivery of goods. Finally, blocks corresponding to cuts whose origin is this terminal, called Origin Blocks, travel from Source to Classification Yard to Assembly Lead just as Normal Blocks do, but they will be treated differently when we define precedence constraints below.

The dynamics of the terminal are captured by internal precedence constraints, which enforce the following rules:

1. ***Inbound Train vs. Block:*** An inbound train i may not enter Disassembly until all blocks b comprising cars

which are on train i have exited the Source and entered the ClassYard, thus identifying and reserving the space where the inbound train will send cars.

2. *Block vs. Trains:* A normal or origin block b may not enter Assembly until the outbound train carrying it has entered the Departure Yard, and a normal or destination Block b may not leave Classification Yard until all inbound trains carrying b's cuts have been completely disassembled.

3. *Outbound Train vs. Block:* An outbound train o may not leave Departure Yard until all its blocks have passed through Assembly Leads.

In practice, the sequence of blocks in an outbound train is partially or wholly specified in a blocking plan. Such constraints can easily be modeled in YSSM by adding appropriate precedence constraints between blocks at the assembly process, and would in principle simplify decision making by restricting the solution space. Where such sequencing is not specified, however, YSSM will generate a locally optimal sequence itself. In what follows, we choose to allow YSSM maximum flexibility to set the block sequences for each outbound train.

The formulation is defined by the following data:

Data for YSSM

Data from MNSP*: Defined appropriately.

$\mathcal{I}$: set of inbound trains i

$\mathcal{B}$: set of normal blocks b

$\mathcal{OB}$: set of origin blocks b

$\mathcal{DB}$: set of destination blocks b

$\mathcal{O}$: set of outbound trains o

$\mathcal{C}$: set of cuts traveling through terminal

$\mathcal{C}(i)$: set of cuts arriving on inbound train i

$\mathcal{C}(o)$: set of cuts departing on outbound train o

$\mathcal{C}(b)$: set of cuts comprised by block b. For blocks $b \in \mathcal{OB} \cup \mathcal{DB}$, $|\mathcal{C}(b)| = 1$.

s_{bi}: needed buffer time between normal block or dest block b entering (i.e. space being allocated in) the classification yard and inbound train i entering the disassembly.

s_{bo}: needed buffer time between normal or origin block being pulled through assembly leads and outbound train b starting outbound inspection.

s_{ib}: needed buffer time between inbound train i finishing disassembly and either Normal Block b entering assembly, or dest block b arriving at sink.

s_{ob}: needed buffer time between outbound train o entering departure yard and normal or origin block b entering assembly.

1.3.4 Application to Yard Dispatching

YSSM may be more complicated than necessary. Our objective is a terminal model which captures the essential combinatorial constraints involved with trains moving through classification terminals, without necessarily making decisions about operations in the classification yard. Namely, we are interested in a model which will tell us how trains optimally arrive and depart from the terminal. Moreover, we would like to model terminals more general than hump yards or classification terminals

We therefore present a simplification of YSSM which is both a more economical description of the switching and sequencing problem and a more general model of terminal operations as a whole, modeling possibly flat yards and terminals as well as large classification terminals. This is our third and final application of MNSP* to rail scheduling, and we call it the *Yard Dispatching Model* (YDM).

Modeling The Dispatching Process

YDM is very similar to YSSM, except that we no longer explicitly model the classification yard, or the cuts and blocks. Rather, what happens there is captured by judicious use of precedence constraints between inbound and outbound trains, as well as a re-interpretation of the Switch and Assembly processes.

In order to create YDM, we wanted to model constraints on the departure of outbound trains from a terminal as a function of the arrival times of inbound trains. There are many ways one might do this. One initial idea is to place precedence constraints between the departure of any outbound train and the arrival of all inbound trains carrying its cars. For example, we might say that an outbound train o may not begin outbound inspection at a hump yard until s_{io} time periods after the humping of each inbound train i carrying one or more of its cars, where s_{io} is a reasonable estimate of the time to hump train i and pull that o's cars from the bowl, and would probably be a function of the number of cuts on both i and o.

Consider, however, what might happen in the planning scenario of Figure 1-8. In a terminal with multiple outbound inspection crews, for example, multiple trains would be able to leave simultaneously, which is impossible given that blocks must be pulled onto departure tracks one at a time.

This motivates an attempt to capture explicitly the time required to perform assembly of trains, and the fact that only N blocks, where N is the number of assembly engines and leads, can be pulled from the classification yard, or around the flat yard, at any one time. How can we do this

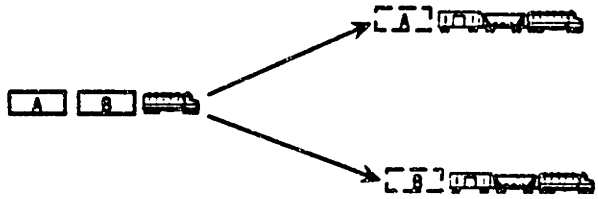


Figure 1-8: Inbound train arrives carrying the last cars to complete two different outbound trains. Under our constraints, both these outbound trains would be eligible to enter outbound inspection at the same time, and the terminal would have them depart as close together as the time for outbound inspection permits.

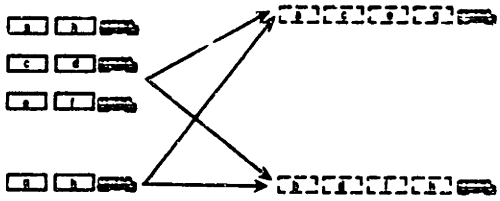


Figure 1-9: There are two outbound trains, both taking cuts from four different inbound trains. Three of the four trains arrive early in the morning, and the fourth arrives late in the evening.

without explicitly modeling the existence of blocks and cuts? We add a process to the terminal called Assembly, just as in YSSM, but in this case it is modeled as a *process* which the outbound train must undergo before outbound inspection, rather than the physical track through which a block must be pulled.

If

T_p = ordinary time to pull and connect a single block to an outbound train

T_h = ordinary time to disassemble a single inbound train

$N_b(o)$ = number of blocks on outbound Train o

we might say that the duration of Assembly for outbound train o is $T_p \times N_b(o)$. This is one way we might explicitly capture the very time-consuming assembly operation without worrying about scheduling and sequencing blocks through the classification yard and assembly leads, and without relying on a blunt standard delay generator.

Unfortunately, this approach can also be inappropriate. Consider for example the scenario of Figure 1-9. Under the new model, only one of the two outbound trains could be assembled during the day; the second would have to wait to begin assembly only late in the evening after the first was complete. We therefore completely lose the fact that we can partially assemble both trains during the day, so that when the last inbound train arrives, we need only pull the last block onto each of our two outbound trains before dispatching both of them. This then suggests that the appropriate

duration for Assembly should be the time to pull just one (the last, $N_b(o)$'th) block, not all blocks on a given outbound train. But where do we account for the previous $N_b(o) - 1$ blocks?

We can do so at Disassembly. Imagine, for example, slowing down the hump at a hump yard, so that disassembly is as time consuming as assembly. Specifically, if

$N_b(i)$ = number of blocks on inbound train i

then we set the duration of Disassembly for inbound train i to be: $T_p \times (N_b(i) - 1) + T_h$, and the duration for the assembly process simply to T_p. In this way we again introduce a process to YDM which, while not corresponding to any unique element in the classification procedure, does have the effect of helping us mimic real constraints on train dispatch times. In this case, we capture the fact that, before any train may leave, all the blocks on that train must have been pulled, although we model the "pulling" of the first $N_b - 1$ blocks as an action occurring on the inbound, not the outbound, train. The key point is that we still account for all the work that must be done in a terminal before an outbound train can be dispatched, and the advantage of this is that we capture the effect of being able to start and stop assembly on a given outbound train, and thus simulate the building of multiple trains simultaneously using the same pull engine.

The disadvantage, unfortunately, is that we are *forced* to do this, and lose the ability to hump a sequence of inbound trains very quickly while using the trim engine to pull cars for one, possibly urgent, outbound train. In such a case, slowing down the disassembly process to account for work we know will be done eventually during the assembly process can be a costly simplification.

We therefore are left with two extremes. Where we are more likely to assemble multiple outbound trains with each inbound hump, appropriate durations for Disassembly and Assembly are, respectively, $T_h + T_p \times (N_b(i) - 1)$ and T_p. Where we are more likely to assemble a single outbound train with a sequence of multiple humped inbound trains, appropriate durations for Disassembly and Assembly are T_h and $(T_p \times N_b(o))$, respectively. Our solution in the general case is to set these times to some convex combination of the two:

$$\begin{bmatrix} \text{Disassembly Time} \\ \text{Assembly Time} \end{bmatrix} = \alpha \begin{bmatrix} T_h \\ T_p \times N_b(o) \end{bmatrix} + (1 - \alpha) \begin{bmatrix} T_p \times (N_b(i) - 1) + T_h \\ T_h \end{bmatrix}$$

where the parameter α should depend on the traffic profile of the particular terminal, and should be thought of as a modeling parameter to be fit by the user of YDM.

With this background complete, YDM comprises the segments listed in Table 1.2. In YDM,

Segment ID	Name	Duration	Capacity
1	Source	0	∞
2	RecYard	0	20
3	InInspec	1	1
4	Disassembly	To be fit	1
5	Assembly	To be fit	1
6	DepYard	0	20
7	OutInsp	2	1
8	Sink	0	∞

Table 1.2: Topology of Yard Dispatching Model

there are now just two types of trains: inbound and outbound. Inbound trains travel from Source to RecYard, to Inbound Inspection, through Disassembly, and out to Sink. Outbound trains travel from Source to Assembly Lead, to Departure Yard, through Outbound Inspection, and on to Sink. In many flat yards, the Receiving and Departure Yards are the same physical entity, so we must be sure to combine capacity constraints involving these areas into one. Also, our interpretation of Disassembly as an abstract delay generator rather than a physical area is important as we model flat yards as well. In both hump yards and large flat yards, we shall interpret Segment 4 (Disassembly) to be the process of pulling and separating blocks from inbound trains *and* assembling them into outbound ones. YDM, therefore, is a much more general and flexible model than was YSSM, in the sense that it can model not just large classification terminals, but also the far more numerous flat yards and switching terminals of any rail network.

Model Formulation

Our formulation will contain the following data:

Data from MNSP* with appropriate interpretation

$\mathcal{I}$: set of Inbound Trains i

$\mathcal{O}$: set of Outbound Trains o

$\mathcal{C}$: set of cuts traveling through yard

$\mathcal{C}(i)$: set of cuts arriving on Inbound Train i

$\mathcal{C}(o)$: set of cuts departing on Outbound Train o

s_{io}: needed buffer time between completion of disassembly of inbound train i and start of assembly of outbound train o, where $\mathcal{C}(i) \cap \mathcal{C}(o) \neq \emptyset$.

This completes the formal definition of the four models which are the focus of the thesis. RNSP, LMPM, YSSM, and YDM.

1.4 Rail Operations Literature Review

While there has been little work in the O.R. literature addressing railroad scheduling problems at the highest, network level, there is a great deal of work on various important components of railroad operating plans, as well as of scheduling problems in general. An excellent but dated overview of research in rail operations may be found in Assad's classic 1980 papers [5, 6]. Although the reader will find no references there to work from the last two decades, these papers do provide an overall taxonomy of rail operational problems and modeling approaches for them which is still useful and relevant today.

1.4.1 Freight Routing, Network Service Design, Blocking

The principal operating decisions to be made by a railroad, before any scheduling questions can be addressed, involve the strategic questions of how to route freight from origin to destination, on what lanes to run trains, and with what frequency. Such problems fall under the rubric of what is known in the railroad industry as *service design*, which can perhaps best be thought of as a type of network-loading problem. Intimately related to this service design problem is the question of how blocking is to be done at each yard, which specifies how, when, and where to break up and re-sort trains. For example, a railroad might decide that all its Chicago-Jacksonville freight will be routed through Knoxville and Atlanta. This is specified by the *freight routing plan* (in the language of Section 1.2). Then, the *blocking plan* might specify that such freight will be re-sorted only once, at Knoxville, and pass through Atlanta with minimal handling. To create a train, several blocks from Chicago to Atlanta must be combined to create a *planning train*, which becomes part of the *train-plan* (or *trip plan*). Finally, only once actual identities are given to each planning train, we are ready to begin making the scheduling decisions which are the subject of this thesis.

Much of the most notable work in rail operations addresses these tactical decisions which precede scheduling. Bodin, *et. al.* [10] propose a nonlinear, mixed integer programming model of the blocking and service design problems based on steady state O-D traffic demand data. Their model is quite detailed yet required manual interaction to solve a single test instance, derived from data from the Norfolk and Western Railroad. Namely, they fixed the blocking decisions (equivalent to solving the fixed-charge part of a network-loading problem), solved for the remaining flow variables, then manually guided a short branching procedure to find local improvements to this plan. While such

a solution methodology is obviously not applicable in general, the paper is notable as the first to introduce the notion of the railroad *service network*, which is the explicit formulation of the blocking problem as a network-loading problem with arcs representing blocking decisions.

Such a network model is utilized by Crainic, *et. al.* [25], who proposes a *path-based* formulation of the loading problem on the service network, coupled with an intriguing nonlinear objective function. The objective function is derived from analytical expressions of steady-state car delays in classification yards as a function of traffic volumes and train departure frequencies (see [77]). Their solution procedure is similar in spirit to Bodin *et. al.*, as is every other model of this section. Combinatorial decisions are fixed heuristically; the remaining nonlinear programming problem is decomposed and solved via Lagrange relaxation, and then integral variables are adjusted for local improvements. Unlike Bodin *et. al.* however, Crainic *et. al.* propose a generic heuristic for the local improvement which is based on calculating a statistic analogous to a "reduced cost" for each integer variable and branching only on the handful of variables with the largest such statistic. Crainic, *et. al.* [26] expand on [25] by adding a more formal column generation procedure to their solution. More recently, Newton [59] added polyhedral enumeration via cutting planes to a path-based formulation similar to [25, 26] to accelerate the solution even more. He also utilized a simpler, linear objective function which does not consider the fixed-charge cost of creating new trains, and reports favorable computational results with data derived from CSX Transportation, Inc.

Keaton also concludes that the path-based formulations of Crainic, *et. al.* lead to superior solutions. Although his initial paper, [45], proposes an arc-based formulation of the service design problem which minimizes the fixed costs of train creations, this formulation is abandoned in [46] in favor of a path-based one. He keeps the same linear objective function of [45], however, and in both papers utilizes Lagrange relaxation to decompose the MIP into a network of simpler problems which are coordinated with a subgradient-based dual price adjustment method. Finally, Haghani [37] formulates a rather comprehensive time-space service design model which also captures the flow of locomotives and empty cars throughout the network over a 7-15 day planning horizon. He proposes a heuristic solution procedure, although reports computational results on only one problem instance.

In short, there has been a great deal of progress in the solution of the tactical decision problems which precede scheduling. In particular, several research teams have noted that path-based formulations of the rail service design problem utilizing column-generation-like enumeration schemes

coupled with one or more other solution technique (Newton's cutting planes, Crainic's branching heuristics, or Keaton's Lagrange decomposition) have worked successfully for problems of realistic complexity. With this in mind, we feel that the imperative to create useful scheduling tools to complement optimization-based methods for service design is stronger then ever. Ultimately, we might even envision the coupling of the scheduling and service design decisions. Computer generated blocking plans could be used to create a schedule which may then suggest updates to the blocking plan, and so on. We therefore proceed to discuss the state of the art in the two node-level scheduling problems which occupy this thesis, linehaul meet-pass planning and yard switching and sequencing. These models are candidates to become the CAD planners mentioned earlier.

1.4.2 Linehaul Decision Modeling

Models of linehaul decisions fall into two broad classes. Combinatorial models of meet-pass planning involve explicit formulations of all the meet and pass decisions that must be made to schedule trains across a line, and are typically solved by some form of partial enumeration of the decision tree (such as branch and bound). There are also analytical delay models which attempt to predict optimal or "realistic" linehaul traversal times for trains as a function of the traffic profile under a given scenario.

In principle, these delay models are all that one requires when solving network-level scheduling problems. However, there are two considerations which might motivate a preference for the more detailed, enumerative models. The first, of course, is the perhaps limited accuracy of an analytical model, which must after all approximate a complex, discontinuous function of train start-times. Moreover, since these functions typically estimate expected delays, they have the effect of "averaging" the delays in any conflict among all the lowest priority trains involved, while a network scheduler is really interested in the actual delays of each specific train.

We will focus our discussion, therefore, on enumerative models of linehaul planning. The first notable work is by Sauder and Westerman [70], who describe initial success with an enumeration scheme to find meet-pass plans at a Norfolk Southern line. Many of the details of their tree search procedure are unfortunately left out of the discussion. Almost a decade later, Jovanovic and Harker [43] present a more comprehensive discussion of enumeration procedures for meet-pass planning. Their first contribution is the expression of linehaul topology as a series of mainline track segments (single, double, or triple track) connected by segments with sidings. These they generically call *meetpoints* and model as zero-dimensional entities where trains are permitted to meet, pass, and overtake each

other. This conceptualization of linehaul topology is shared by [57, 20], as well as the IP formulation of LMPM presented in Chapter 3. A second contribution is that they articulate the case for complete scheduling of linehaul traffic, and identify it as a modeling assumption even though it is not consistent with current operating practice in most railroads. In fact, a large amount of linehaul traffic comprises local and unit trains which are not scheduled. Nevertheless, their position that railroads should and will move toward more comprehensively scheduled traffic is shared by this thesis and motivates what we do. Their solution technique, however, is similar in spirit to [70]. They present an enumeration tree encoding the meet-pass logic and apply a guided depth-first search to find a feasible schedule, as well as a fathoming scheme based on running trains at full speed to their destinations after a meet.

They do not attempt to optimize meet-pass decisions, only to find a plan which gets each train to its destination on time, or to show that no such plan exists. In this sense, they call their model a schedule "analyzer" for network schedules, and report impressive computation times on sample schedules from the Burlington Northern railroad. This performance suggests that these sample schedules were either "highly" feasible or "highly" infeasible, so that a meet-pass plan is generated, or the entire tree fathomed, without much exploration of the decision space. In any case, the authors suggest that their model be used as a tool to evaluate the kinds of network scheduling decisions addressed in this thesis.

Kraay [47] does something like that by considering a network of linehaul dispatcher regions connected by yards and junctions, where lines are modeled as in [43] and yards simply as delay generators. Moreover, rather than merely assess the feasibility of a sample schedule, he attempts to generate meet-pass plans that minimize weighted tardiness costs for each train. His work has two notable features. First, it utilizes a decomposition of the resulting mixed integer program into integer and continuous variables: integer variables are fixed, the continuous-variable subproblem is solved, and a local search is utilized to update the integer decisions. This is the same scheme utilized by all the service design models mentioned above. The second notable feature is that a database of historic problem instances (with corresponding solutions) is maintained as part of the model. Each new instance is compared with those in the database to find a "closest instance," and the meet-pass plan from this closest instance is utilized to generate an initial solution to the new instance. In principle, as the size of the historic database increases, or the quality of the solutions in the database improves, the ability of the model to solve meet-pass planning problems

similarly advances. To our knowledge, it is the only model of rail operations capable of "learning" as its solution experience increases. Learning techniques will be important in the more general approximate dynamic programming models discussed in Section 1.5. In any case, Kraay presents experimental results on realistically sized line networks where he reports encouraging computational times but offers no estimation of the quality of reported schedules.

Two other works involving heuristic solution to combinatorial linehaul meet-pass problems are by Cai and Goh [20] and by Mees [57]. Both papers formulate mathematical models of linehaul scheduling problems with a representation of line topology as in [43] and Section 1.3.2, then define detailed heuristics for their solution. Cai and Goh describe a depth-first branching strategy very similar to [43] to find meet-pass plans, and Mees formulates the same problem as a time-space multicommodity network flow problem with side constraints. These constraints (as well as the multicommodity capacity constraints) are relaxed, leaving a set of shortest path problems, and Lagrange prices are updated heuristically. Unfortunately, neither paper presents any computational experimentation.

In short, there is to date in the literature not a single implementation of a combinatorial model of linehaul meet-pass planning which is solved to provable optimality. In this respect, we feel that the IP models of Chapter 3 represent an important contribution.

The remaining literature on linehaul modeling makes up the second class of linehaul models and involves analytic and probabilistic estimation of linehaul traversal times. Following the initial work of Peterson in the 1970's [61, 62], Chen and Harker [23] formulate a model of delay on a single-track line, and Harker and Hong [39] expand this model to include sections of double track. Both papers rely on the calculation of fixed points of detailed nonlinear equations involving the mean and variance of traversal times. Greenberg [36] approximates traffic on a single track line as an M/D/1 queue to generate the same moments, but her approximation is appropriate only under conditions of low traffic congestion. Moreover, just as Kraay extended [43] to a network of interconnected lines, Hallowell [41] does the same for [23, 39]. Unlike Kraay, however, she considers the problem of evaluating the feasibility of schedules in an expected value sense, where meet-pass decisions are made probabilistically, and does not worry about the actual specification of such decisions. Her only decision parameter is a global "temperature" like term which determines the probability that a lower priority train should take precedence over a higher priority one at a meet or overtake. She optimizes this term using an iterative procedure, which she calls the *Line Scheduling Algorithm*.

1.4.3 Yard Switching and Sequencing

There is a considerably smaller literature in the optimal scheduling of yard operations. Moreover, this literature is susceptible to two generic criticisms. First, it is, with the exception of one work [80], entirely heuristic in nature and utilizes no formal optimization methodology. Second, and again with the exception of one pair of related papers [34, 4], we shall see that the most popular objective functions are appropriate for steady-state analysis of control policies but not necessarily for a particular planning scenario over a specific time horizon.

To our knowledge, the only sequencing model which utilizes an explicit optimization methodology is due to Yagar, *et. al.* [80]. They formulate a sequencing model for the hump as a dynamic program, and then heuristically determine the assembly sequence as a function of the optimal hump sequence. They present computational results from two realistic scheduling instances. However, their objective is to maximize the general throughput of the yard by minimizing the time till the last inbound car is pulled from the bowl, rather than any function of delay for departing trains. Their model is therefore a generalized single-machine min-makespan problem. Moreover, while maximizing throughput is certainly a sensible objective over steady-state policies, it is not clear that it is appropriate within any given scheduling scenario, especially under the assumption that all outbound trains have individual target departure times and priority weights.

Armacost [4], following the work of Ferguson [34], considers the classification terminal by focusing on its two principal processes, disassembly and assembly. All other tasks (inspections, etc.) are set around that sequencing. Their work extends the work of Yagar, *et. al.* in two important ways. First, sequencing decisions are made at both the hump and at assembly, taking the one-machine model of Yagar, *et. al.* and extending it to two "machines." Second, it explicitly incorporates target departure times for outbound trains, making it much more relevant for scheduled rail operations. However, unlike Yagar, *et. al.*, they do not optimize these sequencing decisions, but rather utilize heuristic sequencing rules which correspond to our Min Lateness heuristic (see Section 4.2). Moreover, unlike in YSSM, outbound train priorities and inbound train arrival times are not considered, and fixed, complete *a priori* sequencing of cuts within outbound blocks is assumed. On the other hand, their work does utilize a more realistic model of the humping process than YSSM. Namely, in YSSM, constraints on the assembly of outbound trains specify that a block may not be pulled from the classification yard until all inbound trains carrying cars to that block have been humped. The Ferguson/Armacost model considers the position of the last car in an outbound block within an

inbound train. If this car is at the front of its inbound train, for example, then the block to which it is destined can in principle be pulled from the bowl much sooner than the completion of disassembly for the entire inbound train.

One other notable body of work in the modeling of classification yard operations is due to Daganzo [29, 30], who analyzes the physical space requirements of general sorting strategies within classification yards. Daganzo, *et. al.* perform a similar analysis of average yard throughput [31]. These papers do not formulate models for scheduling. Rather, they simply describe and analyze the steady-state properties of various control policies. In this sense, they are also not appropriate for scenario-specific yard control.

1.5 Scheduling and ADP Literature Review

We feel that the contributions of this thesis have potential implications beyond railroad operations alone, due to the many other applications of MNSP and MNSP*. Moreover, the work of Chapters 4 and 4 represents one of the first applications of approximate dynamic programming techniques to the direct solution of a problem with as broad economic importance as MNSP.

We therefore now review the literature surrounding the job shop scheduling problem, which is the most well known special case of MNSP*. We follow that by a discussion of one use of approximate dynamic programming to solve the job shop problem, which then motivates a brief introduction to some of the "state of the art" in ADP solutions for various control problems.

1.5.1 Job Shop Scheduling

The *job shop* scheduling problem is one of the most notoriously challenging computational problems in operations research and operations management. Almost all the literature around this problem deals specifically with the *min-makespan* objective, rather than the *min-tardiness-cost* objective of MNSP, but under either objective function, it a strongly $\mathcal{NP}$-hard problem which is also empirically difficult in practice. A classic but otherwise un-extraordinary single instance of this problem, involving only 10 jobs and 10 processors, was published in an industrial engineering textbook in 1963 [55] and remained unconquered for over two decades until finally solved to provable optimality by Carlier and Pinson [21].

Their approach was to utilize a branch and bound scheme, where branching was done on pairs

of jobs (f_a, f_b) at a competing processor. Each branch specified that either f_a precedes f_b at the processor, or *vice versa*. Fathoming of nodes is done by calculating bounds utilizing *heads* and *tails* for jobs at processors, which are statistics similar to the *Previous Processing* and *Remaining Processing* time features utilized in this thesis (see Section 4.2). In [22], they refine the definitions of these statistics to improve the computational performance of their solution algorithm. To our knowledge, their work is the most empirically successful among the class of branch and bound algorithms. Brucker, *et. al.* [19] do report better computational performance with a more complicated branching scheme, but it is relevant only to min-makespan problems, while Carlier and Pinson's branching could in principle be modified for different objective functions (by using tails to estimate tardiness costs, for example).

In addition to branch and bound approaches to solving job shop problems, there has been a great deal of effort in describing the convex hull of integer solutions to the integer programming formulation of job shop, first introduced by Balas [7], with cutting planes. Unfortunately, these efforts on their own have yielded no effective solution algorithms. Applegate and Cook [3] describe the incorporation of cutting planes to assist a branch and bound procedure which generates reasonable performance, but is still inferior to [21, 22].

Overall, however, it seems that the most empirically successful solution to job shop scheduling, at least in the O.R. literature, remains a heuristic known as the *shifting bottleneck procedure* [2]. This is true even though heuristic solution approaches represent a very small proportion of the published literature. Shifting bottleneck can be thought of as an exterior-point local search procedure, and works by starting with an infeasible schedule generated by ignoring all processor capacity constraints. A processor is then chosen according to a heuristic which attempts to identify the most "important" (or bottleneck) processor in the network. That processor is sequenced via the solution of a single-machine sequencing problem, and a new schedule is generated in which the derived sequence at the bottleneck processor is respected, but capacity constraints at all others are ignored. Although single-machine sequencing problems with release dates are themselves strongly $\mathcal{NP}$-hard [51], in practice they can be effectively solved [64]. The new schedule is used to identify a new bottleneck processor among the remaining un-sequenced processors, and the method repeats, sequencing each processor in the network one at a time. Occasionally, a kind of backtracking is performed, where previously sequenced processors are re-sequenced. The algorithm concludes with a feasible schedule.

Shifting bottleneck is a special case of what is known in the artificial intelligence community as

an *iterative repair* heuristic. In such heuristics, one starts with an infeasible schedule and iteratively makes local "repairs" in order to find a feasible schedule of desirable quality. In shifting bottleneck, the repairs correspond to the sequencing (or re-sequencing) of individual processors. Zweben, *et. al.* [83] describe a simpler repair procedure where repairs are performed on the schedule by identifying classes of violated constraints (such as processor capacity constraints) and shifting the schedule of only one job at a time. An important issue here is how to choose which job to "shift" in any iteration (akin to choosing a step direction in local search). The authors address this question by using a *scoring* function to compare various (infeasible) schedules and assess the value of adjusting the schedules of any jobs. If a repair leads to a schedule with better value, it is accepted; otherwise, a new repair is tried. The authors also add to this scoring function a noise term whose magnitude decays exponentially in time, as in simulated annealing.

Therefore, while there has been a tremendous amount research in the O.R. community on the job shop scheduling problem, heuristic scheduling methods are perhaps insufficiently represented in that work, especially considering that they account for the most effective solutions to this problem. Moreover, we make again the point that almost all the literature surrounding this problem, at least in the O.R. community, involves the minimum-makespan objective function rather than minimizing tardiness costs, which is the objective function of concern to this thesis. We believe that this is due to historical accident, as it was such a problem that was published in Muth and Thomson [55] and subsequently captured the imagination of the research community, rather than to any greater utility or difficulty of one objective criterion over the other.

1.5.2 Approximate Dynamic Programming

Zhang and Dietterich [81, 82] tackle the same scheduling problem of Zweben, *et. al.* by utilizing iterative repair, but with a better scoring function. They use the temporal difference method TD(λ) of Sutton [71] (see Section 4.4) to train a neural network to make scoring decisions. In their first effort [81] they utilize a human-made *feature extraction mapping* (see Section 4.1.1) to assist the training of the neural network, and in [82] they utilize a more primitive set encoding of the schedule and essentially allow their neural network to find its own features itself. They find that this latter approach is modestly better than the former and generates schedules which are substantially superior than those of [83]. It is important to point out that their model really solves the problem of optimally controlling the iterative repair heuristic of Zweben, *et. al.* rather than actually controlling a job

shop itself.

In any case, their work suggests the efficacy of approximate dynamic programming (ADP, see Section 4.1) methods to solve combinatorial optimization problems. Although ADP is by no means new, it has enjoyed an increased interest lately due to a few recent, exciting successes. The most celebrated involve the work of Tesauro [72, 73, 74, 75] in creating a world-class backgammon player. His work, taken as a whole, represents an outstanding case-study of the challenges and opportunities offered by these methods, and illustrates many of the implementation issues which arise again and again in other applications, including those of Chapter 4.

He begins by supervised training of a neural network with a feature extraction mapping on examples of expert play, and creates what he calls *Neurogammon* [72], which won the backgammon championship at the 1989 Computer Olympiad. Of course, the performance of such a program is in principle "bounded" by the quality of the expert play on which it trains, and so he then explores the more compelling prize of the ADP community, which is to develop a program capable of "learning" from its own experience [74]. To do this, he utilized TD(λ) to train two neural networks. One maps a raw encoding of the board position to a cost-to-go estimation; the second utilizes a feature extraction mapping to map aggregated states to costs-to-go. Tesauro finds that by judicious use of features, he is able to significantly improve the performance of his new backgammon player, which he calls TD-gammon. Thus, while TD-gammon utilizing raw data played as well as Neurogammon, TD-gammon utilizing the same feature extraction mapping of Neurogammon significantly outperformed it and played very well against three world-class backgammon players. It is interesting to contrast his experience with human-generated features, which improved the performance of his ADP, with that of Zhang and Dietterich [81, 82], mentioned above, who did better without them. Another interesting observation made by Tesauro, which is reiterated by Zhang and Dietterich, as well as by the computational experimentation of Chapter 4, is that the quality of TD-gammon bore little seeming relationship with the quality of its cost-to-go predictions. Namely, while its predictions were typically around 10% from true costs-to-go, the difference between an optimal and suboptimal control decision was often on the order of 1% or less. This is an important issue which we address in Section 4.4.4. Finally, in [75], Tesauro and Galperin introduce the notion of heuristic rollouts (see Section 4.4.3), which are more fully explored and analyzed for the first time by Bertsekas, Tsitsiklis, and Wu in [14]. These policies will be extraordinarily important to us in our application, and will prove to be the most powerful ADP tool we explore.

Other important successes involving ADP methods include a feature-based Tetris player [76, 13] which performed several orders of magnitude better than a heuristic player (but only after increasing the dimension of the feature space from 2 to 21), a retailer inventory management controller which beats optimized threshold policies (the most common heuristic solution methods) by as much as 10% [78], and a controller for a bank of elevators [27].

On the whole, it is difficult to say much about this body of work, and about these methods in general. This is mostly because they are still very new, and computational experience with them is still limited. In general, there is little theoretical guidance, and implementations, even more than in nonlinear or combinatorial optimization, involve more art than science. Some themes do emerge of course, such as the importance of sufficiently rich features, exploration (first identified by [9] and discussed in Section 4.4), and the power of rollouts. Moreover, it is important to observe that with the exception of [81, 82], as well as the work described by this thesis, all these applications involve non-deterministic problem, which are believed to be more amenable to solution by ADP. It is also important to observe that in no application is a systematic comparison between the ADP and optimal controllers available. As for general observations, there is a great deal of contradictory experience. Some find neural networks useful, others not. Sometimes extending exploration improves performance by an order of magnitude, other times it has little effect. We shall encounter many of these issues again in Chapter 4.

1.5.3 Multiclass Queuing Networks and Rail Operations

This chapter has introduced several problems in rail operations and articulated the position that they can be considered special applications of a more fundamental problem, MNSP (or MNSP*), which is a central subject of this thesis.

In true rail operations, however, processing times at lines or terminals are never deterministic quantities. Nor are start times, nor inter-train precedence constraints. Unscheduled unit coal trains may emerge from a mine-head with little warning, and scheduled trains may break down, depart without a complete consist, or simply never be dispatched at all. Moreover, many railroads would perhaps have a difficult time generating the kind of linearly separable cost function of train arrival times which we take for granted as the key minimization objective of MNSP. While some shipments, such as intermodal trains, might have financial penalties attached to late arrivals, there are others, such as coal trains, which have tremendous schedule flexibility.

In the long run, what railroads truly need is not just a tool to help them get a specific set of shipments in some short-term planning horizon to their destinations. What they need is to be able to generate schedules which will be both robust to random disturbances and efficient in that they permit the transportation of the greatest value of freight with the smallest possible steady-state use of resources.

With this in mind, consider a network similar to MNSP which we'll call a Multiclass Queuing Network (MQN). It is a network of processors just as MNSP, except that jobs arrive subject to some probabilistic distribution and are ordinarily distinguished from each other only to the extent that they belong to one of some small number of job classes. We no longer, in other words, consider every single job in the network to be a unique commodity.

Each class has its own distinct priority level and cost profile, its own route through the network (which may be stochastic, or a control decision), and its own set of processing speeds, which are also possibly stochastic. Moreover, costs in the system are not functions of arrival times, but are paid as rents levied to each job class at each processor. The rates are a function of both the job class and the processor. For example, a job representing a shipment of hazardous materials might have an especially high charge in a terminal or line near a heavily populated area, encouraging MQN to either route it away or move it through with minimal delay.

Within this framework, the control space for MQN comprises some or all of the following:

1. continuous-time sequencing decisions to determine which job *classes* receive processing at different processors at each time,

2. dynamic re-routing of jobs where possible, and

3. real-time admission authority to accept or reject newly arriving jobs at the network.

In short, MQN is in many ways a more compelling model of rail operations than MNSP. Moreover, there is a tremendous literature surrounding these problems. Unfortunately, MQN is extremely hard. Papadimitriou and Tsitsiklis [63] show that at least one special case is actually EXP-complete. For this reason, a large majority of the research effort directed at this problem has been more *descriptive* than prescriptive. Namely, for most reasonable examples of MQN, estimating distributions or even moments of key performance measures such as queue lengths, weighted WIP, or total sojourn times is challenging enough to make the question of "optimizing" these statistics over control policies completely impractical.

Fortunately, however, there are some encouraging signs. Progress toward generating control policies for special cases of MQN have actually come out of much of the prescriptive research mentioned above. Fluid approximations to MQN, for example, while originally developed to help analyze the stability of queuing networks, have been used with some success by Kumar and also by Ricard [49, 67] to motivate heuristic control policies. Similarly, conservation laws for some classes of MQN were first used to generate bounds on the feasible region of performance vectors in work such as that of Bertsimas, Paschalidis, and Tsitsiklis [17], who utilized potential functions to formulate a general expression of achievable performance regions. It was then extended by Niño-Mora [58] to identify simple but optimal heuristic control policies. Even diffusion approximations have been used to formulate effective control policies, although for small networks [38].

In short, although the "state of the art" of solution methods of MQN control problems is far behind that for other, more traditional optimization problems, important progress is being made. Moreover, another advantage of such formulations is that they naturally lead to a different emphasis in control objectives. Because controllers in stochastic problems such as MQN must by definition optimize over expectations of costs, they perhaps lead to policies which emphasize steady-state objectives such as long-term average queue length or long-term average delay, over transient objectives such as getting a specific tardy train to its destination on time.

Chapter 2

Decomposable Solution of RNSP with Node Planning Tools

2.1 Overview

Chapter 1 reviewed the recent advances in optimization based planning tools for the railroad industry, and introduced the notion of *computer aided dispatching* (CAD) tools. It is believed by many that such tools will play an increasingly important and useful role, certainly in the real-time control of linehaul train movements, and eventually in the control of train sequencing and car switching in flat yards and classification terminals. Nevertheless, the vision for such tools so far has remained largely focused on the real-time domain. In a hierarchical scheduling context, such tools naturally fit in the last stage of train control. After all strategic freight-routing and blocking decisions have been made, after tactical trip-plan and train scheduling decisions have been made, a CAD tool's role would be to help line-haul and yard dispatchers execute as closely as possible the operating plan.

In this chapter we would like to discuss the next step of such research. Rather than use a CAD tool only in a real-time setting to help control local train movement, can we use these same CAD tools in a tactical planning setting to actually derive the network schedule? Specifically, can we formulate RNSP in such a way that we can then decompose it into a set of subproblems, each of which is identical to the planning problem solved (or at least addressed) by these CAD tools?

To answer this question, we decompose MNSP into a set of decoupled, single-processor planning

problems, or *node problems*, and notice that these node problems correspond, in the rail domain, to the problems solved by our CAD tools. In the context of general MNSP, we will refer to such tools as *node planners*, or *node planning algorithms* (NPA). Each node planner will be input a set of arrival times for jobs (each job's arrival time at the processor), as well as an objective function, by a higher-level entity which we call the *coordinating algorithm*, or the *master planner*. This high-level entity will then examine the output of the node planners, update the arrival times, and repeat the entire process until some convergence criterion has been satisfied.

The key question in such a formulation, of course, is how to update arrival times and the objective functions which are passed to the node planners at every iteration. In this chapter, we present a heuristic update procedure which does not necessarily generate an optimal schedule, but empirically (at least for RNSP) appears to generate acceptable, feasible network schedules quickly.

2.2 Decomposition of MNSP

We begin by introducing some new notation, and then presenting the decomposition of MNSP, which leads naturally to a similar decomposition of RNSP.

2.2.1 Definition of Terms

Let:

$I(j)$: set of jobs f that pass through node j.

$E(j)$: set of jobs f that end their journey at node j.

$x_{f,j}$: time at which job f arrives at node j.

$\vec{x_j}$: a subvector of x, indicating arrival times of all jobs at node j.

$\vec{x_j^{nxt}}$: a subvector of x, indicating arrival times of each job at the *next* node in its path after node j. For example, if jobs 1 and 2 both traveled through node j, but then Job 1 proceeded to k and Job 2 to l, $\vec{x_j^{nxt}}$ would be $[x_{1,k}, x_{2,l}]^T$.

x: the entire vector of arrival times.

x_k: the k'th element of the vector x. In our notation, x_k is equivalent to $x_{f,j}$ for some job f and node j. In this chapter we will find it helpful to sometimes designate each element of x with just one index instead of two.

$\lambda, \vec{\lambda_j}, \vec{\lambda_j^{nxt}}, \lambda_k$: vectors of Lagrange multipliers defined as their analogues in x.

$y, \vec{y_j}, \vec{y_j^{nxt}}, y_k$: replicated arrival times defined just as x.

$\mathcal{P}$: the set of all feasible vectors x, i.e.

$$x:\ x_{f,\nu_f(j)} - x_{f,j} = T_{f,j}(x)\ \ \forall f,j,$$

where $T_{f,j}(\cdot)$ is the processing time function introduced in Section 1.3. In the context of RNSP, $\mathcal{P}$ is the set of vectors corresponding to a feasible meet-pass plan at every line, a feasible switching and sequencing plan at every terminal, etc.

$F(x)$: some convex (possibly linear), continuously differentiable cost function which is linearly separable in x,

$$F(x) = \sum_{f,j} F_{f,j}(x_{f,j}).$$

Given $\mathcal{P}$, a logical statement for our problem would then be:

$$\min_{x \in \mathcal{P}} F(x).$$

2.2.2 Problem Decomposition

Consider the set $\mathcal{P}$ in the case of the RNSP. Here, for any particular meet-pass plan or yard switching and sequencing plan, and under fairly general modeling assumptions, $\mathcal{P}$ defines a polyhedron in x. In fact, $\mathcal{P}$ is most often specified by a (linear or nonlinear) mixed integer programming formulation of our scheduling problem (see [43] for one example for a line meet-pass problem; see also [8, 7] for a general discussion of the structure of a deterministic scheduling "polytope."), so $\mathcal{P}$ itself would be the union of all the polyhedra for all feasible sequencing plans at each node.

For example, imagine the scenario of Figure 2-1. The space of feasible arrival times is given by

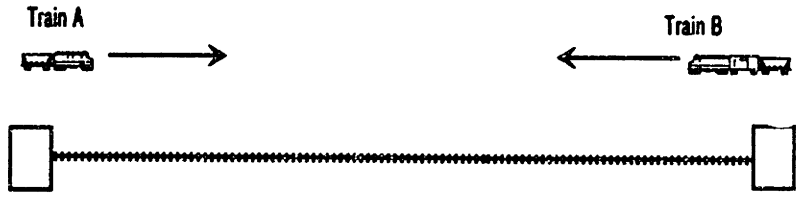


Figure 2-1: We have one line with no meetpoints, which takes one time-period to traverse. There are two trains at either end. Both are ready to start moving at time zero.

Figure 2-2. Queyranne [66] and Balas [7] each imagine similar pictures when thinking about more general scheduling problems.

To see how we might decouple separate scheduling polyhedra in a job shop, rail network, or other application of MNSP, consider the scenario of Figure 2-3. In Figure 2-3, a single train travels unimpeded over distinct dispatcher regions, where each region has borders delimited by the five

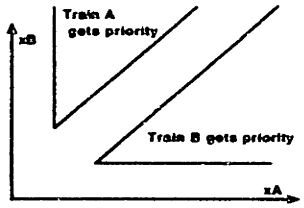


Figure 2-2: Decision space $\mathcal{P}$ for scenario of Figure 2-1. Two polyhedra represent the two possible dispatching decisions: allow Train A, or Train B, to proceed first.

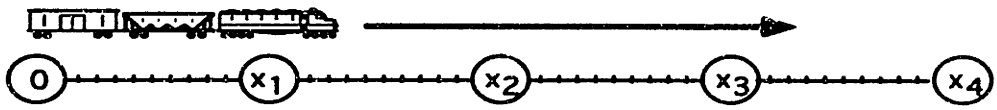


Figure 2-3: Illustrative Scheduling Scenario

reporting stations: 0, 1, 2, 3, and 4. If we assume the train starts at station 0 at time 0, and takes two hours (120 minutes) to traverse the track between each pair of reporting stations, then the set of feasible schedules $\mathcal{P}$ in this case is given by:

$$\mathcal{P} = \left\{ x \;\middle|\; \begin{array}{c} x_1 \geq 120 \\ x_2 \geq 120 + x_1 \\ x_3 \geq 120 + x_2 \\ x_4 \geq 120 + x_3 \end{array} \right\}$$

Now we define a variational set (a set whose definition varies with the values of some input vector) $\mathcal{P}(y)$ as follows:

$$\mathcal{P}(y) = \left\{ x \;\middle|\; \begin{array}{c} x_1 \geq 120 \\ x_2 \geq 120 + y_1 \\ x_3 \geq 120 + y_2 \\ x_4 \geq 120 + y_3 \end{array} \right\}$$

and we observe that

$$x \in \mathcal{P} \Longleftrightarrow x \in \mathcal{P}(x) \tag{2.1}$$

We also observe that for any fixed y, $\mathcal{P}(y)$ is the Cartesian product of four independent sets,

$$\begin{array}{llllll} \{ & x_1 & \geq & 120 & & \} \times \\ \{ & x_2 & \geq & 120 + & y_1 & \} \times \\ \{ & x_3 & \geq & 120 + & y_2 & \} \times \\ \{ & x_4 & \geq & 120 + & y_3 & \} \end{array}$$

and that each of these independent sets defines the set of decision variables for each of the four dispatchers in our scenario. We have therefore decomposed our problem into a set of independent dispatching problems, one for each planning node. Another way of saying this is that we can write

$$x \in \mathcal{P}$$

as either

$$x \in \mathcal{P}(x)$$

or

$$x_j^{nxt} \in \mathcal{P}_j(\vec{x_j}),$$

for suitably defined sets $\mathcal{P}_j(\cdot)$.

We would therefore like to do two things at once: we would like to formulate our problem above in a form which may not require an explicit characterization of $\mathcal{P}$. We would also like to be able to decompose this into separate problems for each node.

Let us define a duplicate vector y to be another vector of arrival times, precisely identical to x. As long as x and y are identical, we can freely interchange them. Doing so gives us the following equivalent restatement of our problem:

$$\begin{array}{c} \min_{x,y} \sum_{j \in V} \sum_{f \in E(j)} F_{f,j}(x_{f,j}) \\ \text{such that} \\ \vec{x_j}^{nxt} \in \mathcal{P}_j(\vec{y_j}), \quad \forall j \in V, \\ x = y. \end{array} \tag{2.2}$$

Now, let us incorporate the constraints $x = y$ into the objective function via a Lagrange relaxation,

which gives us:

$$\min_{x,y} \max_{\lambda} \left[\sum_{j \in V} \sum_{f \in E(j)} F_{f,j}(x_{f,j}) + \sum_{j \in V} \sum_{f \in T(j)} \lambda^{ij}(x_{f,j} - y_{f,j}) \right]$$

such that

$$\vec{x_j}^{nxt} \in \mathcal{P}_j(\vec{y_j}) \quad \forall j \in V.$$

Letting

$$\mathcal{L}(x, y, \lambda) = \sum_{j \in V} \sum_{f \in E(j)} F_{f,j}(x_{f,j}) + \sum_{j \in V} \sum_{f \in T(j)} \lambda^{ij}(x_{f,j} - y_{f,j}),$$

we have

$$\min_{(x,y):\vec{x_j}^{nxt} \in P_j(\vec{y_j})} \max_{\lambda} \mathcal{L}(x, y, \lambda) \equiv \min_{y} \min_{x:\vec{x_j}^{nxt} \in P_j(\vec{y_j})} \max_{\lambda} \mathcal{L}(x, y, \lambda).$$

Finally, we can write

$$\vec{x_j}^{nxt} \in \mathcal{P}_j(\vec{y_j}) \quad \forall j \in V.$$

in the shorthand of Equation 2.1 as $x \in \mathcal{P}(y)$, which gives us:

$$\min_{y} \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda).$$

We therefore see the outlines of a decomposition strategy for this problem, which will involve a master problem coordinating several, parallel subproblems.

2.2.3 Combinatorial Abstraction

We start by stating our first important assertion, whose purpose is to lead to a formulation that is decomposable by lines and will allow us to manage $\mathcal{P}$ abstractly.

Proposition 2.1 *At any $y \in \mathcal{P}$, the inner two optimizations $\min_{x:x \in P(y)}$ and $\max_{\lambda} \mathcal{L}(x, y, \lambda)$ satisfy Strong Duality, so that:*

$$\min_{x \in \mathcal{P}} F(x) = \min_{y} \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda) = \min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda).$$

Proof: First we prove the second statement, without resorting to the first. Define $v(P) = \min_{x \in \mathcal{P}} F(x)$ and note that

$$v(P) = \min_{y} \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda).$$

Since any feasible solution y^* to this problem is surely in $\mathcal{P}$, we can say

$$\begin{aligned} v(P) &= \min_{y \in \mathcal{P}} \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda) \\ &= \min_{y \in \mathcal{P}} \min_{x = y} \mathcal{L}(x, y, \lambda) \\ &= \min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathbb{R}^n} \mathcal{L}(x, y, \lambda) \\ &\leq \min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda). \end{aligned}$$

So we have $v(P) \leq \min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda)$. But by Weak Duality for nonlinear programs. for all y,

$$\begin{aligned} \max_{\lambda} \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda) &\leq \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda) \\ \min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda) &\leq \min_{y \in \mathcal{P}} \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda), \end{aligned}$$

so that

$$\min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda) \leq v(P).$$

We therefore have:

$$v(P) = \min_{y} \min_{x \in \mathcal{P}(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda) = \min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in \mathcal{P}} \mathcal{L}(x, y, \lambda).$$

The proof of Strong Duality of the inner problems appears above, and follows from the fact that if y is feasible ($y \in \mathcal{P}$), the set $\{x : x = y\} \subset \{x \in \mathcal{P}\}$, and,

$$\min_{x \in P_j(y)} \max_{\lambda} \mathcal{L}(x, y, \lambda) \equiv \min_{x = y} \mathcal{L}(x, y, \lambda) = \min_{x} \max_{\lambda} \mathcal{L}(x, y, \lambda) = \max_{\lambda} \min_{x} \mathcal{L}(x, y, \lambda)$$

by Strong Duality for convex programs with affine equality constraints. ∎

Proposition 2.1 gives us a new problem:

$$\min_{y \in \mathcal{P}} \max_{\lambda} \min_{x \in P(y)} \mathcal{L}(x, y, \lambda)$$

where the innermost problem expands to:

$$\min_{\vec{x_j}^{nxt} \in P_j(\vec{y_j})} \sum_{j \in V} \sum_{f \in E(j)} F_{f,j}(x_{f,j}) + \sum_{j \in V} \sum_{f \in T(j)} \lambda^{ij}(x_{f,j} - y_{f,j}) = \tag{2.3}$$

$$\sum_{j \in V} \left[\min_{\vec{x_j}^{nxt} \in P_j(\vec{y_j})} \left\{ \sum_{f \in E^{nxt}(j)} F_{f,j^{nxt}}(x_{f,j^{nxt}}) + \sum_{f \in T(j)} \lambda^{ij^{nxt}} x_{f,j^{nxt}} \right\} \right] - \tag{2.4}$$

$$\sum_{j \in V} \sum_{f \in T(j)} \lambda^{ij^{nxt}} y_{f,j^{nxt}}. \tag{2.5}$$

Let us define the set of functions NPA_j for each node j as follows:

$$NPA_j(\vec{y_j}, \lambda_{j^{nxt}}) = \min_{\vec{x_i}^{nxt} \in P_j(\vec{y_j})} \left\{ \sum_{f \in E^{nxt}(j)} F_{f,j^{nxt}}(x_{f,j^{nxt}}) + \sum_{f \in T(j)} \lambda^{ij^{nxt}} x_{f,j^{nxt}} \right\} \tag{2.6}$$

Notice that in going from Equation 2.3 to Equation 2.5, we have lost all variables $x_{f,j}, y_{f,j}$ and λ^{ij} that refer to the first node in any job's path. This is acceptable, since we can assume that these are fixed, that $x_{f,j} = y_{f,j}$, and hence that $\lambda^{ij} = 0$ at all these points. Otherwise, the two forms are equivalent.

The reader should recognize $NPA_j(\cdot)$ as the node planner mentioned Chapter 1. In the context of RNSP, it is a CAD tool. In the context of job shop scheduling, it might be a single-machine sequencing algorithm. Because we assume that we have access to such tools, at least for RNSP, we need not explicitly worry about solving $NPA(\cdot)$, nor about characterizing $\mathcal{P}$ in the innermost problem. Also also need not worry about what type of node j might be. The important thing is that $NPA_j(\cdot)$ takes a vector of times $\vec{y_j}$ (which are the arrival times at the node), and of prices $\vec{\lambda_{j^{nxt}}}$, and calculates an optimal vector $\vec{x_j}^{nxt}$, returning the optimal cost. Notice that the vector of all the NPA's (which we'll denote $NPA(y, \lambda)$) inputs the entire vectors y and λ and calculates the entire vector x, returning the realized cost $F(x) + \lambda'(x - y)$. This gives us, for a global problem:

$$\min_{y \in \mathcal{P}} \max_{\lambda} NPA(y, \lambda) - \lambda' y$$

$$\min_{y \in \mathcal{P}} \max_{\lambda} \sum_{j \in V} NPA_j(\vec{y_j}, \lambda_{j^{nxt}}) - \sum_{j \in V} \sum_{f \in T(j)} \lambda^j \vec{y_j} \tag{2.7}$$

$$= \min_{y \in \mathcal{P}} \sum_{j \in V} \left\{ \max_{\lambda} NPA_j(\vec{y_j}, \lambda_{j^{nxt}}) - \sum_{f \in T(j)} \lambda_j^{nxt} \vec{y_j}^{nxt} \right\}$$

From an implementation perspective, the key is that for every node, the outer "loop" (i.e. the outer two optimization problems) passes to the inner-most minimization problem nothing more than the arrival times for all jobs at that node (in the vector y), as well as an initial augmentation to the objective function for each sub-problem. The $NPA(\cdot)$ will then calculate the optimal arrival times at the "next" nodes for every job and write them to the vector x.

For now, the objective function passed by the master planner to $NPA(\cdot)$ is $F(x)$ with a linear augmentation term $\lambda' x$, which was motivated by our Lagrange decomposition. Later, we shall see that a more general augmentation term will be appropriate.

2.3 Normal Gradient Descent over y and λ

How shall we determine y and λ? We begin by introducing the following useful notation:

$$q(\lambda, y) = \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda) = \min_{x \in \mathcal{P}(y)} F(x) + \lambda'(x - y).$$

$$x^*(\lambda, y) = \arg \min_{x \in \mathcal{P}(y)} \mathcal{L}(x, y, \lambda)$$

Also, define $\nabla x^*(\lambda, y)$ to be the matrix

$$\{\nabla x^*(\lambda, y)\}_{f,j} = \frac{\partial x^*(\lambda, y)_f}{\partial y_j},$$

which may or may not exist at a particular (λ, y). We believe, however, that this gradient is defined almost everywhere. Finally, for all those points where $\nabla_y x^*(\lambda, y)$ exists, define:

$$\begin{aligned} \nabla_y q(\lambda, y) &= \nabla_y x^*(\lambda, y)' \nabla F(x^*) + \nabla_y x^*(\lambda, y)' \lambda - \lambda \\ &= \nabla_y x^*(\lambda, y)' (\nabla F(x^*) + \lambda) - \lambda. \end{aligned}$$

What we want of course is that $q(\lambda, y)$ be convex in y, and that when $\nabla_y q(\cdot)$ is not defined, we have available some simple procedure for calculating a subgradient. Then we might try relaxing the constraint $y \in \mathcal{P}$, at least temporarily, and using following iterative procedure for the high-level optimization over y and λ:

$$y^{k+1} := y^k - \alpha_k \left\{ \left(\nabla F(x^*(\lambda^k, y^k) + \lambda^k \right)' \nabla_y x^*(\lambda^k, y^k) - \lambda^k \right\}, \tag{2.8}$$

$$\lambda^{k+1} := \lambda^k + \beta_k \left\{ x^*(\lambda^k, y^k) - y^k \right\}, \tag{2.9}$$

where α_k and β_k are stepsizes satisfying:

$$\begin{array}{rclrcl} \sum_k \alpha_k & = & \infty & \sum_k \beta_k & = & \infty \\ \sum_k \alpha_k^2 & < & \infty & \sum_k \beta_k^2 & < & \infty \end{array}$$

We know that $q(\cdot)$ is concave in λ, so if it is strictly convex in y, this procedure will converge nicely for proper choices of α and β. Unfortunately, consider the example illustrated by Figure 2-4. We assume each line takes a single time period to traverse, that the train starts at Line 1 at Time

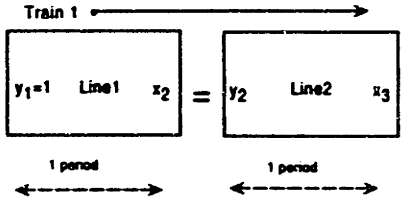


Figure 2-4: One Train, Two Line Network

1 (so y_1 is fixed at 1), and that the objective is to minimize the cost of arrival, which we say for simplicity is cx_3. The function we are maxi-minimizing is therefore:

$$\begin{aligned} q(\lambda, y) &= cx_3^*(y) + \lambda(x_2^*(y) - y_2) \\ &= c(y_2 + 1) + \lambda(2 - y_2). \end{aligned}$$

Let us examine the y and λ update rules in more detail:

$$\begin{array}{rcrcl} y_2 & := & y_2 & - & \alpha(c - \lambda) \\ \lambda & := & \lambda & + & \beta(2 - y_2), \end{array}$$

or, equivalently,

$$\begin{pmatrix} y \\ \lambda \end{pmatrix} = \begin{pmatrix} 1 & \alpha \\ -\beta & 1 \end{pmatrix} \begin{pmatrix} y \\ \lambda \end{pmatrix} + \begin{pmatrix} -\alpha c \\ 2\beta \end{pmatrix},$$

which is an update rule of the form $z^k = Az^{k-1} + b$. Expanding this, we see that

$$z^k = A^k z^0 + \left(\sum_{i=0}^{k-1} A^i \right) b,$$

so the sequence z^k is convergent if and only if the spectral radius (magnitude of the largest eigenvalue) of A is strictly less than one. In our case above, the magnitude of both eigenvalues is $\sqrt{1+\alpha\beta}$, so when both stepsizes are positive, our method must diverge.

We also had poor computational performance with more sophisticated approaches, such as *augmented Lagrangian techniques* [11], which attempt to "convexify" the Lagrangian by adding the term

$$\frac{1}{2}(x^* - y)' M (x^* - y),$$

where M is some diagonal matrix of strictly positive terms. Although these techniques were not divergent, it was still very difficult to control their convergence. We therefore suggest an alternative approach.

2.4 Method of Successive Approximations

We introduce a new approach to setting y, and temporarily put aside the issue of proper λ.

2.4.1 Feasible Schedule as Fixed Point

We recall from Section 2.2.2 that we stated that our combinatorial constraint set $\mathcal{P}$ was such that $y \in \mathcal{P}(y) \iff y \in \mathcal{P}$. Therefore, rather than focus on finding an optimal schedule y, let us rather think about finding a reasonable feasible schedule as quickly as possible. Define

$$\mathcal{X}^*(y,\lambda) = \left\{ x^* \;\middle|\; \begin{array}{c} F(x^*) + \lambda'(x^* - y) = \\ \min_{x \in \mathcal{P}(y)} F(x) + \lambda(x - y) \end{array} \right\}$$

and recall that if, for some λ, $y \in \mathcal{X}^*(y,\lambda)$, then by definition $y \in \mathcal{P}(y)$ and so y is a feasible schedule. Moreover, since $x^*(y,\lambda) : (y,\lambda) \mapsto \mathcal{X}^*(y,\lambda)$, we know that if $x^*(y,\lambda)$ should ever return y, we have found a feasible schedule. Our new approach is therefore to temporarily forego updating λ, leaving it fixed at some $\hat{\lambda}$, and attempt to find a fixed point of $x^*(y,\hat{\lambda})$.

There are many different methods for finding fixed points of functions. One simple averaging technique known by some as the *method of successive approximations* (see [68, 18, 60]), is as follows:

Method of Successive Approximations (MSA)

1. Start with some schedule y^0.
2. At every stage k:
$$y^{k+1} = \gamma x^*(y^k,\hat{\lambda}) + (1-\gamma)y^k$$
where $\gamma \in (0,1]$ is a relaxation factor.
3. Repeat until $y^{k+1} = y^k$.

There are several things we can say about this method. The first is that it is known to be convergent when x^* is a *contraction mapping* with respect to some norm $||\cdot||$, which means that there exists a $\beta < 1$ such that

$$||x^*(y^*,\hat{\lambda}) - x^*(y^\dagger,\hat{\lambda})|| \leq \beta||y^* - y^\dagger||$$

for any two schedules y^* and $y^\dagger$. Unfortunately, as $x^*(y^\dagger,\hat{\lambda})$ is typically not even continuous, it is certainly not a contraction, at least for common norms such as $||\cdot||_2$ or $||\cdot||_\infty$.

It is therefore not clear whether we can count on convergence of the method under general conditions. What about under special conditions? Consider the pathological case were the processing time of each job at a node, $T_{f,j}$, is independent of traffic conditions. In this case, the reader should notice that the optimal network solution is trivially calculated by solving

$$\begin{aligned} x_{f,P(f,1)} &= r_f \\ x_{f,j} &= x_{f,\pi_f(j)} + T_{f,\pi_f(j)}. \end{aligned}$$

In this case, will MSA converge? Fortunately, it will. To see this more clearly, we write $x^*(\cdot)$ as:

$$x^*(y,\hat{\lambda}) = Ry + T(y) \tag{2.10}$$

where $T_k(y)$, with element k corresponding to job f and node j, returns the traversal time $T_{f,\pi_f}(j)$ under schedule y. If node j is the first node in f's itinerary, then $T_k(y)$ returns zero. The matrix R in Equation 2.10 is the "back-shift" operator defined by setting $R_{kl} = 1$ if

1. for element k, the corresponding job, node pair is f, j, and for element l the job, node pair is f, $\pi_f(j)$, or
2. element k corresponds to a job,node pair f, $P(f,1)$,

and setting R_{kl} to zero otherwise. As an illustration, consider the case of a single job traversing four nodes, as in Figure 2-3. The back-shift matrix R takes the form:

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In this notation, MSA works as follows. Start with an initial schedule y^0 and perform the following iterations:

$$\begin{aligned} y^1 &= Ry^0 + T(y^0) \\ y^2 &= Ry^1 + T(y^1) \\ &= R^2y^0 + RT(y^1) + T(y^2) \\ y^3 &= Ry^2 + T(y^2) \\ &= R^3y^0 + R^2T(y^0) + RT(y^1) + T(y^2) \\ &\vdots \\ y^{N-2} &= \bar{R}y^0 + R^{N-3}T(y^0) + R^{N-4}T(y^1) + \cdots + \\ &\quad R^2T(y^{N-5}) + RT(y^{N-4}) + T(y^{N-3}) \end{aligned}$$

The reader should notice that R^2 is a "double-back-shift" operator, and $R^{N-2} = R^{N-1} = R^N = \bar{R}$ is the matrix which shifts every row, corresponding to job, node pair f, j, to the column

train	*start time*	*cost coeff*	*target arrival*
1	0	10	2
2	0	9	1
3	1	11	2

Table 2.1: Two Yard Scenario Data

corresponding to the first node in job f's itinerary. Therefore, $\bar{R}T(y) = 0$, so

$$y^{N-1} = \bar{R}y^0 + R^{N-2}T(y^0) + R^{N-3}T(y^1) + \cdots + R^2T(y^{N-4}) + RT(y^{N-3}) + T(y^{N-2}) \quad (2.11)$$

$$\vdots$$

$$y^{N+m} = \bar{R}y^0 + R^{N+m-1}T(y^0) + R^{N+m-2}T(y^1) + \cdots + R^2T(y^{N+m-3}) + RT(y^{N+m-2}) + T(y^{N+m-1}) \quad (2.12)$$

$$(2.13)$$

Clearly, if $T(y) = T$, then $y^{N-1} = y^{N-2}$ and MSA converges in $N-2$ iterations. ∎

2.4.2 Importance of Objective Function

The actual fixed point to which MSA converges, of course, depends on the choice of $\hat{\lambda}$. Consider, for example, the scenario of Figure 2-5. There are three trains traveling through a network of three

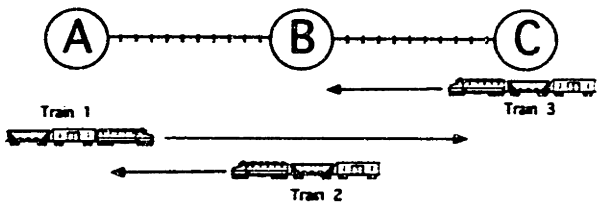


Figure 2-5: Price Critical Planning Scenario

reporting stations (A, B, and C) delimiting two dispatcher regions, which are the track separating A and B, and the track separating B and C. We assume there are no sidings on any of this track, except at the reporting stations A, B, and C. Data for this scenario are given by Table 2.1. Train 1 travels from A to C, Train 2 from B to A, and Train 3 from C to B. We assume each dispatching region takes one hour to cross, and our global objective function is to minimize this weighted sum

train	*Station A*	*Station B*	*Station C*
1	0	1	3
2	2	0	—
3	—	2	1

Table 2.2: Single Fixed Point when $\hat{\lambda_{1B}} = 10$.

of train tardiness:

$$10[x_{1C} - 2]^+ + 9[x_{2A} - 1]^+ + 11[x_{3B} - 2]^+$$

where the operator $[\cdot]^+$ returns the positive part of its operand. In this scenario, there are two subproblems, and only one interesting y variable, y_{1B}, which is an input to the planning problem corresponding to dispatcher region B-C. There is therefore also only one interesting dual price λ, λ_{1B}. We need to fix this price to some value, and since the final price for Train 1 is 10, this could be a reasonable choice for $\hat{\lambda}_{1B}$. The reader should verify, however, that with this choice of $\hat{\lambda}$, there is only one fixed point schedule y, which is given in Table 2.2. In this schedule, Train 1 takes priority over Train 2 in region A-B (because its implied price, $\hat{\lambda}_{1B} = 10$, is greater than the price of Train 2), and then waits at B for Train 3, arriving at its final destination at time 3. The cost of delay under this fixed-point schedule is $10(3-2)^+ + 9(2-1)^+ + 11(2-2)^+ = 19$, while an optimal schedule would have given Train 2 priority over Train 1. Train 1 would still arrive at time 3; Train 3 still at time 2, but Train 2 would have arrived exactly on time, giving a cost of 10, an almost 50% difference. This optimal schedule, incidentally, would have been the unique fixed point had we set $\hat{\lambda}_{1B}$ to anything less than 9.

It is therefore clear, perhaps not surprisingly, that our choice of $\hat{\lambda}$ can matter a great deal. By appropriately setting $\hat{\lambda}$, can we expect to find the optimal schedule y? One logical choice for an "appropriate" $\hat{\lambda}$ are the optimal dual prices from the Lagrange decomposition of our original formulation

$$\min_{y \in \mathcal{P}} \max_{\lambda} \{ \min_{x \in \mathcal{P}(y)} F(x) + \lambda'(x - y) \}$$

Namely, we are tempted to say:

Conjecture 1 *For any pair (y^*, λ^*) that solves Master Problem, λ^* is such that*

$$\hat{y} \in \mathcal{X}^*(\hat{y}, \lambda^*) \implies \hat{y} \text{ optimal}$$

Unfortunately, conjecture 1 is false. Consider the scenario of Figure 2-6. A single train travels across two rail segments, each taking 100 minutes to traverse, and we wish to minimize arrival time at Station 2. Clearly, $y^* = (100, 200), \lambda^* = 0$ is one possible solution to Master Problem. However,

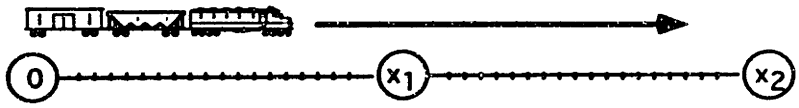


Figure 2-6: Line Planning Scenario

consider $\hat{y} = (1000, 1100)$.

$$\mathcal{X}(\hat{y}, \lambda^*) = \{y \mid y_2 = 1100\} \ni \hat{y}$$

but $\hat{y}$ is not optimal, contradicting Conjecture 1.

2.4.3 Target Times

What we are looking for is a set of "magic prices" which will drive MSA to the optimal schedule. Unfortunately, there is no clear reason why such prices should exist. Perhaps, however, we can generalize the notion of Lagrange dual prices. Namely, define $\widetilde{\mathcal{X}}^*(y, c)$ to be a more general version of $\mathcal{X}^*(y, \lambda)$:

$$\widetilde{\mathcal{X}}^*(y, c) = \left\{ x^* \;\middle|\; \begin{array}{c} F(x^*) + c(x, y) = \\ \inf_{x \in \mathcal{P}(y)} F(x) + c(x, y) \end{array} \right\}$$

where $c(x, y)$ is an arbitrary function of x and y. If we require $c(\cdot)$ to be linear in x and y, then this corresponds to our previous case, which derives from a Lagrange dual of our planning problem. Where $c(\cdot)$ is permitted to take more general forms, we can think of $\widetilde{\mathcal{X}}^*(y, c)$ as being derived from a more generalized dual. As we would expect, there are many functions $c(\cdot)$ such that $y \in \widetilde{\mathcal{X}}^*(y, c) \iff$ y minimizes $F(x)$ over $\mathcal{P}$. In this thesis, however, we shall explore functions of the form:

$$c(x, y) = \sum_i \lambda_i [x_i - x_i^{\text{target}}]^+ \tag{2.14}$$

We choose functions of this form for several reasons. First, these are the standard lateness and tardiness penalty functions which a railroad's CAD node planning tool might be expected to optimize. Secondly, because of the greater degree of freedom (we now get to choose target times

x^{target}, as well as dual prices λ), we expect these dual augmentation functions to be more powerful than linear functions.

We also speculate that the most important ingredients to these functions will be the target times. In fact, as a degenerate case, if we could identify the optimal schedule y^* and set $x^{\text{target}} = y^*$, we could simply pass to $x^*(\cdot)$ our function $c(\cdot)$ as an objective function and be sure that $y^* \in \widetilde{\mathcal{X}}^*(y^*, c)$. Of course, we cannot know y^* in advance, but can we make a good guess? If so, we could then rely on MSA to find a feasible schedule which might be close to y^*. For this reason, the rest of this thesis will describe our two approaches for finding good target times for $x^*(\cdot)$, which are described in Chapters 3, and 4. First, however, we mention preliminary computational experiments which demonstrate both the power of MSA to find feasible schedules to realistic planning problems, as well as the importance of starting with an appropriate set of target times.

2.5 Computational Experiments

We created several experimental scenarios involving yards and lines. The yards were developed from data reflecting Conrail's Selkirk yard and are described in more detail in [4]. The lines were modeled after a 50 mile section of Burlington Northern track between the Pasco and Yardley stations in Washington state. Each line had four sidings. Each scenario involved 60 different trains, 4 lines, and 5 yards, with varied train start times to model different levels of congestion. In Chapter 3, we will see that schedule congestion is a key ingredient in the final "difficulty" of a scheduling scenario.

Table 2.3 demonstrates the effectiveness of MSA in coordinating a network of CAD planners for these scenarios. The particular CAD tools utilized were experimental node planners under development by the C.S. Draper Laboratory, Inc., known as PLATO [32] and YARD [4], respectively. We formally record results for eight different scenarios, but note that during extensive experimentation and debugging with our node planners, we never encountered a failure of MSA to quickly converge, at least when the relaxation factor γ of Section 2.4 was less than 1. We did sometimes encounter cycling with $\gamma = 1$, but we were always able to attribute it to suboptimal and inconsistent behavior of our CAD planners.

Our next set of experiments assess the value of target times with an MSA implementation utilizing our particular CAD planners. We generate targets in two ways. First, we solve an IP model of the planning problem as described in Chapter 3 to get one set of targets. Second, we utilize the following

	Num Iterations	Time
A	12	3:05
B	13	3:30
C	11	2:55
D	14	3:40
E	13	3:25
F	15	4:00
G	12	3:10
H	12	3:10

Table 2.3: Convergence of MSA under scenarios with varied congestion

heuristic rule to get another set:

1. generate an initial schedule using MSA with our CAD planners,
2. record where in the network each train incurs delay,
3. if a train f incurs a delay of t minutes on a particular node in its itinerary, increase its target arrival time at that and every subsequent node in its itinerary by αt,
4. repeat as often as desired.

In our case, α was $\frac{1}{3}$, and we iterated only once.

Table 2.4 presents the results of these approaches on the same network described above, except that YARD was replaced with a simple delay generator. We chose not to use YARD here because its optimization functionality was not yet developed at the time of these experiments, so offering it target times would have had no affect on its performance. We note that in half the cases, utilizing target times generated by an IP led to an improved solution, and in one case, utilizing targets generated by our heuristic did also. In no case did using target times degrade the solution, although there is no guarantee in general that it wouldn't.

The experiments of Table 2.4 were coded in C++ and run on a PowerMac 950 with a PowerPC 601 processor. The experiments of Table 2.3 were run on a PowerMac 700, and times were recorded with a wristwatch, so are accurate only to the nearest few seconds. We considered this an acceptable accuracy considering the times involved.

	Cost No Targets	*Cost w/ IP Targets*	*Cost w/ H Targets*	*Cost Optimal*	*Time MSA*	*Time IP*
I	5980	**5930**	5980	5930	1:17	0:15
J	5900	5900	5900	5900	1:32	0:10
K	6420	**6350**	**6400**	6350	1:24	0:17
L	7460	7460	7460	6890	1:26	0:2.67
M	7380	7380	7380	6690	1:20	0:3.05
N	8210	**8120**	8210	7300	1:24	0:6.27

Table 2.4: We ran MSA without target times, then with times generated from an analytical planner, and finally with times generated with a heuristic. We list the costs of final schedules generated by each approach and compare them to the cost of the optimal schedule. Finally, we present the time required to generate schedules with MSA and then with our analytical model.

2.6 Summary and Conclusions

In this chapter, we formulated MNSP and RNSP as a set of parallel planning subproblems. Although these subproblems can be difficult in their own right, they are often much more tractable than the overall planning problem, and a great deal of progress on them has already been made. In RNSP, the solution to such subproblems corresponds to the many CAD planning tools being developed. We then developed a heuristic coordination algorithm based on our understanding of the scheduling problem as a fixed-point problem. Although this heuristic approach will not necessarily find an optimal schedule, it empirically seems to do quite well at finding acceptable, feasible schedules for RNSP. However, the quality of our heuristic strongly depends on the kind of objective function we provide our sub-problems. For this reason, we define a class of objective functions which penalize deviation from a set of schedule targets which we hope in some sense to be "close" to optimal. In the next two chapters, we explore methods to generate such targets. We finally concluded by pointing out that our heuristic approach, and decomposition formulation in general, has potential applications in scheduling problems which are much broader than RNSP.

Chapter 3

Three IP Models for Railroad Scheduling and Sequencing

3.1 Introduction and Overview

In Chapter 2, we saw that the quality of the feasible schedules generated by our network model strongly depended on our ability to hand each sub-problem a good set of target arrival times. These target times could be generated off-line and utilized whenever we needed them in a real time setting. In the next two chapters, therefore, we explore methods of doing exactly this. In this chapter, we provide a {0-1} integer programming formulation of our rail scheduling problem and solve it to create one possible set of target times. In Chapter 4, we employ several heuristics which generate target times of comparable quality to those in this chapter, but which consume much less computational resource.

3.1.1 Chapter Summary and Statement of Intent

The IP models in this chapter are motivated by the work of Bertsimas and Stock [15] in modeling two sequencing and scheduling problems for air traffic, which they call the Ground-Holding Problem (GHP) and the Air-Traffic Flow Management Problem (TFMP). Both models are variations of MNSP*, and we call their IP formulation of MNSP* the Generic Bertsimas Stock model (GBS). GBS is typicaily a very large {0-1} IP which possess four important properties which make it an

attractive generic model:

1. *Strong Formulation:* The LP relaxation of GBS yields a good approximation to the underlying IP. This was observed and reported by Bertsimas and Stock [15].
2. *Hidden Network Structure:* The LP relaxation of GBS contains an underlying network structure which can be exploited to yield better computational performance.
3. *Reliable Branching Strategy:* We empirically observe a certain branching rule to be effective at solving GBS.
4. *Natural Decomposition:* GBS allows for a natural decomposition via Lagrange relaxation, and the dual, decomposed problem empirically seems to be both:
   (a) A strong dual (has zero duality gap), and
   (b) Well solved via a certain heuristic price update procedure, known as *Everett's method.*

For all these reasons, GBS is appealing as a generic IP model. The last three points represent contributions of this thesis. GBS's principle drawback is that it involves a large number of variables and constraints.

We formulate three application specific models which we derive from MNSP*: the Line Meet-Pass Model (LMPM), the Yard Switching and Sequencing Model (YSSM), and the Yard Dispatching Model (YDM); all were introduced in Chapter 1. Extensive computational experiments which we perform with all three models demonstrate that GBS and its three derivative models have promising potential to be utilized as planning tools under the following conditions:

1. Line topology is as homogeneous as possible, and
2. Traffic intensity is low to moderate, and
3. Expected delay times are low to moderate,

or

1. Off-line, batch scheduling is all that is needed, not iterative scheduling with human interaction, and
2. Computational resources are significant.

Where these conditions are not satisfied, we expect that the heuristic approaches of Chapter 4 will be preferable to any of the GBS derivative models of this chapter.

3.1.2 Chapter Outline

The remainder of this chapter is structured as follows:

1. *Sections 3.2, 3.3, and 3.4, GBS:* We present GBS as an IP formulation of MNSP* and introduce most of the notation needed for this chapter. We then introduce IP formulations of LMPM, YSSM, and YDM.

2. *Section 3.6, Solution of Models:* We introduce and explore the principal features of GBS which make it conducive to effective solution:

   (a) Strong Formulation

   (b) Underlying Network Structure

   (c) Effective Branching Strategy

   (d) Natural Decomposition Scheme

   All these features are exploited to allow us to solve much larger planning problems than we initially thought possible.

3. *Section 3.7, Computational Experiments:* We describe realistically sized computational experiments involving all three application models, and discuss our impressions from their results.

4. *Section 3.8, Conclusion and Summary:* Finally, we summarize our conclusions about modeling rail operations with a large-scale {0-1} IP, and restate our contributions toward this effort.

3.2 Generic Bertsimas/Stock IP Model (GBS)

We introduce the IP formulation of MNSP*, which we call GBS, and will form the basis of the IP formulations of the three application models presented in Section 1.3: LMPM, YSSM, and YDM. Our decision variables, in the notation of [15], look like:

Decision Variables:

$$w_{ft}^{j} = \begin{cases} 1 & \text{if job } f \text{ arrives at processor } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

These variables will be defined only for those processors in the list P_f for any job f, and for those times in T_f^j for any job f and processor $j \in P_f$. It is the modeler's imperative, therefore, to keep the lists T_f^j as small as possible in order to restrain the size of any model derived from GBS. To illustrate these variables, consider the LMPM scenario of Figure 3-1. The encoding of this journey is represented in Figure 3-2.

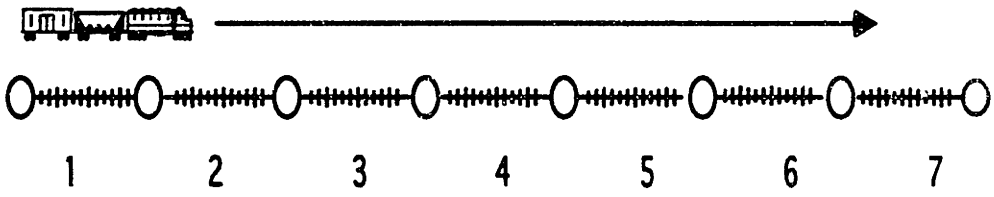


Figure 3-1: Single train travels across a line with seven track segments. Each segment takes one time period to traverse. Train arrives at segment 1 at time 1.

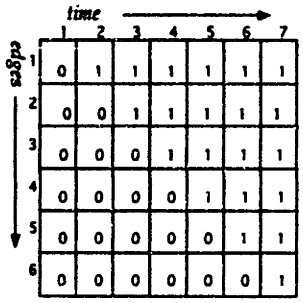


Figure 3-2: Encoding of trajectory of Figure 3-1.

Given these variables, the term

$$w_{ft}^{P(f,N_f)} - w_{f,t-1}^{P(f,N_f)}$$

will be 1 precisely when t equals the time job f completes all its processing, and zero otherwise. The objective function implied by the decision variables and the cost of arrival codified by data c_{ft} is therefore given by:

Objective Function:

$$\min \quad \sum_{f \in \mathcal{F}} \sum_{t \in T_f^{P(f,N_f)}} c_{ft}(w_{ft}^{P(f,N_f)} - w_{f,t-1}^{P(f,N_f)})$$

Note moreover that using c_{ft}, one can model any arbitrary function of arrival times which is separable by job. Setting

$$c_{ft} = \begin{cases} 0 & \text{for } t \leq d_f \\ t - d_f & \text{otherwise} \end{cases}$$

for example, minimizes the total tardiness of all jobs, while

$$c_{ft} = (t - d_f)^2$$

minimizes the sum squared deviation from target arrival times, and

$$c_{ft} = \begin{cases} 0 & \text{for } t \leq d_f \\ 1 & \text{otherwise} \end{cases}$$

minimizes the number of late jobs. Finally, the constraints implied by these decision variables and the problem data fall into four generic categories: capacity constraints, time-connectivity constrains, processor connectivity constraints, and job connectivity constraints.

Capacity constraints enforce capacity requirements for each processor at each time period. If a job has processor $j, \kappa_f(j) < N_f$ in its path, then

$$w_{ft}^{j} - w_{ft}^{\nu_f(j)}$$

is 1 if job j is at processor j at time t, and zero otherwise, so the number of jobs at any processor j at any time t is expressed

$$\sum_{f, j:\kappa_f(j)<N_f} (w_{ft}^{j} - w_{ft}^{\nu_f(j)})$$

and our capacity constraints are modeled as:

$$\sum_{f,j:\kappa_f(j)<N_f} (w^j_{ft} - w^{\nu_f(j)}_{ft}) \leq C_j(t) \quad \forall j \in \mathcal{J}, t \in \mathcal{T}$$

Time connectivity constraints simply require that $w^j_{f,t+1}$ be 1 if $w^j_{ft} = 1$, as is implied by the definition of w:

$$w^j_{ft} - w^j_{f,t-1} \geq 0 \quad \forall f \in \mathcal{F}, t \in T^j_f, j \in P_f.$$

Processor connectivity constraints enforce processing time requirements by requiring that f not enter processor $\nu_f(j)$ in its path until at least l_{fj} time periods after it has entered processor j:

$$w^{\nu_f(j)}_{f,t+l_{fj}} - w^j_{ft} \leq 0 \quad \forall f \in \mathcal{F}, t + l_{fj} \in T^{\nu_f(j)}_f, \kappa_f(j) < N_f.$$

Job connectivity constraints enforce the inter-job precedence constraints $\mathcal{R}$. If $r = ((f_r, j_r); (f'_r, j'_r); s_r) \in \mathcal{R}$, then f'_r may not arrive at j'_r until at least s_r time periods after f_r has arrived at j_r:

$$w^{j'_r}_{f'_r,t} - w^{j_r}_{f_r,t-s_r} \leq 0 \quad \forall r \in \mathcal{R}.$$

If we define $\overline{T}_{f_j}$ to be the time after the last time in T^j_f, and $\underline{T}_{f_j}$ to be the period before the first period in T^j_f, then we can set

$$w^j_{ft} \equiv \begin{cases} 1 & t > \overline{T}_{f_j} \\ 0 & t < \underline{T}_{f_j} \end{cases}$$

and eliminate these variables. Thus all traffic must travel through the network and arrive at its destination eventually, but no sooner than is permitted by the exogenous constraints of the model.

The complete generic Bertsimas/Stock formulation of MNSP*:

[GBS]:

$$\min \quad \sum_{f \in \mathcal{F}} \sum_{t \in T^{P(f,N_f)}_f} c_{ft}(w^{P(f,N_f)}_{ft} - w^{P(f,N_f)}_{f,t-1})$$

such that:

$$\sum_{f,j:\kappa_f(j)<N_f} (w^j_{ft} - w^{\nu_f(j)}_{ft}) \leq C_j(t) \quad \forall j \in \mathcal{J}, t \in \mathcal{T} \tag{3.1}$$

$$w^j_{ft} - w^j_{f,t-1} \geq 0 \quad \forall f \in \mathcal{F}, t \in T^j_f, j \in P_f \tag{3.2}$$

$$w^{\nu_f(j)}_{f,t+l_{fj}} - w^j_{ft} \leq 0 \quad \forall f \in \mathcal{F}, t + l_{fj} \in T^{\nu_f(j)}_f, \kappa_f(j) < N_f \tag{3.3}$$

$$w^{j'_r}_{f'_r,t} - w^{j_r}_{f_r,t-s_r} \leq 0 \quad \forall r \in \mathcal{R} \tag{3.4}$$

$$w^j_{ft} \in \{0,1\} \tag{3.5}$$

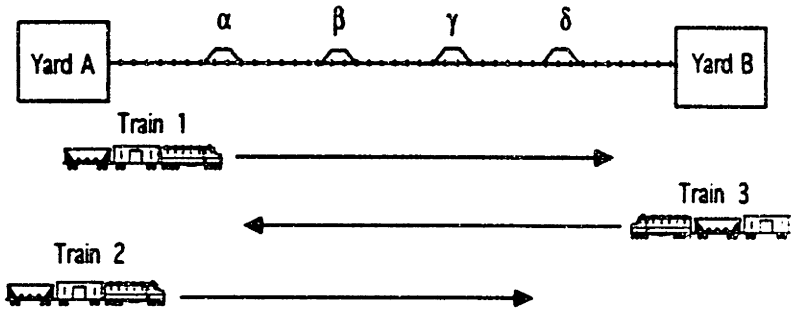
We call the region defined by the LP relaxation of GBS the *Bertsimas/Stock Polyhedron*, or *BS Polyhedron*. As was said before, our IP formulations of LMPM, YSSM, and YDM will require different modifications of the above, but the fundamental ideas will be the same. We therefore now describe these three applications, focusing on the differences between them and the generic model GBS. In all three applications, jobs will become trains and processors will become either track segments (in LMPM) or terminal areas and resources (YSSM, YDM).

3.3 First Application: Line Meet-Pass Model (LMPM)

Our first application of GBS is to modeling linehaul dispatching as described in Section 1.2.3 of this thesis and represents the evolution of work first described in [24].

3.3.1 Example of linehaul dispatching problem

As an example of the combinatorial subtlety of linehaul dispatching, consider the scenario depicted in Figure 3-3. There, we have three trains crossing a short line of track with four sidings, at α, β, γ and δ, and release dates, priorities, and target arrival times given in the table. Let us assume, for example, that this is a sparse section of track, so that the sidings are a full hour's travel time apart for each train, but the sidings themselves take a negligible time to traverse. One's initial inclination in such a situation might be to have Train 3, the highest priority train, take priority over 1 at siding



train	*start time*	*cost coeff*	*target arrival*
1	1	10	6
2	3	9	8
3	1	11	6

Figure 3-3: Train priorities suggest Train 3 takes priority over 1, but the reverse is actually optimal.

β, forcing 1 to wait at the siding. This is in fact the solution chosen by at least one experimental line planning tool with which we have worked, and would be the decision made by any priority-rule based scheduling system. In reality, the reader should verify for himself that the optimal solution is to have Train 1 take priority over 3 at siding γ. Doing so will allow Train 1 and 3 to time their subsequent meet perfectly, minimizing unnecessary waiting delay in the second meet of the scenario.

3.3.2 Formulation of Model

Data

The model data are the same as those presented in Section 1.3.2.

Decision Variables:

$$w_{ft}^{\rightarrow j} = \begin{cases} 1 & \text{if train } f \text{ traveling} \rightarrow \text{arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

$$w_{ft}^{\leftarrow j} = \begin{cases} 1 & \text{if train } f \text{ traveling} \leftarrow \text{arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_{jt}^{\rightarrow} = \begin{cases} 1 & \text{if segment } j \ (j \notin \mathcal{M}) \text{ is occupied by trains} \rightarrow \text{at during } (t, t+1) \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_{jt}^{\leftarrow} = \begin{cases} 1 & \text{if segment } j \ (j \notin \mathcal{M}) \text{ is occupied by trains} \leftarrow \text{at during } (t, t+1) \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{mt}^{\rightarrow} = \begin{cases} 1 & \text{if} \rightarrow \text{traffic around meetpoint } m \text{ at during } (t, t+1) \text{ will take siding} \\ 0 & \text{otherwise} \end{cases}$$

$$\gamma_{mt}^{\leftarrow} = \begin{cases} 1 & \text{if} \leftarrow \text{traffic around meetpoint } m \text{ during } (t, t+1) \text{ will take siding} \\ 0 & \text{otherwise} \end{cases}$$

The principal addition to GBS in this formulation are the binary variables $\delta_{jt}^{\leftarrow}$, $\delta_{jt}^{\rightarrow}$, $\gamma_{mt}^{\leftarrow}$, and $\gamma_{mt}^{\rightarrow}$. The δ's were first introduced in [24], and are indicator variables designating the states s_k of our "processors," as described in Section 1.3. We exclude definition of δ's on meetpoints, because these segments are in reality double track. Trains can travel across them in both directions simultaneously. We will discuss the use of the γ's shortly.

Besides these changes, the formulation is exactly as presented in Section 3.2.

Objective Function to minimize:

$$\sum_{f \in \mathcal{F}^{\rightarrow}} \sum_{t \in T_f^{P(f,N_f)}} c_{ft}(w_{ft}^{\rightarrow P(f,N_f)} - w_{f,t-1}^{\rightarrow P(f,N_f)}) + \sum_{f \in \mathcal{F}^{\leftarrow}} \sum_{t \in T_f^{P(f,N_f)}} c_{ft}(w_{ft}^{\leftarrow P(f,N_f)} - w_{f,t-1}^{\leftarrow P(f,N_f)})$$

Our capacity constraints are almost exactly as before, except more numerous.

$$z_{jt}^{\rightarrow} = \sum_{f,j:\kappa_f(j)<N_f} (w_{ft}^{\rightarrow j} - w_{ft}^{\rightarrow \nu_f(j)}) \leq C_j(t)\, \delta_{jt}^{\rightarrow} \quad \forall j \in \mathcal{J} \backslash \mathcal{M}, t \in \mathcal{T} \tag{3.6}$$

$$z_{jt}^{\leftarrow} = \sum_{f,j:\kappa_f(j)<N_f} (w_{ft}^{\leftarrow j} - w_{ft}^{\leftarrow \nu_f(j)}) \leq C_j(t)\,\delta_{jt}^{\leftarrow} \quad \forall j \in \mathcal{J}\backslash\mathcal{M}, t \in \mathcal{T} \tag{3.7}$$

$$z_{mt}^{\rightarrow} = \sum_{f,j:\kappa_f(j)<N_f} (w_{ft}^{\rightarrow m} - w_{ft}^{\rightarrow \nu_f(j)}) \leq C_m \quad \forall m \in \mathcal{M}, t \in \mathcal{T} \tag{3.8}$$

$$z_{mt}^{\leftarrow} = \sum_{f,j:\kappa_f(j)<N_f} (w_{ft}^{\leftarrow m} - w_{ft}^{\leftarrow \nu_f(j)}) \leq C_m \quad \forall m \in \mathcal{M}, t \in \mathcal{T} \tag{3.9}$$

where the z_{jt}'s count the number of trains of the given orientation on segment j at time t. The next constraint ensures that at any time, a non-meetpoint segment of track is assigned one and only one direction:

$$\delta_{jt}^{\rightarrow} + \delta_{jt}^{\leftarrow} = 1 \qquad \forall j \in \mathcal{J}\backslash\mathcal{M}, t \in \mathcal{T} \tag{3.10}$$

Capacity constraints are not enough in LMPM, however, as the scenario of Figure 3-4 makes clear. We therefore make the following claim: if segments j and $j+1$ are separated by meetpoint

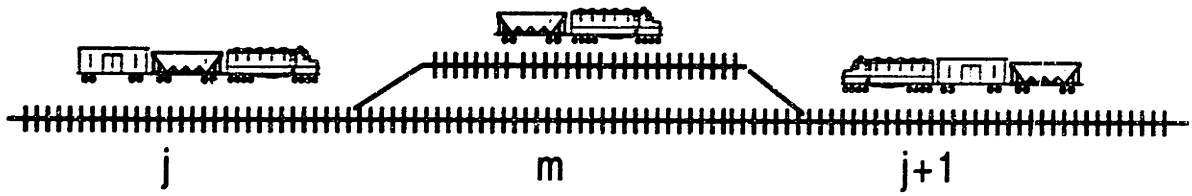


Figure 3-4: Let the capacity of segments j and $j+1$ be high, say five, and that of the siding at meetpoint m be one. At time t, there are five trains traveling east on j, and five trains traveling west on $j+1$. At time $t+1$, they switch, so that the five eastbound trains move to $j+1$, and the five westbound trains to j. Clearly, at both times t and $t+1$, the capacity constraints are satisfied, but we know that the only way for two sets of trains traveling in opposite directions to meet like this is for one of the *entire* sets to cross over the siding. Since the capacity of the siding is one, this clearly cannot happen.

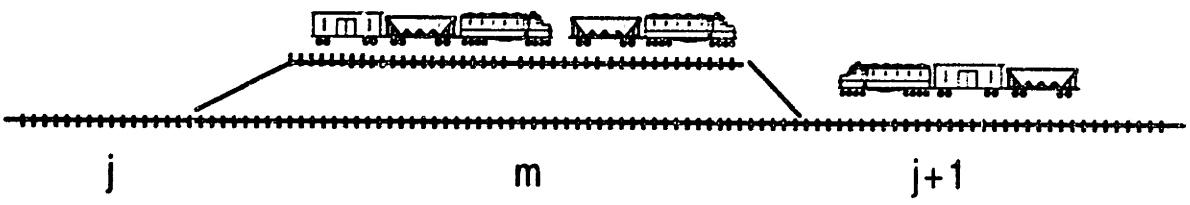


Figure 3-5: Right-bound traffic takes siding

m and if $\delta_{jt}^{\rightarrow} = 1$ and $\delta_{j+1,t}^{\leftarrow} = 1$ then either

$$z_{jt}^{\rightarrow} + z_{mt}^{\rightarrow} \leq C_m,$$

and the entire set of right-bound trains takes the siding, as in Figure 3-5, or

$$z_{jt}^{\leftarrow} + z_{mt}^{\leftarrow} \leq C_m,$$

and the entire set of left-bound trains "takes the siding," which means they take either the siding itself or the track under the siding. We recall the definition of our binary decision variables γ, and capture this constraint with:

$$z_{jt}^{\rightarrow} + z_{mt}^{\rightarrow} \leq C_m + C_j(t)[3 - \gamma_{mt}^{\rightarrow} - \delta_{j,t}^{\rightarrow} - \delta_{j+1,t}^{\leftarrow}] \quad \forall j \in \mathcal{J}\backslash\mathcal{M}, t \in \mathcal{T} \tag{3.11}$$

$$z_{j+1,t}^{\leftarrow} + z_m^{\leftarrow}(t) \leq C_m + C_{j+1}(t)[3 - \gamma_{mt}^{\leftarrow} - \delta_{j,t}^{\rightarrow} - \delta_{j+1,t}^{\leftarrow}] \quad \forall j+1 \in \mathcal{J}\backslash\mathcal{M}, t \in \mathcal{T}. \tag{3.12}$$

If $\delta_{jt}^{\rightarrow} = \delta_{j+1,t}^{\rightarrow}$, there is no meet and movement is valid so long as our capacity constraints are valid.

We conclude our formulation with the remaining constraints, all of which follow directly from the motivation in Section 3.2, and were explained there.

$$w_{f,t+l_{fj}}^{\rightarrow\nu_f(j)} - w_{ft}^{\rightarrow j} \leq 0 \tag{3.13}$$

$$w_{f,t+l_{fj}}^{\leftarrow\nu_f(j)} - w_{ft}^{\leftarrow j} \leq 0 \tag{3.14}$$

$$\forall f \in \mathcal{F},\ t \in T_f^j,\ j : \kappa_f(j) < N_f$$

$$w_{ft}^{\rightarrow j} - w_{f,t-1}^{\rightarrow j} \geq 0 \tag{3.15}$$

$$w_{ft}^{\leftarrow j} - w_{f,t-1}^{\leftarrow j} \geq 0 \tag{3.16}$$

$$\forall f \in \mathcal{F},\ t \in T_f^j,\ j \in P_f$$

$$w_{f_r',t}^{\rightarrow j_r'} - w_{f_r,t-s_r}^{\rightarrow j_r} \leq 0 \tag{3.17}$$

$$w^{\overleftarrow{j'_r}}_{f'_r,t} - w^{\overleftarrow{j_r}}_{f_r,t-s_r} \leq 0 \tag{3.18}$$

$$\forall r \in \mathcal{R}$$

$$w^{\overrightarrow{j}}_{ft}, w^{\overleftarrow{j}}_{ft}, \gamma^{\rightarrow}_{mt}, \gamma^{\leftarrow}_{mt}, \delta^{\rightarrow}_{jt}, \delta^{\leftarrow}_{jt} \in \{0,1\} \tag{3.19}$$

Constraints (3.13) and (3.14) represent connectivity between the segments, constraints (3.15) and (3.16) represent time connectivity, and constraints (3.17) and (3.18) represent connectivity between trains in separate lines. In the case of LMPM, such connectivity could result because one train carries cars that must go into another at a terminal, or, more commonly, because two trains j and j' in separate dispatcher regions correspond to the same physical train, but under the authority of different shift masters.

This completes our IP formulation of LMPM. We note that in this formulation, we need not model just a single line, but conceivably any number of interdependent lines linked by constraints (3.17) and (3.18), which can be relaxed to create separate problems. In Section 3.6.3 we will discuss this decomposition, and in Section 3.7, we will discuss computational results from test scheduling scenarios.

3.3.3 Segments as Signal Blocks

What happens on heavily used lines or corridors, where the frequency of sidings and meetpoints on the track is very large? There, track segments (the track between meetpoints) tend to be fairly short, so that at most one signal block[1] separates each pair of sidings. If we assume we have a line where all segments correspond to individual signal blocks, we can set all non-meetpoint segment capacities to one, which significantly simplifies our formulation. In this case, the z_{ij}'s are either zero or one, and hence completely determine the δ's for such segments, which allows us to eliminate them. In the case of meetpoints, we can also eliminate the need for γ variables by lowering total meetpoint capacities from 2 to 1. Although this technically does not correspond to physical reality, we can imagine the "second" train at any meetpoint to in fact occupy the adjacent segment from which it came. Therefore, under the special case of single signal-block per segment, we are left with

[1] A *signal block* is an edge of track on which, because of the signaling mechanism, only one train is ever allowed at a time. It corresponds to the smallest "discrete" segment of track.

an IP in the original variables of Section 3.2, with the addition of orientations: $w_{ft}^{\rightarrow j}$ and $w_{ft}^{\leftarrow j}$.

The capacity constraints (3.6) and (3.7) are combined into the following:

$$z_{jt}^{\rightarrow} + z_{jt}^{\leftarrow} \leq 1 \qquad \forall j \in \mathcal{J}\ t \in \mathcal{T} \tag{3.20}$$

Moreover, we can also compact constraints (3.11) and (3.12). We want to capture the following physical restrictions:

- If $z_{mt}^{\rightarrow} = 1$ then $z_{jt}^{\rightarrow} + z_{j+1,t}^{\leftarrow} \leq 2$
- If $z_{mt}^{\leftarrow} = 1$ then $z_{jt}^{\rightarrow} + z_{j+1,t}^{\leftarrow} \leq 2$
- If $z_{mt}^{\rightarrow} + z_{mt}^{\leftarrow} = 0$ then $z_{jt}^{\rightarrow} + z_{j+1,t}^{\leftarrow} \leq 2$

But these are automatically enforced by constraints (3.20), so we need not worry about them.

3.3.4 Extensions and Features of Formulation

The above formulation of LMPM can be expanded in various ways for added fidelity or performance.

1. *Double Track:* We can model normal double track in the exact framework we have established. By "normal double track," we mean double track such that each track has a fixed direction. We can do this simply by fixing the relevant direction variables δ to their appropriate values. Double track with unspecified orientation can also be modeled, but will require the addition of elementary routing decisions (see below).

2. *Arbitrary Topology:* The reader should observe that nothing in our formulation requires that the "line" we model with this methodology be linear. What we really are modeling are dispatcher regions, which we call "lines" for historical reasons. As long as the dispatcher region is made up of segments which have only two orientations, the segments themselves may be connected in any arbitrary fashion. However, meetpoints that have more than two adjacent segments will require a separate set of constraints of the form of (3.11) and (3.12) for each pair of adjacent segments.

3. *Routing Decisions :* The model can also make minor routing decisions, as discussed in [15]. For example, suppose we have double track where the orientation of each track is unspecified.

The dispatcher therefore must choose which track to send a given train over, and can use the double track for meets. We can model this choice by modifying our decision variables w_{ft} to be of the form $w_{ft}^{\rightarrow j,k}$ and $w_{ft}^{\leftarrow j,k}$, where k represents one of the possible routes of train f. Clearly, the size of the formulation will increase exponentially fast with the number of possible decisions, so this can be realistic only if there are a very small number of places we are willing to allow alternate routing.

4. *Maintenance of Way:* Regular, pre-scheduled maintenance of track (called "MOW") may be modeled in this approach simply by adjusting time-based capacities $C(t)$ at appropriate segments.

3.3.5 Limitations of LMPM

The principal limitation of the model is that the selection of appropriate time-lengths can be a problem. The desire for greater fidelity drives us to discretize time into as small units as possible, especially since events such as the traversal of a siding can take as little as five minutes. Moreover, varying train speeds can aggravate this situation. If we wish to accurately model the fact that one train takes 20 minutes to traverse a segment while another takes only 17, then our time unit must be no more than 1 minute. Unfortunately, this model is extremely sensitive to dimension, and the modeler using it must be very careful to prune its dimension as much as possible. Similarly, acceleration and deceleration delays, which can be very significant for long, heavy trains, are not captured by our discrete-space, discrete-time model. Here, every segment takes a fixed number of time periods to traverse, regardless of what a train's velocity was before entering the segment. This is certainly not the case in practice. Finally, our representation of meetpoints as nothing more than abstract entities which permit some number of trains to overtake some other number of trains is an important simplification. Often, with complicated sidings and spurs, the possible meet-pass scenarios depend on the lengths of the trains involved and the order in which they arrive.

For all these reasons we do not see a role for LMPM as a reliable online aid in linehaul dispatching, but rather as a line-traversal or line-delay model to assist the general scheduling process. Given a set of start times for trains traversing a line, LMPM could be counted on to predict optimal traversal times for each train, and do so in a way that takes explicit account of the combinatorial constraints involved in each train's itinerary. In this sense it might be preferable to the many two or three-factor models of line delay in the literature mentioned in Section 1.4 ([23], [61], [62], [39]).

3.4 Second Application: Yard Switching and Sequencing

Our second application is to car switching and sequencing in the classification terminal as described by Armacost [4]. In Section 1.2.2, we described the physical process being modeled, and in Section 1.3.3, we described the application of MNSP* to YSSM. Here we describe our representation of YSSM as an instance of GBS.

3.4.1 YSSM Formulation

IP formulation of YSSM is exactly as the generic model GBS.

Generic Decision Variable:

$$w_{it}^{j} = \begin{cases} 1 & \text{if Inbound Train } i \text{ arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

$$w_{ot}^{j} = \begin{cases} 1 & \text{if Outbound Train } o \text{ arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

$$w_{bt}^{j} = \begin{cases} 1 & \text{if Block } b \text{ arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

YSSM Objective Function:

$$\min \quad \sum_{b \in \mathcal{DB}} \sum_{t \in T_b^9} c_{bt}(w_{bt}^{9} - w_{b,t-1}^{9}) + \sum_{o \in O} \sum_{t \in T_o^9} c_{ot}(w_{ot}^{9} - w_{o,t-1}^{9})$$

YSSM Constraints

Capacity constraints, Time Connectivity constraints, and Segment connectivity constraints are exactly as specified generically in GBS. The precedence constraints are also identical, yet more precisely specified for this particular application:

$$w^4_{it} - w^5_{b,t-s_{bi}} \leq 0 \quad \forall (i,b) : C(i) \cap C(b) \neq \emptyset \tag{3.21}$$

$$w^8_{ot} - w^9_{b,t-s_{bo}} \leq 0 \quad \forall (o,b) : C(o) \cap C(b) \neq \emptyset \tag{3.22}$$

$$w^6_{bt} - w^9_{i,t-s_{ib}} \leq 0 \quad \forall (i,b) : C(o) \cap C(b) \neq \emptyset \tag{3.23}$$

$$w^7_{bt} - w^6_{o,t-s_{ob}} \leq 0 \quad \forall (o,b) : C(o) \cap C(b) \neq \emptyset \tag{3.24}$$

3.4.2 Limitations of YSSM

The principal limitation of the YSSM is that the dimensions of this model are large. A typical car spends 20 hours in a major classification terminal, with a standard deviation of 10 hours [33]. Moreover, a single train can carry as many as six or seven different blocks from a terminal [44], which means that a large classification terminal can see hundreds of blocks per planning day. If we discretize time by 15 minute intervals, which is the typical time it takes to pull a single block from the Classification Yard, a YSSM IP model would have approximately $4 \times 4 \times 20 \times 200 = 64.000$ columns for normal blocks alone. While this might acceptable for a stand-alone model, it is not where it is to be just one subproblem in a network of interdependent models.

Fortunately, the number of major classification terminals is usually rather small (4 - 6) for a Class I railroad. Most terminals are what we call switching terminals (see Chapter 1). Moreover, for the purposes of network planning, it is possible that explicit modeling of block movement through the classification yard is more detail than we need, which motivated the development of YDM. In Section 3.7, we present computational experience with our IP formulation of YSSM. Chapter 4 presents computational experience for a different formulation of YSSM.

3.5 Third Application: Yard Dispatching Model

YDM was first introduced in Section 1.3.4, and we know present our IP formulation of it.

Generic Decision Variable:

Our decision variables are as in YSSM, except that we no longer model blocks, and outbound train paths are different:

$$w_{it}^{j} = \begin{cases} 1 & \text{if Inbound Train } i \text{ arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

$$w_{ot}^{j} = \begin{cases} 1 & \text{if Outbound Train } o \text{ arrives at segment } j \text{ by time } t \\ 0 & \text{otherwise} \end{cases}$$

YDM Objective Function:

The objective function is exactly as in YSSM, except that we no longer explicitly model the delivery of Dest Cuts.

$$\min \qquad \sum_{o \in O} \sum_{t \in T_o^8} c_{ot}(w_{ot}^{8} - w_{o,t-1}^{8})$$

Formulation Constraints

Capacity constraints, Time Connectivity constraints, and Segment connectivity constraints are exactly as specified generically in Section 3.2. The precedence constraints state that an outbound train may not begin assembly until all inbound trains carrying its cars have been disassembled or switched.

$$w_{ot}^{5} - w_{i,t-s_{io}}^{8} \leq 0 \quad \forall (i,b) : \mathcal{C}(i) \cap \mathcal{C}(b) \neq \emptyset \tag{3.25}$$

The key combinatorial flavor of the assembly/disassembly process is captured and honed by varying the time it takes takes to traverse Assembly and Disassembly, as well as the buffer s_{io}.

3.6 Solving the IP Models

Now that we have presented the generic Bertsimas Stock formulation of MNSP*, and our three rail-specific applications of it, we are ready to turn to the issue of solving these models. They are,

of course, large scale {0-1} IP's, so solving any of these models shall be a challenge. In order to do so, we shall rely on four key features of the models which were first mentioned in Section 3.1.

1. The LP relaxation yields a strong approximation to the underlying IP. This is a key property of the Bertsimas/Stock Polyhedron.
2. The LP relaxations also have an underlying network structure which can be exploited to speed solution time.
3. There exist branching rules which can be effective at solving the IP's under certain conditions.
4. Lagrange relaxation can be used to decompose the network-wide IP into independent subproblems to be solved simultaneously.

In this section, we explore the implications of each of these in the general context of GBS.

3.6.1 Bertsimas/Stock Polyhedron

The most immediately interesting feature of GBS is that it is a "strong formulation." Namely, Bertsimas and Stock observe in GHP, their principal application, that they almost always generate integer-optimal solutions after solving only the LP relaxation of their IP's. Their explanation for this observation is Proposition 3.1.

Proposition 3.1 *Constraints (3.2), (3.3), and (3.4) define facets of the convex hull of integer solutions defined by GBS.*

Proof: See [15] for proof. ■

Therefore, the BS Polyhedron is likely to be close to the convex hull of integer solutions. We have an additional explanation for that same observation, and it is given by the following Proposition:

Proposition 3.2 *The BS polyhedron, without constraints (3.1), is the dual to a Network Flow polyhedron, and hence completely integral.*

Proof: Consider relaxing capacity constraints (3.1), modeling only constraints (3.2), (3.3), and (3.4). Notice that these constraints put exactly one +1 and one −1 in every row of the constraint matrix, so that this constraint matrix is the transpose of a {0,1} node-arc incidence matrix. It therefore represents the constraint matrix of the dual to some network flow problem. Integrality of the polyhedron follows from the integrality of the right hand side. ■

Problem	IP Value	Relax Value	Ratio
1	-4554.5264	-4560.1399	0.999
2	- 678.00	-679.37	0.998
3	-3754.5285	-3847.3461	0.976
4	-426.493	-427.591	0.997
5	-4295.588	-4301.7103	0.999
6	-1340.00	-1405.00	0.954
7	-110.00	-155.00	0.710
8	-1990.00	-2020.00	0.985
9	-2280.00	-2480.00	0.919
10	-1140.00	-1240.00	0.919

Table 3.1: Bounds from LP relaxation, BS Polyhedron

This is an appealing observation. After all, the scheduling problems modeled by this formulation should cease to be difficult once capacity constraints disappear. After all, without capacity constraints, trains can move independently of each other, so the optimal solution is trivial to calculate. Moreover, the cardinality of capacity constraints is just $|J| \times |T|$, while the cardinalities of each of the other three sets of constraints are $O(|J| \times |T| \times |F|)$, so that capacity constraints, which can be thought of as the "complicating constraints" in our problem, are a minority of the cuts defining the BS Polyhedron.

In any case, in GHP most capacity constraints are not modeled; only arrival constraints at airports (landing restrictions) are considered. In our applications, unfortunately, capacity constraints are much more numerous. Consider, for example, the linehaul dispatching problem LMPM. Any segment of track on any line anywhere is in theory a potential bottleneck for the network. In our applications, therefore, and unlike the case in the GHP, we will almost never be able to get integer solutions from the LP relaxations of these IP's, and must therefore solve the IP's themselves. Fortunately, we will at least see that our LP relaxations can give us very good bounds, as Table 3.1 demonstrates. There, we have randomly sampled 10 different IP's solved as part of the experiments in Section 3.7, and compared the values of the IP's and the corresponding LP relaxations. In part because of such bounds, utilizing a branch and bound scheme to solve the full IP need not be ruled out.

The value of Proposition 3.2, however, is in more than just another explanation for the results of Table 3.1. We can speed the solution of the LP relaxation of GBS significantly by exploiting an important implication of that result. Rather than solving the LP relaxations directly, we relax the capacity constraints and solve the remaining network flow problem, getting an initial, infeasible

	trains	*segments*	*cols*	*rows*	*LP time*	*Net time*
A	20	15	18000	25000	1:28:51	0:1:25
B	20	15	12000	18000	0:47:19	0:0:41
C	20	15	8000	12000	0:31:27	0:0:27

Table 3.2: Combined Network-Simplex vs. Simplex Method

basis. We then add our capacity constraints to our formulation and solve the LP, starting from that network basis. Possible performance gains of using this approach are illustrated in Table 3.2, which lists three sample instances of LMPM. The final two columns give, first, the time to solve the LP directly, and then the time to solve the LP by first solving the underlying network problem. All problems were solved with CPLEX 3.0 on a PowerMac 950, and demonstrate that changing the way we solve our LP relaxation can have a very large effect on computation times.

Moreover, for many objective functions, we need not even solve the embedded network flow problem to get the initial, network optimal basis. If, for example, the objective function is such that each train should be rushed to its destination as soon as possible, then the solution to the embedded network-flow problem will be to set all decision variables equal to 1, which is always possible in absence of capacity constraints due to the definitions of T_f^j.

The dividend of starting with an initial network-flow basis depends strongly on the length of the time-windows T_f^j. The greater these are relative to the total length of the planning horizon $|T|$, the greater the relative cardinality of constraints (3.2), (3.3), and (3.4) to capacity constraints (3.1), the closer our formulation is to the underlying network flow problem, and the greater is the value of computing the advanced network-flow basis. Later in this chapter we will discuss the profound importance of trimming these time-windows as much as possible, which will therefore reduce the relative value of solving the embedded network flow problem significantly. Even after doing so, however, starting from a network-flow basis does still yield important (40-75%) performance improvements which will be critical to our ability to solve any of the derivative models of GBS as full IP's.

3.6.2 Branching Strategy

In addition to a network structure which speeds solution of the LP relaxation of GBS, we can also make use of a particular branching strategy which empirically seems to be effective at solving GBS. In what follows, the term "destination variable" will refer to any variable w_{ft}^j such that $j = P(f.N_f)$,

and "origin variable" will refer to any w^j_{ft} such that $j = P(f,1)$. The strategy comprises the three elements of Procedure 1, which describes a basic version of the strategy applicable where arrival costs are linear in time ($c_{f,t+1} - c_{f,t} = c_f$, $\forall f \in \mathcal{F}$). A more general version of this strategy will be described in Section 3.6.4.

1: *Selective Integrality Enforcement:* Enforce binary requirements only on destination variables. All other integrality restrictions are relaxed.
2: **for all** Binary Variables **do**
3: *Fixed Branching Order:* Assign branching priority based on an index which is calculated as follows. Generate index I_f:

$$I_f = \alpha c_f + (1-\alpha) \sum_{r \in \mathcal{R}: f = f_r} c_{f'}$$

for each train f, then utilize index

$$I^j_{f,t} = I_f + \epsilon t$$

for each variable $w^j_{f,t}$, where $\epsilon > 0$ but small.
4: **end for**
5: *Specified Branching Direction:* Set preferred branch direction to "up," which means we fix variable to 1 before fixing to 0.

Procedure 1: Apply when arrival costs are linear in time. Note that I_f is a weighted combination of the value of f plus the sum of the values of any trains with which it has a precedence relationship.

One interesting empirical observation we have made, but which we have so far been unable to fully explain, is that the branching strategy of Procedure 1 always generates integral solutions, even though we relax explicit integrality requirements on all but a minority of variables.

The reader should notice what a powerful branching strategy this is. Setting $w^6_{f,7} = 1$ in the scenario of Figure 3-1, for example, immediately forces the upper triangle of Figure 3-2 to one, completely fixing the schedule. Thus, at each node in the branching tree, we expect the right-branch LP to be significantly smaller than either the left-branch LP or parent LP.

3.6.3 Lagrange Decomposition and Everett's Method

We have utilized the branching scheme of Section 3.6.2 to successfully solve IP's with 5,000 - 15,000 columns and 15,000 - 30,000 rows, in times ranging from under one second to 15 minutes. Unfortunately, the large-scale applications we have described in this chapter can require 150,000 - 250,000 columns, far too large for our current time and especially physical memory constraints.

Our obvious options are some sort of column generation or decomposition, and since these network scheduling problems possess a natural geographical separability, the latter was the approach we took. We were also attracted to a decomposition strategy because we could then exploit par-

allelization in the solution process, distributing computation to multiple, simultaneous processors. In particular, we divided our network into lines and terminals, and removed constraints connecting train movements between these subproblems by bringing them into our objective function via a Lagrange relaxation. All such constraints are of the form of train connectivity constraints:

$$w^{j'_r}_{f'_r t} - w^{j_r}_{f_r, t-s_r} \leq 0$$

where j'_r is some segment in one subproblem (one line or terminal), at which train f'_r may not arrive until s_r time periods after f_r has arrived at j_r, which by definition must be in some other subproblem. We therefore let $\hat{\mathcal{R}} \subset \mathcal{R}$ be this set of connectivity constraints relating trains in separate sub-problems which are to be relaxed.

These constraints are brought into our objective function via the term

$$\sum_{r \in \hat{\mathcal{R}}} \lambda_r (w^{j'_r}_{f'_r t} - w^{j_r}_{f_r, t-s_r}) \tag{3.26}$$

where $\lambda = (\lambda_1, \ldots, \lambda_{\hat{\mathcal{R}}})$ must be chosen properly.

We employ an iterative price update procedure. At every iteration, some guess $\lambda(n)$ is made for λ, the independent sub-problems are solved, and the resulting train-connectivity infeasibility (or over-feasibility) is used to update $\lambda(n+1)$. One idea is a simple subgradient update rule:

$$\lambda_r(n+1) := \lambda_r(n) + \alpha(n)(w^{j'_r}_{f'_r t} - w^{j_r}_{f_r, t-s_r}) \tag{3.27}$$

where $\alpha(n)$ is an appropriately diminishing stepsize. As is common with this method, however, we found it very hard to control, even for small test problems. Making $\alpha(n)$ shrink too fast leads to premature convergence; making it shrink too slowly leads nowhere. In fact, although the preferred scheme is to shrink α arithmetically in n, making convergence at a non stationary point impossible, we found that, just as in Chapter 2, doing so almost always lead to unacceptable computational performance.

We therefore update prices via a little known technique attributed to Everett by Pugh [65] and suggested to us by Bertsimas [16], which we call *Everett's Method*. The technique is similar in spirit to the subgradient update rule: at each iteration n, consider every relaxed constraint row r and its associated price λ_r. If, at iteration n, constraint r is violated by the current solution $w(n)$,

Initialize constants δ_1 and δ_2 (1.15 and .33)			
Initialize prices and update parameters			
$V_r(0)$	:=	0.0;	
$\epsilon_r(0)$	:=	0.1;	
$\lambda_r(1)$	:=	1.0;	
...and for all n greater than 0			
$V_r(n)$	:=	$(w^{j_r}_{f_r t}(n) - w^{j'_r}_{f'_r, t-s_r}(n))$	
$\epsilon_r(n)$	:=	$\delta_1 \epsilon_r(n-1)$	if $V_r(n)V_r(n-1) > 0$;
$\epsilon_r(n)$	:=	$\delta_2 \epsilon_i(n-1)$	if $V_r(n)V_r(n-1) < 0$;
$\epsilon_r(n)$	:=	$\epsilon_r(n-1)$	if $V_r(n)V_r(n-1) = 0$;
$\lambda_r(n+1)$	:=	$(1+\epsilon_r(n))\lambda_r(n)$	if $V_r(n) > 0$;
$\lambda_r(n+1)$	:=	$(1-\epsilon_r(n))\lambda_r(n)$	if $V_r(n) \leq 0$.

Table 3.3: Initial Everett's Price Update Method

increase λ_r. If it is over-satisfied, decrease it, and if it is perfectly satisfied, do nothing. The only difference is that the amount of increase or decrease in λ_r has nothing to do with the magnitude of the infeasibility ($w^{j'_r}_{f'_r t} - w^{j_r}_{f_r, t-s_r}$) corresponding to row r, while with a subgradient update, this same λ adjustment is linear in the infeasibility. With Everett, we update λ_r *geometrically* according to a pre-determined stepsize which depends only on the history and sign of the infeasibility of row r. Namely, with $(w^{j'_r}_{f'_r t} - w^{j_r}_{f_r, t-s_r})$ as the r'th row of our relaxed constraint set $\hat{\mathcal{R}}$, at every iteration n, we generate Lagrange multipliers $\lambda_r(n)$, as well as update parameters $\epsilon_r(n)$ by the method of Table 3.3.

If $V(n)V(n-1) > 0$, the most recent two adjustments were in the same direction and ϵ is increased. If $V(n)V(n-1) < 0$, the directions are changed and ϵ is decreased. The rule is reminiscent of the *heavy ball method* of Poljak [12], but unlike that method, this one incorporates memory of the process into $\lambda(n+1)$ geometrically via ϵ, rather than arithmetically via the term $\lambda(n) - \lambda(n-1)$.

In any case, Table 3.3 is Everett's Method as presented in [65]. We made only two modifications. The first is that we prevent λ from ever falling below zero, as we are enforcing inequality, not equality constraints. The second modification involves a slight problem we had with prices hitting zero. Once they do in Everett's Method, they stay there. We therefore defined $\hat{\lambda} = \lambda + \gamma$, where γ is some positive constant, and applied the above updates to $\hat{\lambda}$, always keeping it from falling below γ. This translates to what we call the *Extended Everett's Method*, which is given in Table 3.4.

As a second interesting empirical observation, we never saw any evidence of a duality gap in the solution of our decomposed problem, although we have been unable to prove that our decomposition

Update V and ϵ as in Initial Everett Method				
$\lambda(n+1)$	:=	$(1+\epsilon(n))\lambda(n)+\gamma\epsilon(n)$	if	$V(n)>0$;
$\lambda(n+1)$	:=	$(1-\epsilon(n))\lambda(n)-\gamma\epsilon(n)$	if	$V(n)\leq 0$.

Table 3.4: Extended Everett's Price Update Method

would or should satisfy any type of strong duality.

3.6.4 Updated Branching Strategy

Everett's method introduces one important complication. Once we introduce large prices λ into our objective function, the feasible solutions generated by the branching strategy outlined in Section 3.6.2 need no longer tend toward optimal. Our branching strategy favors "high-priced" trains, and has them arrive at their destination as fast as possible. However, dual prices attached to origin variables on these same trains may make early departure from an origin very expensive, something which the branching rule does not consider.

We therefore introduce an updated branching strategy in Procedure 2, which is the one implemented in the computational experiments of LMPM and YDM of Section 3.7. This strategy not only accommodates the presence of prices attached to origin variables, but also objective functions which are nonlinear in arrival time. Note that just as forcing a column corresponding to a destination

1: *Selective Integrality Enforcement:* Enforce binary requirements not just on destination variables, but also origin variables having nonzero coefficients.
2: **for all** Binary Variables **do**
3: *Fixed Branching Order:* Assign its branching order based on the index determined in Section 3.6.2 if that variable is a destination variable and train costs are linear in arrival time, or the absolute value of that column's coefficient, with higher values preceding lower ones, otherwise.
4: **end for**
5: *Specified Branching Direction:* Finally, we set our preferred branch direction to "up" (choose 1 over 0) for all destination variables and "down" for all selected origin variables.

Procedure 2: Branching strategy executed in experiments of Section 3.7.

within a subproblem to be 1 is a "strong" branch, equally so is forcing a column corresponding to an origin to be zero.

In summary, our total decomposition and solution procedure is given by Procedure 3.

repeat
Decomposition: Decompose our network problem via Lagrange relaxation.
for all Node-specific subproblems **do**
Solve by exploiting its network structure and using the branching strategy of Procedure 1 or 2.
end for
Price Updates: Assess infeasibility of connection constraints and update prices according to Everett's Method.
until Find a feasible solution with small complementary slackness

Procedure 3: Summary of our final solution strategy for a network of IP models.

3.7 Computational Experiments with Applications

Thanks to each of the contributions described in Section 3.6, we have been able to solve some very large instances of each of our application models, LMPM, YSSM, and YDM. What we have done for each application is assemble a battery of instances of increasing difficulty and observe how our computation times vary. Our primary concern was to identify the conditions under which simply solving the IP models would or would not be feasible. We have not validated the results of the models with actual railroad data or planning decisions, and are therefore not prepared to make qualitative assessments about the generated schedules themselves.

For each application, holding the topology of the network fixed, there seem to be two principal determinants of the difficulty of an instance. The first, of course, is the number of trains. The second is the congestion of the schedule.

In conditions of heavy congestion, two things happen. One is that the initial solution generated by our branching strategy is less likely to be close to an optimal schedule. The second is that we are less able to *a priori* prune the set of time windows T_f^j, since trains will be expected to experience significant and unpredictable delays. Both these effects combine to make our models highly sensitive to scheduling congestion.

In fact, as we've said before, the conclusions we shall draw from all our experiments are that these models can be very effective where the conditions listed in Section 3.1 apply. Across heavily traveled networks with highly heterogeneous traffic and large variances in delays, at least where "on-line" solution times are important, we expect that none of our models should perform very well and recommend instead the more specialized and efficient heuristics of Chapter 4

3.7.1 Experimental Environment

In our experiments, results for LMPM and YDM are all given in terms of *networks* of interconnected lines or terminals, where each LMPM or YDM represents a sub-problem whose solutions are coor-

machine	model operating system	processor CPLEX version	cache	real memory
(a)	Sparc 20 Model 61 SunOS 4.1.4	60 MHz SuperSPARC 3.0	1 MB	64 MB
(b)	Sparc 20 Model 50 SunOS 5.5	50 MHz SuperSPARC 4.0	36 kB	96 MB
(c)	Sparc 10 Model 41 SunOS 4.1.4	40 MHz SuperSPARC 3.0	1 MB	32 MB

Table 3.5: Machines on which computational experiments were run

machine	multiplier
(a)	0.63 (±0.09)
(b)	1.00
(c)	0.41 (±0.04)

Table 3.6: These multipliers apply for our battery of applications only, and do not represent general performance comparisons between the three platforms.

dinated via the Extended Everett's price update rule as described in Section 3.6.3. Only for YSSM are computational results presented for a single node.

All our computational experiments were coded in C++ calling CPLEX callable libraries (v3.0 or 4.0), and done on one of three machines, which will henceforth be designated by the letters (a), (b), and (c). All CPU times listed in this chapter will have a corresponding machine designation. The three machines are described in Table 3.5 To compare the three machines, we ran several identical experiments on each one, and derived conversion multipliers for CPU times for our applications which will allow us to translate CPU times for each machine into CPU times for machine (b). These multipliers are presented in Table 3.6. With each experiment in this chapter, we report both native machine CPU times, as well as CPU times converted to machine (b). We include error ranges because, while the CPU ratio between machines (a) and (c) was very stable (at .65), the CPU ratio between machines (a) and (b) varied between 1.41 and 1.79.

3.7.2 Line Meet-Pass Model

We first attempted to solve a large series of line planing problems without the presence of terminals. The lines were connected to each other only at sets of "junctions," which could be crew-change points, terminals, or any boundary between two or more linehaul dispatcher regions. For our purposes, trains changed identity once crossing a junction, so that each train traversed only one line. Also, all of our

	Congest	Trns	Jnct	Lns	Sg/Line	Mtp/Line	Mem	S CPU	P CPU
A	loose	240	5	4	11	4	9 MB	0:00:59 (b)	0:00:24 (b)
B	medium	240	5	4	11	4	9 MB	0:02:14 (b)	0:00:55 (b)
C	tight	240	5	4	11	4	9 MB	0:04:56 (b)	0:01:49 (b)
D	tight	240	5	4	11	4	9 MB	0:05:07 (b)	0:01:52 (b)
E	V tight	240	5	4	11	4	10 MB	0:08:21 (b)	0:04:18 (b)
F	V tight	240	5	4	35	16	18 MB	1:03:35 (b)	0:37:26 (b)
G	V tight	480	9	8	35	16	26 MB	2:16:49 (b)	0:46:39 (b)
H	loose	960	17	16	35	16	22 MB	0:09:51 (a)	0:00:51 (a)
I	medium	960	17	16	35	16	40 MB	0:22:20 (a)	0:02:06 (a)
J	V tight	960	17	16	35	16	43 MB	4:34:12 (b)	0:47:08 (b)

Table 3.7: LMPM Computational Experiments

Experiment	S CPU	P CPU
A	0:00:59	0:00:24
B	0:02:14	0:00:55
C	0:04:56	0:01:49
D	0:05:07	0:01:52
E	0:08:21	0:04:18
F	1:03:35	0:37:26
G	2:16:49	0:46:39
H	0:06:01	0:00:32
I	0:14:04	0:00:79
J	4:34:12	0:47:08

Table 3.8: LMPM CPU Times as Machine (b) times

lines were assumed to satisfy the single signal-block per segment assumption (Section 3.3.3), and arrival costs were linear in arrival time.

Our experimental planning scenarios were of three types: those involving 4 lines and 240 trains, 8 lines and 480 trains, and 16 lines and 960 trains. In all cases, each line had 60 trains traversing it over a period of 37, 25, 13, and 5 hours, corresponding to scenario schedule congestion which we call Loose, Medium, Tight, and Very Tight, respectively. We also repeated the 240 train, 11 segment per line, Tight congestion scenario once with a different set of train priorities, and found that they made little difference.

As for performance statistics, we report three in Table 3.7. Memory usage in column 7 is given as the maximum of our application's resident set size and size in virtual memory during the course of execution. Serial CPU in column 8 is a recording of the total CPU time spent solving integer programs (i.e. the total time calling CPLEX solvers). Parallel CPU in column 9 indicates what total

CPLEX CPU time would have been had we had a separate processor for each sub-problem. Namely, at each iteration of Everett's Method, Serial CPU is incremented by the total solution time of all sub-problems, and Parallel CPU is incremented by the maximum solution time of all sub-problems. We felt it important to report this last statistic since, as we said before, one of our motivations in choosing a Lagrange decomposition was to exploit parallelization in the solution of our problem. Finally, we do not include time on overhead between solves or set-up time before our first solve since we have made no effort to optimize this part of our code. Nevertheless, we should say that it is typically of a similar order of magnitude as the solution of our IP's. In Table 3.8, we report all CPU times converted to times for machine (b) using the conversion factors of Table 3.6.

In any case, the first thing we notice about our results is that computation times, holding the number of trains and segments constant, are dramatically correlated with schedule congestion. This is not surprising, as it is consistent with our claim in Section 3.6.1 that capacity constraints are what make this scheduling problem hard. At the extremes of scheduling congestion. Very Tight scenarios correspond to trains departing from their origin junctions once every fifteen minutes, which is unrealistically high congestion in all but the busiest corridors. Nevertheless, we chose to emphasize these scenarios as we wanted to see how our formulation reacted to conditions of such extreme congestion.

The second important thing to point out is that even with close to 1000 trains, and under conditions of extreme congestion, Everett's Method managed to converge to a provable optimal solution in 4.5 hours CPU time, and that number could be reduced to less than one hour if one were to have parallel machines or processors assigned to each individual line. We feel this performance would be consistent with the needs of an off-line scheduling tool, while not of a model intended either to assist in real-time planning, or to be used in an interactive, iterative schedule building paradigm with human planners.

As for Everett's Method, we observed some interesting behavior, which we describe in greater detail in Section 3.7.5. What we saw was that Everett's Method finds a feasible global solution very quickly (10-20% of total computation time), finds the optimal solution moderately quickly (40-70% of total computation time), and spends the rest of its time simply confirming the optimality of its best solution by finding optimal dual prices and driving complementary slackness to zero. As we've said before, we never saw any evidence of a duality gap in any of our experiments. For this reason, we state here a point which will be made again in Section 3.7.5, that a heuristic implementation

Scenar	congest	InTrns	OutTrns	Blocks	Memory	CPU
A	loose	10	10	40	6.5 MB	0:09:09 (c)
B	medium	10	10	40	8 MB	0:17:42 (b)
C	tight	10	10	40	16 MB	0:33:43 (b)
D	tight	10	10	40	20 MB	7:02:29 (a)
E	loose	15	15	60	6.5 MB	0:20:41 (a)
F	medium	15	15	60	7.5 MB	1:45:15 (a)
G	tight	15	15	60	20 MB	> 75 hrs (a)
H	loose	20	20	80	13 MB	1:29:21 (c)
I	medium	20	20	80	15 MB	2:08:31 (c)
J	tight	20	20	80	40 MB	> 40 hrs (a)

Table 3.9: YSSM Computational Experiments

of Everett's Method where we stop computation some time after our current best-feasible solution hasn't improved would certainly be a sensible implementation option, and could have reduced the execution times in Table 3.8 considerably.

3.7.3 Yard Switching And Sequencing Model

For YSSMs, we did not create networks of terminals, but rather solved single Yard Switching and Sequencing Problems individually. There was therefore no decomposition and no Everett's Method to consider. We created scenarios by generating a list of inbound trains and outbound trains, as well as a set of cuts, and randomly distribute our cuts among the available trains to create a car-to-train assignment. We then took this list and generated appropriate precedence relationships between inbound trains, outbound trains, and the different blocks. Just as in the case of LMPM, we were primarily interested in observing our two key performance measures, time and memory, and how they acted as a function of the dimensions of our problem. We report in Table 3.9 results for different planning scenarios, where, for the same classification terminal, we vary the number of inbound trains, outbound trains, and classification blocks. For each scenario, we report total MIP computation time, as well as memory usage. In Table 3.10, we report all CPU times converted into times for machine (b), using the conversion ratios of Table 3.6

For YSSM, we divided our experiments into three congestion classes. Tight, Medium, and Loose congestion scenarios corresponded to train arrivals at the terminal that were spaced 45 minutes, 1 hour, and 1.5 hours apart, respectively. We discretized time into intervals of 15 minutes. Here, we see that the effect of schedule congestion is even more important than in LMPM, and even more

Experiment	S CPU
A	0:03:45
B	0:17:42
C	0:33:43
D	2:47:41
E	0:13:02
F	1:06:18
G	> 47 hrs
H	0:36:38
I	0:52:42
J	> 25 hrs

Table 3.10: YSSM CPU Times as Machine (b) times

Scenar	Trns	Yrds	Lns	Memory	S CPU	P CPU
A-tight	16	4	4	3.3 MB	0:05:29 (a)	0:01:55 (a)
B-loose	80	4	4	4.6 MB	0:08:58 (a)	0:02:53 (a)
C-tight	80	4	4	9.4 MB	0:26:49 (a)	0:10:50 (a)
D-loose	240	9	12	8 MB	0:13:19 (a)	0:05:07 (a)
E-med	240	9	12	18 MB	0:53:18 (a)	0:32:20 (a)
F-tight	240	9	12	32 MB	5:12:41 (a)	3:01:55 (a)
G-loose	800	24	40	16 MB	0:36:55 (a)	0:05:57 (a)
H-med	800	24	40	45 MB	6:07:38 (b)	1:50:57 (b)
I-tight	800	24	40	50 MB	7:57:28 (b)	1:15:47 (b)
J-Vtight	800	24	40	62 MB	— (b)	— (b)

Table 3.11: YDM Computational Experiments

important than the number of trains. In fact, two of our tight congestion scenarios were not solved within any reasonable amount of time.

3.7.4 Yard Dispatching Model

Our computational experiments with a network of YDM's were done in much the same way as with LMPM. Our network was constructed by utilizing formulations of lines as simple delay generators. We record in Table 3.11 the same performance measures as before, namely Memory, Serial CPU time, and Parallel CPU time. Again, we identify our scenarios by number of trains, number of terminals, and congestion of scheduling scenario.

Computational results with our network of terminals are very similar to those with LMPM described earlier. Just as before, we see execution times dependent on the degree of congestion in our scenarios. Also just as before we have execution times for realistically sized problems (24

Experiment	S CPU	P CPU
A	0:03:27	0:01:12
B	0:05:39	0:01:49
C	0:16:54	0:06:24
D	0:08:23	0:03:13
E	0:33:35	0:20:22
F	0:23:15	1:54:36
G	3:17:00	0:03:45
H	6:07:38	1:50:57
I	7:57:28	1:15:47
J	—	—

Table 3.12: YDM CPU Times as Machine (b) times

terminals, 800 trains) being solved within half of one day serially and in under two hours with parallel processors. Unfortunately, under very tight congestion, our largest problem became unsolvable.

One important difference between our network of YDM's and our previous network of LMPM's is that execution times for individual subproblems fluctuated much more here than before. In fact, in the case of our largest planing scenarios, subproblem solution times could be anything from 3 seconds to 2000 seconds. The reason for this is that time windows are much larger in our modeling of terminals than they had to be in our modeling of lines. The standard deviation of the time it takes a car to travel through any classification terminal can be as high as 10 hours [53], while line traversal times are much more predictable. Larger time windows, of course, expand the space of feasible solutions for every sub-problem and make solving it much more difficult.

3.7.5 Progress of Everett's Method

The final point which should be made comes from examination of the progress of Everett's Method itself. In Figures 3.7.5, we plot a history of the infeasibility, over-satisfiability, objective function, and complementary slackness (dual price multiplied by infeasibility) of the current solution as a function of iteration of Everett's Method. The chart in Figure 3.7.5 is for YDM Scenario C, but all others demonstrated the same qualitative behavior, both in YDM and LMPM. We notice two important points. The first is that we very quickly get a feasible solution (by Iteration 3), and also fairly quickly found a reasonably good feasible schedule (Iteration 12). For this reason, in very little time (one sixth of total CPU time), a user of this model could at least get a decent feasible schedule. The second point is that Everett's Method found the optimal schedule by Iteration 50, and took its

remaining 25 iterations merely to prove the optimality of that schedule (by setting prices to drive complementary slackness to zero). A heuristic stopping rule that specified stopping if, after some number of iterations, the current best feasible schedule is not improved would do quite well at finding good schedules, as well as cut computation times by 20 - 40%.

3.7.6 Speeding up the solution

Before concluding, we present two additional approaches for speeding the solutions of our IP models, besides the relaxed Everett procedure mentioned in Section 3.7.5. We notice that our branch and bound procedure often finds the optimal integer solution quickly. In fact, in all our examples, it was within the first three integer solutions found, although we typically spent some time after finding this solution to fathom the rest of the tree and guarantee optimality. We therefore considered eliminating this step. In fact, we stop our branching procedure as soon as we have found a feasible integer solution whose value is within some large fraction (in our case .5) of the value of our best bound so far. We also modified the backtracking tolerance within CPLEX,[2] raising it to 2.0, which makes it less likely that we will backtrack at un-fathomed nodes and makes our search more depth-first rather than breadth-first.

With this stopping rule, we repeated all the experiments of Section 3.7.3, and present the results in Table 3.13. CPU times translated to Machine (b) times are given in Table 3.14. There were two instances (C and D) where computation times appeared to increase. This is because the different experiments were performed on different machines, and the reader should interpret those results to mean that normal B&B and simplified B&B performed comparably. In most cases execution times under the heuristic were one quarter to one fifth those when optimality is guaranteed, with little sacrifice in solution quality. In one case (F), the performance improvement was very good.

Another way of speeding up the solution is to have the heuristic of Table 3.15 generate the initial feasible integer solution. This would generate a useful upper bound on the value of the objective function very quickly which might help guide a backtracking strategy to find a superior solution. This last idea was not implemented during the experimentation above

[2]The CPLEX backtracking tolerance is the amount of relative degradation in objective function value between any node and its parent tolerated before backtracking up the search tree. [28]

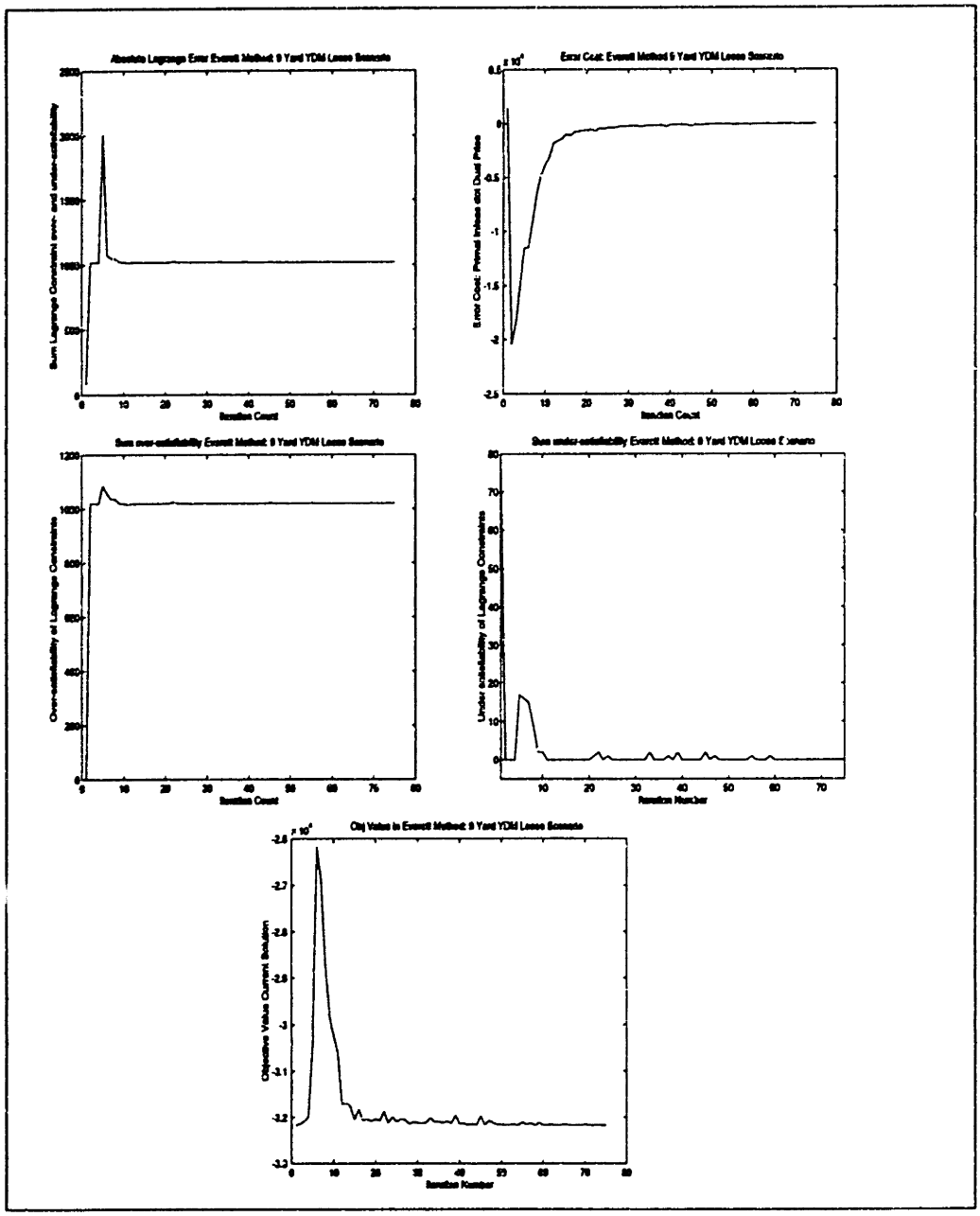


Figure 3-6: Notice that by iteration 3, the method finds a feasible solution, and by iteration 12, it finds a schedule with cost not far from the optimum. Notice also that the optimal schedule is found by iteration 50, but that 25 iterations remain before its optimality is established.

Scenar	congest	InTrns	OutTrns	Blocks	CPU	Opt Gap
A	loose	10	10	40	0:03:30 (c)	0%
B	medium	10	10	40	0:13:31 (c)	0%
C	tight	10	10	40	0:33:45 (b)	0%
D	tight	10	10	40	4:42:07 (a)	0.93%
E	loose	15	15	60	0:05:04 (a)	0%
F	medium	15	15	60	0:14:32 (a)	0%
G	tight	15	15	60	8:11:01 (a)	NA
H	loose	20	20	80	0:24:21 (c)	0%
I	medium	20	20	80	1:49:42 (c)	0.94%
J	tight	20	20	80	> 25 hrs (b)	NA

Table 3.13: YSSM Computational Experiments–Simplified IP and Branching Rule

Experiment	CPU	CPU:Simplified	Opt Gap
A	0:03:45	0:01:26	0%
B	0:17:42	0:05:32	0%
C	0:33:43	0:33:45	0%
D	2:47:41	2:57:44	0.93%
E	0:13:02	0:03:12	0%
F	1:06:18	0:09:09	0%
G	> 47 hrs	5:09:20	NA
H	0:36:38	0:09:59	0%
I	0:52:42	0:44:59	0.94%
J	> 25 hrs	> 25 hrs	NA

Table 3.14: Simplified YSSM CPU vs. Standard YSSM, Machine (b) times

1. Let $\mathcal{U}$ be set of unscheduled trains, and $\mathcal{U}_0 = \mathcal{F}$. For each train f, let v_f by the sum of the costs of all train with which it has a precedence relationship,
$$v_f = \sum_{f':\exists((f,j),(f',j'),s)\in\mathcal{R}} c_{f'}$$
and create index $I_f = \alpha c_f + (1-\alpha)v_f$, where α is as in Procedure 1.
2. Within $\mathcal{U}$, order trains by increasing I_f. Let $\tau = 0$. Set $C_j(t)^0 = C_j(t)$. At each time τ, do:
3. Choose train f_τ in $\mathcal{U}_\tau$ with largest I_f. Send it through network as fast as possible given capacity constraints, ignoring other trains.
4. For each j in $P(f_\tau,\cdot)$ and each t such that $w^j_{f_\tau,t} - w^{j'}_{f_\tau,t} = 1$, where for some i, $j = P(f_\tau,i), j' = P(f_\tau,i+1)$,
$$C_j(t)^{\tau+1} \longleftarrow C_j(t)^\tau - 1.$$
5. Set $\mathcal{U}_{\tau+1} \longleftarrow \mathcal{U}_\tau \backslash f_\tau$, $\tau \longleftarrow \tau + 1$, and goto step 3 if $\mathcal{U}_\tau \neq \emptyset$; stop otherwise.

Table 3.15: Feasible Schedule Heuristic

3.8 Summary and Conclusions

We presented three IP models of instances of MNSP* for railroad scheduling: LMPM, YSSM, YDM. We derived each from a core model, GBS, which was motivated by the work of Bertsimas and Stock [15] on the GHP. We saw that GBS possesses four important properties which made it an attractive generic model for us, and which we reviewed in this chapter.

First, the LP relaxation of GBS often yields a strong approximation to the underlying IP. Second, the BS Polyhedron, the polyhedron defined by the LP relaxation of GBS, contains an underlying network structure which can be exploited to yield better computational performance. Moreover, when solving the full IP GBS, we empirically observe a certain branching rule to be effective at finding integer optimal solutions, even though this branching strategy involves the relaxation of all but a carefully chosen minority of binary requirements. This observation surprised us. Finally, GBS exhibits a natural decomposability, and in two cases, LMPM and YDM, we utilized a Lagrange decomposition with novel heuristic price-update rule to decompose a large problem into ind endent sub-problems, solving instances of these problems which are much larger than what we believe might have been possible before. These relaxations also empirically appear to yield duals with no duality gap, which was another surprising observation.

Attracted by all these properties, we performed extensive computational experiments on IP models of LMPM, YSSM, and YDM which demonstrate that these models have promising potential under conditions of homogeneous line topology, low to moderate traffic intensity, and low to moderate expected delays. Where these conditions do not hold, these IP's still might have a role to play in off-line, batch scheduling that requires little human interaction and feedback, and where computational resources are significant. Where none of these conditions is satisfied, such as on very heavily traveled corridors, or after unanticipated disasters such as the Mississippi flooding of 1993, we expect the heuristics of Chapter 4 and other more specialized scheduling approaches to be more worthwhile.

Chapter 4

Heuristic Scheduling

4.1 Introduction and Motivation

We now consider an alternative approach to solving the same models presented in Chapter 3. Rather than solving integer programming models to optimality, we solve to "approximate" optimality dynamic programming formulations for the same discrete-time scheduling problems. In what follows, we use the term *approximate dynamic programming* (ADP) to refer to the approximate solution of a dynamic program by any means. Our ADP's work by coordinating a battery of scheduling heuristics. In a discrete-time environment, at each time step, the control decision is simply which heuristic control policy to apply for the duration of the time-step. We find in several experimental problems that we are able to generate schedules whose objective values are within a few percent of optimality, yet do so in just a few minutes, rather than the many hours required to solve our IP models.

4.1.1 Approximate DP with Heuristic Delimited Control Space

In this section, we introduce the general concepts involved with our solution approach. We utilize MNSP*, the generic network scheduling problem introduced in Section 1.3. In this chapter, we formulate MNSP* as a DP, where our state i is defined by:

1. the location of every job in our network, denoted $S_f(i)$,

2. the current element of every job in our network, denoted by $K_f(i)$. K and S are related by $S_f(i) = P(f, K_f(i))$, which says that segment $S_f(i)$ is the $K_f(i)$'th segment in f's path.

3. $a_{f,j}(i)$, which is the time each job f arrived at each processor j in its itinerary, where $j \leq K_f(i)$, and

4. the current time, denoted $t(i)$.

At any state, our control space comprises all the ways we could dispatch trains over the next discrete time-period. To facilitate the discussion, we formally introduce some standard notation:

V: set of all states i in our formulation

S: set of all states $i \in V$ such that all jobs are at their destinations. We call such states *leaf* states.

τ: size, in minutes, of our time discretization.

$U(i)$: set of feasible controls at state i.

$d(i,u)$: state $i' \in V$ which would follow state i if control $u \in U(i)$ is applied. We call state i' a *child* state of i.

$\mu : V \mapsto U$: a particular policy. For each state $i \in V$, designates control $\mu(i) \in U(i)$

$g(i,i')$: transition cost from state i to state i', where i' is such that there exists $u \in U(i) : d(i,u) = i'$. Defined to be the cost of arrival for any jobs that arrive at their destination (or complete all their processing) in the transition from i to i'.

$J^*(i)$: Optimal cost-to-go from state $i \in V$, given by the unique solution to *Bellman's Equation:*

$$\begin{aligned} J^*(i) &= 0 \ \ \forall i \in S \\ J^*(i) &= \min_{u \in U(i)} \{g(i, d(i,u)) + J^*(d(i,u))\} \ \ \forall i \notin S \end{aligned} \tag{4.1}$$

$\hat{J}(i)$: Some approximation to the optimal cost-to-go function, $\hat{J}(i) \approx J^*(i)$.

$J_\mu(i)$: Cost-to-go from state $i \in V$ under policy μ, given by the unique solution to

$$\begin{aligned} J_\mu(i) &= 0 \ \ \forall i \in S \\ J_\mu(i) &= g(i, d(i, \mu(i))) + J_\mu(d(i, \mu(i))) \ \ \forall i \notin S \end{aligned} \tag{4.2}$$

$\hat{U}(i)$: Some "representative" subset of $U(i)$.

Knowing J^*, one can solve DP by solving the minimization problem in Bellman's Equation at each state. Unfortunately, it is often the case in DP formulations of realistic problems that both the state space V and the control space $U(i)$, $i \in V$, are extremely large, so that it is neither feasible to calculate J^*, nor to evaluate the minimization in (4.1). The key to a successful ADP, therefore, is to set $\hat{J}$ and $\hat{U}$ in such a way that the new DP is both tractable and still relevant to the original problem.

Our approach is the following. For every state, we calculate a set of p representative *features*. For example, a feature of a state could be the number of jobs which have completed all their processing, or have completed it on time, or the congestion around certain bottleneck processors. We use these features to approximate J^*, or, perhaps, to approximate J_μ for enough relevant policies μ. We therefore define

$F(i)$: the *feature extraction function* which extracts the vector of relevant features from a state i. $F : i \mapsto \Re^p$.

$G_w(F)$: the parameterized function of a feature vector we use to approximate J_μ, i.e. $G : \Re^p \mapsto \Re$ and

$$\hat{J}(i) \equiv G_w(F(i))$$

for some appropriate vector of parameters $w \in \Re^n$.

$G(\cdot)$ could be a polynomial function of F, a neural network, a spline, or any other approximation function.

Of course, utilizing an approximation scheme such as a neural network to estimate J^* and make control decisions accordingly is nothing new, and is the foundation for what is known as *Neuro-Dynamic Programming* [13]. As for forming our representative control sets $\hat{U}(\cdot)$, however, we employ a method which is a bit more novel. We accumulate a battery H of heuristic job sequencing policies $h \in H$. At each state $i \in V$, we consider the $|H|$ controls which correspond to exactly following each sequencing heuristic $h \in H$ for the next τ minutes. This is how we form $\hat{U}(i)$. Thus, at each state $i \in V$, the approximate form of the minimization in Bellman's Equation (4.1) reduces to the steps of Procedure 4.

repeat
  designate current state as i
  for all heuristics $h \in H$ **do** {Generate Children}
    Simulate system for next τ minutes as if *every* dispatcher were to make decisions according to h.
    This creates $|H|$ *child states*, each denoted $d(i,h)$.
  end for
  for all $d(i,h)$ **do** {Evaluate Children}
    Evaluate $I_{d(i,h)} = g(i, d(i,h)) + G_w(d(i,h))$
  end for
  $i \leftarrow \arg\min_{d(i,h)} I_{d(i,h)}$
until $i \in S$ {Stopping Criterion}

Procedure 4: Approximate Bellman Iteration

In addition to this method, we can also think of an *Extended ADP*, which is defined by running Procedure 4, then each heuristic control individually, and finally choosing the best solution. Moreover, we can also perform what we call κ-ADP, which involves storing any child state whose estimated cost-to-go is within κ (in a relative sense) of the best child, and performing Procedure 4 on that child as well. Clearly, as $\kappa \rightarrow \infty$, κ-ADP will involve greater and greater enumeration of the solution space, so κ-ADP $\rightarrow$ DP.

4.1.2 Chapter Outline

Given this general framework, the issues we must address when we wish to specifically implement this in the domain of MNSP* are:

1. *Heuristics:* How shall we populate H? We present a battery of heuristics in Section 4.2 for MNSP*, then discuss application of these heuristics to the two "node" problems of interest in this thesis: linehaul dispatching and terminal switching and sequencing.

2. *Approximation Architecture:* In Section 4.3, we discuss our choice of features F for evaluating states in MNSP*, then in both the linehaul dispatching and yard switching and sequencing problems, and present two candidate parameterized evaluation functions $G_w(F)$.

3. *Training Approximation Architecture:* Given a feature extraction map F and parameterized evaluation function $G_w(F)$, there remains the problem of optimizing over parameter space $w \in \Re^n$. This is discussed in Section 4.4.

4. *Implementation Issues and Extensions:* In Section 4.5, we briefly outline some of the practical issues faced when executing the ADP described below, from numerical instability and data pruning, to expanding our solution procedure to encompass an entire network rather than a single planning node.

We follow the discussion of this chapter with a presentation in Chapter 5 of a case-study application of these methods to YSSM.

4.2 Heuristic Scheduling Methods

We now discuss common heuristic control policies MNSP*. First, however, we present some notation which will be useful to us in the following discussion.

Key Notation and Ideas for Heuristics

All Data for MNSP* (see Section 1.3).

"event $e = (f, j)$:" The arrival of a particular job f at a particular processor j in it's path.

"cost-event:" an event whose "time of completion" is an argument to our objective function. In particular, if $e = (f, j)$, $j = P(f, N_f)$, and $c_f \neq 0$, then we say e is a "cost event." Note that for notational simplicity, we will assume $c_{f,t}$ is such that $c_{f,t+1} - c_{f,t} = c_f \ \forall t \geq d_f$ (see Section 3.2).

"precedent event:" The first event $e_r = (f_r, j_r)$ in any precedence relationship r.

"dependent event:" The second event $e'_r = (f'_r, j'_r)$ in any precedence relationship r.

$tr_f(i)$: The remaining processing time for job f at processor $S_f(i)$. Equals $[l_{f,S_f(i)} + a_{f,S_f(i)}(i) - t(i)]^+$ if $K_f(i) > 1$, and $[r_f - t(i)]^+$ otherwise.

$str_j(i)$: The time remaining until processor j will be able to accommodate a new job. Equals 0 if the utilization of j is less than its capacity, and equals $\min_{f:j=S_f(i)} tr_f(i)$, otherwise.

$\mathcal{R}(i)$: The collection of *remaining* precedence relationships in state i. Namely, $\mathcal{R}(i)$ includes only those precedence relationships r in $\mathcal{R}$ whose precedent events $e_r = (f_r, j_r)$ have yet to be fulfilled.

$\hat{\mathcal{R}}(i)$: An "extension" of $\mathcal{R}(i)$, $\hat{\mathcal{R}}(i)$ is a set of precedence relationships $\hat{r}$ which includes all events having either direct precedence relationships $r \in \mathcal{R}(i)$, or "indirect" relationships generated transitively from the relationships in $\mathcal{R}(i)$. In other words, if there exist r_1 and r_2 in $\mathcal{R}(i)$ or $\hat{\mathcal{R}}(i)$ and $e'_{r_1} = e_{r_2}$, then $\hat{r} = (e_{r_1}, e'_{r_2}, s_{r_1}) \in \hat{\mathcal{R}}(i)$. Note, however, that the ordered pairs in $\hat{\mathcal{R}}(i)$ must always define a non-symmetric relationship among jobs. In models where there are circular precedence relationships (YSSM), special modifications to this definition apply (see Section 4.2.1).

$s_r(i)$ The "buffer remaining" for precedence relationship r in state i. Equals s_r if f_r has not begun processing at j_r in state i, and $[s_r + a_{f_r,j_r}(i) - t(i)]^+$ otherwise.

$\varepsilon_{f,j}$ A lower bound on the earliest possible time a job f can begin processing at a processor j in its itinerary. Calculated as the unique solution to

$$\varepsilon_{f,j} = \max\left\{\varepsilon_{f,\pi_f(j)} + l_{f,\pi_f(j)}, \max_{r\in\mathcal{R}(i):f'_r=f,j'_r=j}\{\varepsilon_{f_r,j_r} + s_r\}\right\} \tag{4.3}$$

$p_r(i)$: The total amount of "competing" processing still to be done with regards to precedence relationship $r = ((f_r, j_r), (f'_r, j'_r), s_r)$. It is defined as:

$$p_r(i) = \frac{\sum_{r_2\in\mathcal{R}(i):\ e'_{r_2}=e'_r, j_{r_2}=j_r} tr_{f_{r_2}}(i)}{\hat{C}_{j_r}(t(i))}, \tag{4.4}$$

where $\hat{C}_j(t)$ is an estimate of the time-average capacity of processor j around time t. For our purposes, it is simply a constant C_j, although could vary with time if $C_j(t)$ did.

$D_{f,j}(i)$: Given state i, an estimate of the time at which event $e = (f, j)$ must occur if job f is to complete processing by d_f, and all events dependent on e, either directly through some $r \in \mathcal{R}$ or indirectly through other precedence relationships $r \in \hat{\mathcal{R}}$, are to complete processing by their own due times $d_{j'_r}$. Calculated as the solution to:

$$\begin{aligned}
\hat{D}_{f,j}(i) &= \begin{cases} d_f & \forall j = P(f,N_f), c_f \neq 0 \\ \infty & \forall j = P(f,N_f), c_f = 0 \end{cases} \\
\hat{D}_{f,j}(i) &= \hat{D}_{f,\nu_f(j))} - l_{f,j} \quad \forall j = P(f,k), k \leq N_f - 1 \\
D_{f,j}(i) &= \min\left\{\hat{D}_{f,j};\ \min_{\forall r \in \mathcal{R}(i):\ e_r=(f,j)} \{D_{f'_r,j'_r}(i) - s_r - p(r)\}\right\}
\end{aligned} \tag{4.5}$$

The "estimation" in the above expression comes from the introduction of the term $p(r)$, which is added to account for other traffic at processor j.

$B_{f,j}(i)$: Given state i, an *upper bound* of the time at which event $e = (f,j)$ must occur if job f is to complete its processing by d_f, and all events dependent on e, either directly through some $r \in \mathcal{R}$ or indirectly through other precedence relationships $r \in \hat{\mathcal{R}}$, are to finish processing by their own due times $d_{j'_r}$. Calculated similarly to $D_{f,j}$, except that no attempt is made to account for competing traffic via $p(r)$:

$$\begin{aligned}
\hat{B}_{f,j}(i) &= \begin{cases} d_f & \forall j = P(f,N_f), c_f \neq 0 \\ \infty & \forall j = P(f,N_f), c_f = 0 \end{cases} \\
\hat{B}_{f,j}(i) &= \hat{B}_{f,\nu_f(j)} - l_{f,j} \quad \forall j = P(f,k), k \leq N_f - 1 \\
B_{f,j}(i) &= \min\left\{\hat{B}_{f,j};\ \min_{\forall r = \in \mathcal{R}(i):\ e_r=(f,j)} \{B_{f'_r,j'_r}(i) - s_r\}\right\}
\end{aligned} \tag{4.6}$$

$E_{f,j}(i)$: Given state i in which event $e = (f,j)$ has yet to occur, the earliest due date of all events depending on e, either directly or indirectly. If e has been realized in i, then $E_{f,j}(i)$ is ∞. It is calculated in a manner similar to $B_{f,j}$, except that processing times $l_{f,j}$ and buffer times s_r are not considered:

$$\begin{aligned}
\hat{E}_{f,j}(i) &= \begin{cases} d_f & \forall j = P(f,N_f), c_f \neq 0 \\ \infty & \forall j = P(f,N_f), c_f = 0 \end{cases} \\
\hat{E}_{f,j}(i) &= \hat{E}_{f,\nu_f(j)} j = P(f,k), k \leq N_f - 1 \\
E_{f,j}(i) &= \min\left\{\hat{E}_{f,j};\ \min_{\forall r \in \mathcal{R}(i):\ e_r=(f,j)} \{E_{f'_r,j'_r}(i)\}\right\}
\end{aligned} \tag{4.7}$$

$L_f(i)$: Expected Lateness for job f in state i, denoted

$$L_f(i) = t(i) + tr_f(i) - D_{f,\nu_f(j)}. \tag{4.8}$$

Slack is therefore $(-1)L_f(i)$, and tardiness is $[L_f(i)]^+$.

$P_{f,j}(i)$: The "Previous Processing Time" for event (f,j) records an estimation of the amount of processing that must be done before (f,j) can be attained. It is calculated as follows:

$$\hat{P}_{f,j}(i) = 0 \quad \forall j = P(f,k), k \leq K_f(i),$$

$$
\begin{aligned}
\hat{P}_{f,j}(i) &= \max\{tr_f(i), 1, str_j(i)\} \quad \forall j = P(f,k), k = K_f(i)+1, \\
\hat{P}_{f,j}(i) &= \hat{P}_{f,\pi_f(j)} + l_{f,\pi_f(j)} \quad \forall j = P(f,k), k > k_S, \\
\bar{P}_{f,j}(i) &= \sum_{\hat{r}\in\hat{\mathcal{R}}(i):e'_{\hat{r}}=(f,j)} \sum_{k=K_{f_{\hat{r}}}}^{k=\kappa_{f_{\hat{r}}}(j_{\hat{r}})-1} l_{f_{\hat{r}},P(f_{\hat{r}},k)} + s_{\hat{r}}(i) \\
P_{f,j}(i) &= \max\left\{ \hat{P}_{f,j}(i) + \bar{P}_{f,j}(i),\ 1 \right\} \quad \forall f,j
\end{aligned} \tag{4.9}
$$

$Q_{f,j}(i)$: The "Minimum Previous Processing Time" for event (f,j) records a *lower bound* on the amount of processing that must be done before (f,j) can be attained. It is calculated as follows:

$$
\begin{aligned}
\hat{Q}_{f,j}(i) &= 0 \quad \forall j = P(f,k), k \le K_f(i), \\
\hat{Q}_{f,j}(i) &= \max\{tr_f(i), 1, str_j(i)\} \quad \forall j = P(f,k), k = K_f(i)+1, \\
\hat{Q}_{f,j}(i) &= \hat{Q}_{f,\pi_f(j)} + l_{f,\pi_f(j)} \quad \forall j = P(f,k), k > k_S, \\
\bar{Q}_{f,j}(i) &= \max_{\hat{r}\in\hat{\mathcal{R}}(i):e'_{\hat{r}}=(f,j)} \left\{ \sum_{k=K_{f_{\hat{r}}}}^{k=\kappa_{f_{\hat{r}}}(j_{\hat{r}})-1} l_{f_{\hat{r}},P(f_{\hat{r}},k)} + s_{\hat{r}}(i) \right\} \\
Q_{f,j}(i) &= \max\left\{ \hat{P}_{f,j}(i),\ \bar{P}_{f,j}(i),\ 1 \right\} \quad \forall f,j
\end{aligned} \tag{4.10}
$$

Because the processing times of precedent events are added as if only one job could be processed in the network at a time, $P_{f,j}(i)$ is an upper bound on the time before event (f,j) can be attained.

$R_{f,j}(i)$: The "Remaining Process Time" for event (f,j) records an estimate on the amount of time after the attainment of event $e=(f,j)$ that a "cost event" might occur. It is calculated by:

$$
\begin{aligned}
\hat{R}_{f,j}(i) &= \begin{cases} 0 & \forall j = P(f,N_f), c_f \neq 0 \\ \infty & \forall j = P(f,N_f), c_f = 0 \end{cases} \\
\hat{R}_{f,j}(i) &= \hat{R}_{f,\nu_f(j)} + P_{f,\nu_f(j)} - P_{f,j} \quad \forall j = P(f,k), k < N_f \\
\bar{R}_{f,j}(i) &= \min_{r\in\mathcal{R}(i):\ e_r=(f,j)} \left\{ R_{f'_r,j'_r}(i) + P_{f'_r,j'_r}(i) - P_{f,j}(i) \right\} \quad \forall j \\
R_{f,j}(i) &= \min\left\{ \hat{R}_{f,j}(i), \bar{R}_{i,j}(i) \right\} \quad \forall f,j
\end{aligned} \tag{4.11}
$$

In what follows, we interpret $R_{f,j}(i) = \infty$ to mean that this quantity is undefined for event (i,j). We also add the correction term $P_{f'_r,j'_r} - P_{f,j}$ to $R_{f'_r,j'_r}(i)$ above to account for the fact that event $e'_r = (f'_r, j'_r)$ may not be ready to start immediately after the realization of event (f,j). Note that this correction term may not be negative, and in fact is at least s_r (see Equation 4.9).

$T_{f,j}(i)$: The "Minimum Remaining Process Time" for event (f,j) records a *lower bound* on the amount of time after the attainment of event $e=(f,j)$ that a "cost event" might occur. It is calculated identically to $R_{f,j}(i)$, except that Q is substituted for P.

$$
\hat{T}_{f,j}(i) = \begin{cases} 0 & \forall j = P(f,N_f), c_f \neq 0 \\ \infty & \forall j = P(f,N_f), c_f = 0 \end{cases}
$$

$$\begin{aligned}
\hat{T}_{f,j}(i) &= \hat{T}_{f,\nu_f(j)}(i) + Q_{f,\nu_f(j)} - Q_{f,j} \quad \forall j = P(f,k), k < N_f \\
\bar{T}_{f,j}(i) &= \min_{r \in \mathcal{R}(i):\ e_r=(f,j)} \left\{ T_{f'_r,j'_r}(i) + Q_{f'_r,j'_r}(i) - Q_{f,j}(i) \right\} \quad \forall j \\
T_{f,j}(i) &= \min\left\{ \hat{T}_{f,j}(i), \bar{T}_{i,j}(i) \right\} \quad \forall f,j
\end{aligned} \tag{4.12}$$

σ_f : If c_f is the cost coefficient of job f, we define σ_f, the *surrogate cost* of job f, as the unique solution to:

$$\sigma_f(i) = c_f + \sum_{\hat{r} \in \mathcal{R}: f_r = f} \sigma_{f'} \tag{4.13}$$

Of course, where job cost-coefficients are time dependent, σ_f must become a function of i and c_f replaced by something such as $c_{f,t(i)+v_f(i)}$, where $v_f(i)$ is some estimate of the minimum or expected time till a train's arrival at its destination. One logical estimate is

$$v_f(i) = d_f + L_f(i) - t(i).$$

4.2.1 Scheduling Heuristics

We present our heuristics in two stages. First, we present heuristics appropriate for MNSP*, which could model either a linehaul dispatcher region or terminal. Then, we follow the discussion of our generic heuristics with specific comments about issues involved with the implementation of each of them in the cases of both lines and terminals.

Generic Processor Network Heuristics (MNSP*)

We identify eighteen different scheduling heuristics for MNSP*.

1. *FIFO:* The first job to reach any processor begins service, taking priority over the other jobs. In other words, we simply order our jobs according to the index:

$$I_f^{FIFO}(i) = a_{f,S_f(i)}(i). \tag{4.14}$$

Ties are broken by giving preference to the job with the lowest ID number. In the presence of uniform job sizes, this heuristic attempts to minimize *resource* idle time. In the context of our rail applications, we might think of this as maximizing the utilization of track resources.

2. *mFIFO:* This is the same as FIFO, except ties are resolved by giving priority to the job with the higher ID Number. This heuristic is what we call a "noise heuristic," and is added to increase the number of different controls to which Procedure 4 will have access.

3. *LIFO* This heuristic is the opposite of FIFO, and takes jobs in the opposite order in which they arrived. In the case of ties, it is also the opposite of FIFO, taking jobs in the order of highest ID number first. This is also a "noise heuristic."

$$I_f^{LIFO}(i) = (-1)a_{f,S_f(i)}(i). \tag{4.15}$$

4. *SPP (Static Priority Pass):* Where jobs conflict, give priority to the one with the higher static priority. In the context of rail scheduling, this means that bulk and local trains always wait for general merchandise trains, which in turn always wait for intermodal traffic. Where priorities tie, apply FIFO.

$$I_f^{SPP}(i) = c_f. \tag{4.16}$$

5. *MD, WMD (Min Delay and Weighted Min Delay):* Here, we attempt to minimize delay, or job "idle time," but consider explicitly job sizes, and, in the case of WMD, priority-based weights of job sizes. For example, we may wish to make an early, large job wait for a later, small one. Where priorities differ, we may even wish to make an early, *small* job wait for a later, larger one with higher priority.

   Assume at some processor j, we have a set Γ of jobs competing to enter j. For each job $f \in \Gamma$, calculate the index I_f^{MD} as follows:

$$\hat{I}_f = l_{f,j} + \max\{\, \varepsilon_{f,j} - t(i),\ T_{f,j}(i),\ tr_f(i) \,\} \tag{4.17}$$

$$\bar{I}_{f'} = \max\{\, \varepsilon_{f',j} - t(i),\ T_{f',j}(i),\ tr_{f'}(i) \,\} \tag{4.18}$$

$$I_f^{MD} = \sum_{f' \in \Gamma \setminus f} \left[\hat{I}_f - \bar{I}_{f'}\right]^+ \tag{4.19}$$

   Now, to add weights to this index, a logical choice might be:

$$I_f^{WMD} = \sum_{f' \in \Gamma \setminus f} \sigma_{f'} \left[\hat{I}_f - \bar{I}_{f'}\right]^+ \tag{4.20}$$

   MD and WMD then assigns priority to each job according to I_f^{MD} and I_f^{WMD}, respectively.

   Note that our first three heuristics do not involve any sense of arrival targets, which is appropriate in many practical applications of MNSP*, such as LMPM with little scheduled traffic, for example.

6. *MS, WMS (Min Slack, Weighted Min Slack):* Simply give priority to the job which has accumulated the most delay already in its schedule, which, for job f in state i competing for processor j, is calculated as:

$$I_f^{MS} = D_{f,j}(i) - t(i) \tag{4.21}$$

   In LMPM, under uniform train priorities, this is similar to the decision rule utilized by PLATO [32], an experimental Line Planning Algorithm, except that PLATO calculates slack under the assumption that each train could travel at unimpeded full speed through the rest of its itinerary to its destination. In our setting, this means that PLATO would use $B_{f,j}(i)$ in place of $D_{f,j}(i)$ above. In YSSM, also under uniform train priorities and where block sequences in outbound trains are completely specified and all inbound trains have identical arrival times, this would correspond to the scheduling logic described by Armacost [4].

   WMS computes the same slack index, then incorporates train priorities via:

$$I_f^{WMS} = [\text{slack}_f(i)]^+ / \sigma_f(i) + [\text{slack}_f(i)]^- \cdot \sigma_f(i) \tag{4.22}$$

   where the operators $[\cdot]^+$ and $[\cdot]^-$ return the positive and negative parts of their arguments.

7. *MT, WMT (Min Imposed Tardiness, Min Imposed Weighted Tardiness):* At each conflict, estimate the minimum extra tardiness that would result by giving each job priority over others, assuming jobs are subsequently processed without delay. Then give priority to jobs which impose the minimum above cost. In this sense, we make decisions based on a *lowest minimum bound criterion.*

   If, in some state i, we have a set of jobs Γ competing for a processor j, what is the imposed tardiness-cost on the system of giving priority to job $f \in \Gamma$? One estimate can be expressed as:

$$\hat{I}_f = l_{f,j} + \max\{\ \varepsilon_{f,j} - t(i),\ T_{f,j}(i),\ tr_f(i)\ \} \tag{4.23}$$

$$\bar{I}_{f'} = \max\{\ \varepsilon_{f',j} - t(i),\ T_{f',j}(i), tr_{f'}(i)\ \} \tag{4.24}$$

$$I_f^{MT} = \sum_{f' \in \Gamma \setminus f} \left[[\hat{I}_f - \bar{I}_{f'}]^+ - B_{f',j}(i) + t(i) \right]^+ \tag{4.25}$$

   Adding weights, this becomes:

$$\hat{I}_f = l_{f,j} + \max\{\ \varepsilon_{f,j} - t(i),\ T_{f,j}(i),\ tr_f(i)\ \} \tag{4.26}$$

$$\bar{I}_{f'} = \max\{\ \varepsilon_{f',j} - t(i),\ T_{f',j}(i), tr_{f'}(i)\ \} \tag{4.27}$$

$$I_f^{WMT} = \sum_{f' \in \Gamma \setminus f} \sigma_{f'} \left[[\hat{I}_f - \bar{I}_{f'}]^+ - B_{f',j}(i) + t(i) \right]^+ \tag{4.28}$$

8. *MET (Min Imposed Expected Tardiness) and WMET (Min Imposed Weighted Expected Tardiness):* These heuristics work just as MT and WMT, except that instead of using full, unimpeded speed to calculate tardiness, we use an estimate of tardiness which takes into account congestion likely to be encountered by a job in the rest of its itinerary. This is accomplished by replacing $B_{f,j}(i)$ in the above formulas with $D_{f,j}(i)$. MET therefore orders each job in a set Γ competing for processor j in state i via the index I_f^{MET}, calculated as:

$$\hat{I}_f = l_{f,j} + \max\{\ \varepsilon_{f,j} - t(i),\ T_{f,j}(i),\ tr_f(i)\ \} \tag{4.29}$$

$$\bar{I}_{f'} = \max\{\ \varepsilon_{f',j} - t(i),\ T_{f',j}(i), tr_{f'}(i)\ \} \tag{4.30}$$

$$I_f^{MET} = \sum_{f' \in \Gamma \setminus f} \left[[\hat{I}_f - \bar{I}_{f'}]^+ - D_{f',j}(i) + t(i) \right]^+ \tag{4.31}$$

   Similarly, WMET orders each train in Γ by the index:

$$\hat{I}_f = l_{f,j} + \max\{\ \varepsilon_{f,j} - t(i),\ T_{f,j}(i),\ tr_f(i)\ \} \tag{4.32}$$

$$\bar{I}_{f'} = \max\{\ \varepsilon_{f',j} - t(i),\ T_{f',j}(i), tr_{f'}(i)\ \} \tag{4.33}$$

$$I_f^{WMET} = \sum_{f' \in \Gamma \setminus f} \sigma_{f'} \left[[\hat{I}_f - \bar{I}_{f'}]^+ - D_{f',j}(i) + t(i) \right]^+ \tag{4.34}$$

9. *MRP (Min Remaining Processing Time):* MRP is extremely important in processor networks with many precedence constraints (such as classification terminals). MRP, which should not be confused with *Material Requirements Planning*, another popular heuristic for shop-floor scheduling, simply orders each job f in a set

Γ competing for processor j by the index $R_{f,j}(i)$:

$$I_f^{MPR} = R_{f,j}(i) \tag{4.35}$$

10. *WMRP (Min Weighted Remaining Processing Time):* WMRP is almost identical to MRP, except that rather than use the priority-less ranking index $R_{f,j}(i)$, we use the index calculated as follows:

$$\hat{I}_f = \max_{r \in \mathcal{R}(i): e_r = (f,j)} \left\{ R_{f,j_r} - s_r - R_{f'_r,j'_r} \right\} \tag{4.36}$$

$$I_f^{WMRP} = \frac{R_{f,j} + l_{f,j} + \hat{I}_f}{\sigma_f} \tag{4.37}$$

11. *EDD (Earliest Due Date):* EDD attempts to avoid the difficulties associated with estimating effects of congestion in the network, and just chooses among a set of jobs Γ competing for j that job f whose movement into j is tied to the most urgent "successor event," and therefore whose index $E_{f,j}(i)$ is the smallest.

$$I_f^{EDD} = E_{f,j}(i) \tag{4.38}$$

Implementation for Linehaul Dispatcher Regions

There are far fewer precedence constraints to consider in the case of linehaul dispatching problems than in the case of classification terminals, so data such as $D(i)$, $B(i)$, $R(i)$, and T are much easier to calculate, making each heuristic simpler to implement.

Heuristic Rules for Classification Terminals

In the case of classification terminals, because the set $\mathcal{R}(i)$ contains "circular" precedence relationships (see Section 3.4), $\hat{\mathcal{R}}(i)$ is defined a bit differently. A train-pair $\hat{r}$ is included under the same criteria as above, except with the added restriction that we exclude all relationships $(f_{\hat{r}}, f'_{\hat{r}})$ where $f_{\hat{r}}$ is an outbound train and $f'_{\hat{r}}$ is a car or cut, and where $f_{\hat{r}}$ is a car or cut and $f'_{\hat{r}}$ an inbound train.

4.2.2 Comments about Individual Heuristics

We now make some comments and observations about the performance of some these heuristics as applied to rail models LMPM and YSSM, paying particular attention to the circumstances under which each will perform well or not so well.

FIFO

Of all heuristics, perhaps FIFO is the most common. It can be an effective heuristic for two reasons. First, it is always a *non-delay policy*, from the perspective of the resource (either a process or segment of track) accepting trains, which means that a resource will never be idle when it need not be. When there is at least one train ready to enter a segment or area of track, it will do so. What happens when two trains arrive, one perhaps several hours before the other, but find whatever resource they must use next busy? When this crew is next available, they are both ready to be inspected. Clearly, as long as the crew immediately inspects one of them, it is following a non-delay policy. Is there ever any advantage to favoring the train which has been waiting longest over the other? There is only if we believe that an inbound train's arrival time is somehow correlated with the departure due times of the outbound trains carrying its freight, and the train priorities are roughly equal, so need not be considered. If this is the case, it would be the second reason FIFO can be effective.

In airport like situations, where we have banks of inbound trains bringing freight to banks of outbound trains (each inbound train has at least one car for each outbound train), then the actual sequence of inbound trains at the hump does not matter, and again we state that a *non-delay* schedule, one of which is the FIFO schedule, should be close to optimal. As a warning, however, consider the situation of Figure 4-1. Inbound train A arrives first and brings cars to outbound train 1, and inbound train B brings cars to outbound train 1, 2, 3, and 4. If the disassembly process is

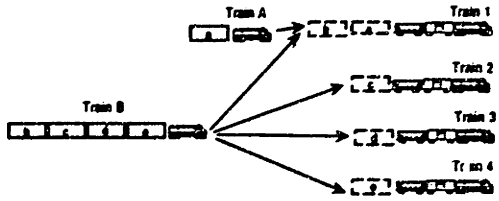


Figure 4-1: Train A arrives at 12:00 carrying cars for outbound train 1. Train B arrives at 12:15 carrying cars for outbound trains 1, 2, 3, and 4. It might be optimal to idle the disassembly engine between 12:00 and 12:15 so that we might disassemble Train B as soon as it arrives and has been inspected.

much slower than the assembly process (imagine this is a flat yard with three assembly engines and only one for disassembly), it may be worthwhile to delay disassembly until B arrives and disassemble it first. Outbound train 1 will not leave until both trains are disassembled, regardless of the sequence, but we might as well attempt to begin work on the assembly of train 2, 3, and 4 in the meantime.

In short, the circumstances under which we expect FIFO to perform poorly are:

1. Outbound trains have wildly different train priorities,
2. Departure target times are sufficiently uncorrelated with arrival times of inbound trains,
3. There is large overlap in hump job sets between outbound trains [1],
4. Different trains have different speeds, so if A and B compete for a resource, making a fast train A wait for slow train B may cause a much higher delay than doing the opposite,
5. There are nonlinear tardiness costs. Consider tardiness costs which rise quickly then flatten, for example. This is the case when we wish to minimize the number of late departures. We may then wish to give priority to the inbound trains which are part of the most hump job sets.

Min-Delay, Weighted-Min Delay

Our second set of heuristics is motivated by 4 above, and involve minimizing delay, not of the resource (track, crew, etc.), but of the train, which we might think of as the "customer." Where train speeds differ, a planning tool will have to take into account the fact that making a fast train wait for a slow train may not make sense, even if the slow train arrives at the resource first. There will therefore be a trade-off between maximizing the utilization of the track and crew resources, and maximizing the utilization of the train resources traveling across the line or terminal. Moreover, as congestion across the planning node increases, the advantage of FIFO (which is that it keeps resources busy whenever possible) will diminish, since non-FIFO decisions will less likely involve the forced idling of a resource. In this circumstance, when train speeds vary, our Min Delay and Weighted Min Delay heuristics should perform much better.

Min-Slack, Weighted Min-Slack

Min Slack was particularly appropriate for our objective function, which is to minimize weighted tardiness, because it is the only heuristic to explicitly take into account some measure of tardiness of trains when making decisions. MS and WMS do have one weakness, however. Consider the scenario of Figure 4-2. According to MS or WMS, Train A will be assigned the minimum slack and given priority, though this will not make the departure of Train 1 occur any sooner (it must still wait for C), while unnecessarily delaying Train 2. In such a situation, MD would be preferable.

[1] A *hump job set* for an outbound train o is the set of inbound trains which must be disassembled in order to assemble o [4].

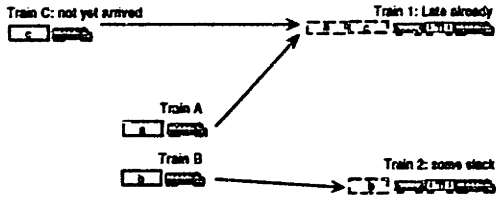


Figure 4-2: Trains A and B compete for hump. Train A carries cars for outbound train 1, which is already tardy, and B for outbound train 2, which has some slack. Outbound train 1, however, must also take cars from inbound train C, which is running very behind. Should A still take priority at the hump?

Min Remaining Processing Time

Consider the scenario of Figure 4-3. In such a situation, Trains A and B have the least slack time, as defined by the MS or WMS heuristics, and would therefore take priority at the hump. However, the decision which minimizes the sum weighted departure times (or tardinesses, if both trains will be tardy) is to give Train C priority at the hump and dispatch Train 2 as soon as possible.

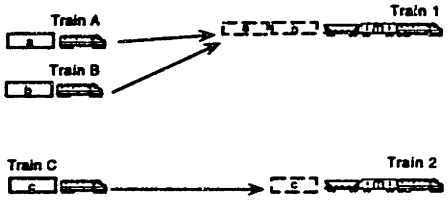


Figure 4-3: Three inbound trains, A, B, and C, wait ready for humping at the same time. A and B carry cars for outbound train 1, while C carries a final car for outbound train 2. Trains 1 and 2 have the same departure target time and priority.

4.3 Approximation of Cost-to-Go

To approximate J^* and J_μ, we employ a feature extraction mapping $F(i)$, coupled with two approximation architectures. Our first is a very simple cubic polynomial $P_w : F \mapsto \Re$, with trainable weights w. Our second is a slightly more sophisticated multi-layer neural network, $N_w : F \mapsto \Re$. In this section, we present the feature extraction mapping F, as well as a more formal discussion of the approximation architectures P_w and N_w. We defer discussion of the training of the parameterized weights w until Section 4.4.

4.3.1 Features

The features we employ to help estimate the cost-to-go of states in our approximate DP are motivated by the same considerations which motivated the heuristics of Section 4.2. We first present features for the generic MNSP*, then follow with any specific comments about how these might be applied to both classification terminals and linehaul dispatcher regions.

Generic Processor Network Features

1. *Average Accumulated Minimum Tardiness:* Assume each job completes the rest of its itinerary without delay. What would all the tardiness magnitudes be? This is the first feature of a a network state, and is calculated as:

$$F_{MT}(i) = \sum_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^- / |\mathcal{F}| \tag{4.39}$$

   We record an average rather than the raw sum, since that is more generalizable to different scheduling scenarios with different numbers of trains.

2. *Average Weighted Accumulated Minimum Tardiness:* Assuming again that each job completes the rest of its itinerary without delay, what would all the tardiness *costs* be? We calculate this statistic as:

$$F_{WMT}(i) = \sum_f \sigma_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^- / |\mathcal{F}| \tag{4.40}$$

3. *Average Accumulated Expected Tardiness:* This is almost equivalent to our first feature described above, expect that we use "expected" process times rather than non-delay times, by taking into account a measure of future network congestion. It is calculated by:

$$F_{ET}(i) = \sum_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^- / |\mathcal{F}| \tag{4.41}$$

4. *Average Accumulated Weighted Expected Tardiness:* Same as above, except weighted by job costs.

$$F_{WET}(i) = \sum_f \sigma_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^- / |\mathcal{F}| \tag{4.42}$$

5. *Average (Weighted) Min and Expected Earliness:* If we define earliness to be the negative part of lateness (or the positive part of slack), we have four new features which are analogues of our first four.

$$F_{ME}(i) = \sum_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^+ / |\mathcal{F}| \tag{4.43}$$

$$F_{WME}(i) = \sum_f \sigma_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^+ / |\mathcal{F}| \tag{4.44}$$

$$F_{EE}(i) = \sum_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^+/|\mathcal{F}| \quad (4.45)$$

$$F_{WEE}(i) = \sum_f \sigma_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^+/|\mathcal{F}| \quad (4.46)$$

6. *Largest Accumulated* {*Non-weighted, Weighted*} {*Minimum, Expected*} {*Tardiness, Earliness*}: We record the maximum of all these statistics over all jobs:

$$F_{lMT}(i) = \max_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^- \quad (4.47)$$

$$F_{lWMT}(i) = \max_f \sigma_f(i) [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^- \quad (4.48)$$

$$F_{lET}(i) = \max_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^- \quad (4.49)$$

$$F_{lWET}(i) = \max_f \sigma_f(i) [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^- \quad (4.50)$$

$$F_{lME}(i) = \max_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^+ \quad (4.51)$$

$$F_{lWME}(i) = \max_f \sigma_f(i) [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^+ \quad (4.52)$$

$$F_{lEE}(i) = \max_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^+ \quad (4.53)$$

$$F_{lWEE}(i) = \max_f \sigma_f(i) [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^+ \quad (4.54)$$

7. *Smallest Accumulated* {*Non-weighted, Weighted*} {*Minimum, Expected*} {*Tardiness, Earliness*}: We record the maximum of all these statistics over all jobs:

$$F_{sMT}(i) = \min_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^- \quad (4.55)$$

$$F_{sWMT}(i) = \max_{f' : f' = \arg\min_f (D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i))^+} \sigma_{f'} F_{sMT} \quad (4.56)$$

$$F_{sET}(i) = \min_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^- \quad (4.57)$$

$$F_{sWET}(i) = \max_{f' : f' = \arg\min_f (D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i))^-} \sigma_{f'} F_{sET} \quad (4.58)$$

$$F_{sME}(i) = \min_f [D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i)]^+ \quad (4.59)$$

$$F_{sWME}(i) = \min_{f' : f' = \arg\min_f (D_{f,P(f,N_f)}(i) - t(i) - Q_{f,P(f,N_f)}(i))^+} \sigma_{f'} F_{sME} \quad (4.60)$$

$$F_{sEE}(i) = \min_f [D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i)]^+ \tag{4.61}$$

$$F_{sWEE}(i) = \min_{f' : f' = \arg\min_f (D_{f,P(f,N_f)}(i) - t(i) - P_{f,P(f,N_f)}(i))^+} \sigma_{f'} F_{sEE} \tag{4.62}$$

8. *Work-in-Process, WIP:* We borrow this notion from operations management. In the setting of our processing network, we express WIP as the number of jobs with nonzero costs yet to complete their processing, divided by the total number of nonzero cost jobs in the system.

$$F_{WIP}(i) = \frac{|\{f : c_f > 0, K_f(i) < N_f\}|}{|\{f : c_f > 0\}|} \tag{4.63}$$

9. *Time-Modified WIP:* For any state i, defined as

$$F_{TWIP}(i) = F_{WIP}(i) \cdot t(i)/\mathcal{T}, \tag{4.64}$$

where $\mathcal{T}$ is the total time-horizon of the scheduling problem. Time-Modified WIP expresses the fact that a WIP close to 1 at time 0 is not so bad, while a WIP close to 1 at a time near $\mathcal{T}$ is another matter entirely.

10. *Average, Smallest Weighted and Non-weighted Expected Remaining-Processing:* This is a measure of how "far" we are from completion of our network objective, which is to complete every job, or at least every "cost-event." If we define the operator $[\cdot]^F$ by

$$[x]^F = x, \ x \in \Re,$$

$$[\infty]^F = 0,$$

then we can define Average Finite Remaining Processing by

$$F_{RP}(i) = \sum_f [R_{f,S_f(i)}(i)]^F / |\mathcal{F}|. \tag{4.65}$$

Similarly, we can define the Smallest Finite Remaining Processing as

$$F_{sRP}(i) = [\min_f R_{f,S_f(i)}(i)]^F \tag{4.66}$$

And finally we can define the corresponding weighted statistics:

$$F_{WRP}(i) = \sum_f [\sigma_f R_{f,S_f(i)}(i)]^F / |\mathcal{F}|. \tag{4.67}$$

$$F_{sWRP}(i) = [\min_{f' : f' = \arg\min R_{f,S_f(i)}(i)} F_{sRP}(i)/\sigma_{f'}]^F \tag{4.68}$$

11. *Average, Smallest Minimum Remaining-Processing:* Defined just as above, except that we use T in place of R.

$$F_{MRP}(i) = \sum_f [T_{f,S_f(i)}(i)]^F / |\mathcal{F}|. \tag{4.69}$$

$$F_{sMRP}(i) = [\min_f T_{f,S_f(i)}(i)]^F \tag{4.70}$$

$$F_{WMRP}(i) = \sum_f \sigma_f [T_{f,S_f(i)}(i)]^F / |\mathcal{F}|. \tag{4.71}$$

$$F_{sWMRP}(i) = [\min_{f':f'=\arg\min T_{f,S_f(i)}(i)} F_{sRP}(i)/\sigma_{f'}]^F \tag{4.72}$$

4.3.2 Approximation Architecture

Even if we believe the features described above are sufficiently descriptive to capture the principal components of a state's cost-to-go function, we may still have chosen an inadequate approximation architecture. As an initial approach, we chose a simple polynomial approximation architecture:

$$P_w \circ F(i) = \sum_{k\in\{1,\ldots,p\}} F_k(i)w_{k,1} + F_k(i)^2 w_{k,2} + F_k(i)^3 w_{k,3}. \tag{4.73}$$

This architecture was chosen for its simplicity. We then tried a multi-layer perceptron with the sigmoidal activation function $\phi : \Re^{m\times n} \mapsto \Re^{m\times n}$ defined by:

$$[\phi(X)]_{ij} = \frac{1}{1+e^{-[X]_{ij}}}. \tag{4.74}$$

This is commonly known as the *logistic function*, and is one of the most popular activation functions for neural network design. We tried two neural net architectures. Our first involved three layers with ten nodes each, two of which were activation layers, and is expressed as:

$$N_w \circ F(i) = w^a \phi(\, w^c \phi(\, w^e F(i) + w^f \,) + w^d \,) + w^b, \tag{4.75}$$

where, if $F(i) \in \Re^p$, $w^a \in \Re^{1\times 10}$, $w^b \in \Re$, $w^c \in \Re^{10\times 10}$, $w^d \in \Re^{10\times 1}$, $w^e \in \Re^{10\times p}$, and $w^f \in \Re^{10\times 1}$.

Our second involved two layers with seven nodes each, only one of which an activation layer, and is expressed as follows:

$$N_w \circ F(i) = w^a \phi(\, w^c F(i) + w^d \,) + w^b, \tag{4.76}$$

where, if $F(i) \in \Re^p$, $w^a \in \Re^{1\times 7}$, $w^b \in \Re$, $w^c \in \Re^{7\times p}$, $w^d \in \Re^{7\times 1}$.

4.4 "Learning" to make decisions

We now describe two methods for training the parameter weights w in $P_w \circ F$ and $N_w \circ F$, what we call, rather loosely, *on-line* training and *off-line* training. Both are analogous to policy iteration [13].

Under each paradigm, we fix w to some initial set of weights w_0, apply Procedure 4 using $P_{w_0} \circ F$ or $N_{w_0} \circ F$ as our cost-to-go function, then update w.

4.4.1 Updating parameter vector w: Online Learning

In the online case, we apply Procedure 4 from our origin state i_0, using P_{w_0} or N_{w_0}. Once a leaf is found, we evaluate the cost of that leaf, and can also calculate the cost-to-go for each state we have visited under the specific policy we have pursued. We use this information to update our weight vector to w_1, return to the root state, and solve again, this time using $P_{w_1} \circ F$ or $N_{w_1} \circ F$ to get a new policy. The method stops when both the policy and weights converge.

Given that we have found a leaf, one common approach for updating w is to use a gradient-like step method. Assume, for example, that while executing Procedure 4, we traversed n states, and that we name them 1 to n, with state n being the leaf, as in Figure 4-4. The trajectory we pursued

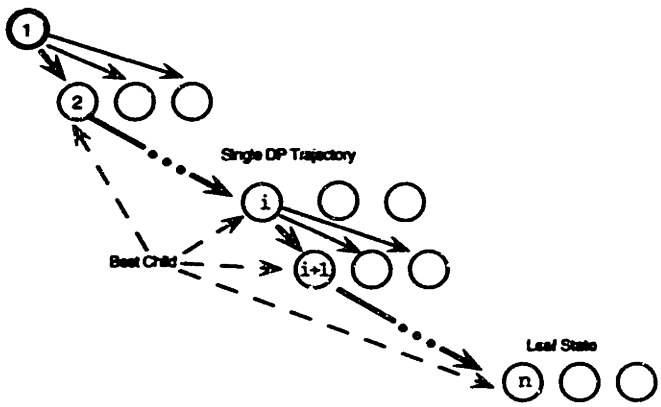


Figure 4-4: We start at the root node, 1, generate children for each heuristic control policy, evaluate each child i using $P_{w_0} \circ F(i)$, choose the best child, and proceed in the same way until we find our first leaf node, n. The path from 1 to n is a single Approximate DP Trajectory, and represents a complete, feasible schedule.

from state 1 to n represents a selection of heuristics at each state, and can be thought of as one possible policy. Since it is the policy which follows from our initial approximation weights w_0, we shall call this policy $\mu(w_o)$. Once we reach state n, we know the true cost-to-go at each state, at least under policy $\mu(w_0)$. This cost-to-go for any state i is given by

$$\begin{aligned} c(n) &= 0 \\ c(i) &= \sum_{i \le k < n} g(k, k+1). \end{aligned} \tag{4.77}$$

After calculating each $c(i)$, we can apply a gradient update step to w as follows:

$$w \leftarrow w - \alpha \left(c(i) - G_w \circ F(i)\right) \nabla_w G_w \tag{4.78}$$

Note that this implies that we make n updates to w each time we generate a full solution "trajectory." There are many variations of the above theme, as well as even more aggressive versions which allow learning to be done even before reaching the first feasible solution. For example, can we update w before reaching our first leaf n, perhaps improving the expected value of the final schedule we will generate? If we have confidence in our current approximations $G_w \circ F$, we might consider the term:

$$G_w(F(i)) - \left(g(i, i+1) + G_w(F(i+1))\right), \tag{4.79}$$

to be one indication of possible error in our approximation architecture, and might update w at every step moving *forward* in our DP as follows:

$$w \leftarrow w - \alpha \left(G_w(F(i)) - g(i, i+1) - G_w(F(i+1)\right) \nabla_w G_w, \tag{4.80}$$

This is the *Temporal Difference Method* TD(0) of Sutton [71]. It can be generalized to a class of parameterized methods TD(λ), which have demonstrated favorable empirical behavior in other applications (see [73, 76, 81, 82]).

Unfortunately, we found such gradient-based methods to be very difficult to make converge, so we tried a more conservative variation. We accumulated the full history of $c(i)$ and solved the stationary optimization problem:

$$\min_w \sum_i \frac{1}{2} \left(c(i) - G_w(F(i))\right)^2 \tag{4.81}$$

In the case of our polynomial architecture P_w, this is the linear least squares problem solved by:

$$w \leftarrow (X'X)^{-1} X'c, \tag{4.82}$$

where c is the vector of observations $c(i)$, and X is the matrix of data points whose rows are indexed

by i, and columns by the terms of the polynomial P_w:

$$X = \begin{bmatrix} F(1) & F(1)^2 & F(1)^3 \\ \vdots & \vdots & \vdots \\ F(n) & F(n)^2 & F(n)^3 \end{bmatrix}.$$

We found this to be much more stable than any variation of TD(λ). In fact, we found ourselves consistently converging to stable weights within 2 - 6 iterations. Unfortunately, the policies to which we converged were uniformly bad, typically worse than the random policy $\mu(w_0)$ and often worse than a majority of our heuristics. In fact, in our computational experiences, we saw no apparent correlation between the quality of $\mu(w)$ and our likelihood of converging to w.

Why did our our on-line learning approaches not succeed? An important issue to consider while training an approximation architecture is what is called *exploration*, discussed by Barto, *et. al.* [9]. We must ensure that, before allowing ourselves to converge to a solution in parameter space, we have sufficiently explored the state space so that our training of $P_w(F)$ is done on a sufficiently informative data set. One way of doing this might be to borrow an idea from simulated annealing and apply random policies for several iterations, then apply Procedure 4 only from there. Another idea is to keep several initial w_0's, and apply Procedure 4 for each of these. Perhaps the strongest way to guarantee sufficient exploration of the state space, however, is to collect a large amount of historical (or simulated) data and train on that. This brings us to discuss what we call *off-line learning*.

4.4.2 Updating parameter vector w: Off-Line Learning

Off-line learning proceeds in a very similar manner to our on-line approach. Approximate policy iteration corresponds to specifying an initial parameter weight w_0, then generating the "greedy" policy $\mu(w_0)$. This time, however, rather than simulate $\mu(w_0)$ on a single problem instance, we simulate it on an entire history (or artificially generated "history") of scheduling scenarios, and collect feature and cost-to-go data on every state in our history.

Alternatively, we may just simulate each heuristics $h \in H$ on a large battery of test problems, collect cost-to-go and feature data for every state, then perform an off-line data fit to find an approximation G_w to each heuristic-specific cost-to-go $J^\mu(i)$. We will see in Section 4.4.3 how we

might wish to use these estimates.

The training of w for our polynomial architecture proceeds identically as before, except that the data matrix X is significantly larger, which raises certain numerical issues addressed in Section 4.5. In the case of our neural net N_w, the training of w was significantly more difficult. Letting $N(w)$, c, F, and $e(w)$ be the vectors defined by

$$N(w)_i = N_w \circ F(i),$$

$$c_i = c(i),$$

$$F_i = F(i)',$$

and

$$e(w) = N(w) - c,$$

where $i = 1, \ldots, q$ indexes all the states in our historic data set, we can say that our least squares problem is to minimize $\frac{1}{2}\|e(w)\|^2$. The Gauss-Newton method [12] involves the iteration:

$$w^{k+1} \leftarrow w^k - (\nabla e(w^k)\nabla e(w^k)')^{-1}\nabla e(w^k)e(w^k).$$

We apply the *Levenberg-Marquardt* variant of this [12], which augments the approximate Hessian

$$\nabla e(w^k)\nabla e(w^k)'$$

with the term $\lambda I_{p\times p}$, where λ is large enough that the condition number of

$$\nabla e(w^k)\nabla e(w^k)' + \lambda I_{p\times p}$$

is smaller than some tolerance.

More formally, the Levenberg-Marquardt update of w is:

$$w^{k+1} \leftarrow w^k - (\nabla N(w^k)\nabla N(w^k)' + \lambda I_{p\times p})^{-1}\nabla N(w^k)N(w^k). \tag{4.83}$$

We also explored adding a momentum term to help us avoid local minima, as in the *Heavy Ball*

Method of Poljak.

$$w^{k+1} \leftarrow w^k - (\nabla N(w^k)\nabla N(w^k)' + \lambda I_{p\times p})^{-1}\nabla N(w^k)N(w^k) + \gamma(w^k - w^{k-1}). \tag{4.84}$$

Note that these expressions allow one to implement gradient steps in an incremental manner, where a step is taken for individual blocks of data at a time, or a more comprehensive manner where gradient steps are taken on the entire data set. Usually, memory constraints prohibit this last approach, except for small data sets, and in any case, it is imperative that an implementation store and manipulate the terms $\nabla N(w^k)$ as sparse matrices, since their calculation involves extensive use of Kronecker products of large matrices.

We embedded Levenberg-Marquardt, as well as a standard gradient descent, within a two-level incremental filter approach, defined in Procedure 5. In the innermost loop, we can take either the gradient steps or approximate Newton steps, and found both techniques to be worthwhile at different times.

4.4.3 Rollout, Approximate Rollout, and Limited Lookahead

Consider a single-heuristic control policy $\mu = h$, and let $J^\mu(i)$ be the exact cost-to-go for state i under μ. We might define, for each state i and control $u \in U(i)$:

$$Q^\mu(i,u) = g(i,d(i,u)) + J_\mu(d(i,u))$$

to be the cost-to-go for i, assuming that we execute control u at i and follow with policy μ afterward. These terms Q^μ are known as Q-factors [79], and there exists an extensive literature surrounding their use and estimation. For our purposes, however, it will suffice to define a *μ-Rollout-Policy* as the policy obtained by choosing, at each state i, the heuristic control $h^* \in H$ which minimizes $Q^\mu(i,h)$ over $h \in H$. Single heuristic rollout policies were first introduced by Tesauro and Galperin [75], and analyzed in detail by Bertsekas, Tsitsiklis, and Wu [14], who then applied it to a machine repair problem. We also define an *Approximate μ-Rollout Policy* obtained by choosing, at each state i the heuristic h which minimizes

$$\hat{Q}^\mu(i,h) = g(i,d(i,h)) + \hat{J}_\mu(d(i,h)).$$

l = BLK_SIZE, $u = 1$,
Initialize MAX_IN_COUNT, w^0, ERR_TOL, IN_ERR_TOL,
Initialize $\gamma \in [0,1)$, α, β, $\Delta_o = 0, \kappa = 0$
Initialize MAX_COUNT $\propto \frac{q}{\text{BLK_SIZE}}$
while $e(w)'e(w) >$ ERR_TOL and $k <$ MAX_COUNT **do**
  if $u +$ BLK_SIZE $> q$ **then**
    $l = 1$, $u =$ BLK_SIZE
  else
    $l = l +$ BLK_SIZE, $u = u +$ BLK_SIZE
  end if
  $\kappa = 0$;
  $\hat{F} = \begin{matrix} F_l \\ \vdots \\ F_u \end{matrix}$
  while $\kappa <$ MAX_IN_COUNT and $\hat{e}(w)'\hat{e}(w) >$ IN_ERR_TOL **do**
    $\hat{w}^\kappa \leftarrow w^k$
    $\hat{N}(\hat{w}^\kappa) = N_{\hat{w}^\kappa} \circ \hat{F}$
    $\hat{e}\hat{w}^\kappa = \hat{N}(\hat{w}^\kappa) - c$
    if we do Levenberg-Marquardt **then**
      $\hat{w}^{\kappa+1} \leftarrow \hat{w}^\kappa - (\nabla\hat{N}(\hat{w}^\kappa)\nabla\hat{N}(\hat{w}^\kappa)' + \lambda I_{p\times p})^{-1}\nabla\hat{N}(\hat{w}^\kappa)\hat{N}(\hat{w}^\kappa)$.
    else { we do simple gradient descent}
      $\hat{w}^{\kappa+1} \leftarrow \hat{w}^\kappa - \nabla\alpha\hat{N}(\hat{w}^\kappa)\hat{N}(\hat{w}^\kappa)$.
    end if
    $\kappa \leftarrow \kappa + 1$
  end while
  $\hat{w} \leftarrow \hat{w}^\kappa$
  $\Delta_k \leftarrow \gamma\Delta_{k-1} + (1-\gamma)(\hat{w} - w^k)$
  $w^{k+1} \leftarrow w^k + \beta\Delta_k$
  $k \leftarrow k + 1$
end while

Procedure 5: Two-Level Incremental Gradient Descent

It is known that μ-Rollout corresponds to one exact policy iteration on the policy μ, and therefore is strictly better than μ, unless μ is itself optimal. Moreover, it has been empirically observed that policy iteration manifests steeply diminishing returns in time. This means that most of the policy improvements come in the first few iterations. The motivation for rollout, therefore, is that perhaps a large amount of this potential improvement might come from the first iteration alone.

We define the more extensive *H-Rollout Policy* to be the policy defined by choosing at each i the heuristic $h^*(i)$ defined by

$$h^*(i) = \arg \min_{h \in H} \min_{\mu \in H} Q^\mu(i, h),$$

and *Approximate H-Rollout Policy* to be defined by

$$h^*(i) = \arg \min_{h \in H} \min_{\mu \in H} \hat{Q}^\mu(i, h).$$

Approximate H-Rollout is actually just Procedure 4 implemented with the unique feature

$$\min_{\mu \in H} \hat{J}_\mu.$$

A final variation which we shall explore is to utilize the mean heuristic-specific approximate costs-to-go, rather than the minimum. In other words, we will make decisions based on the unique feature

$$\text{mean}_{\mu \in H} \hat{J}_\mu.$$

This may seem like a strange feature, because the rollout cost-to-go for any state is an upper bound on its true optimal cost-to-go, and therefore the only rollout cost-to-go we should want to pay attention to for any state is the smallest one. However, if we believe the actual costs-to-to $\hat{J}_\mu(i)$ to be highly positively correlated across different policies μ, (for example, if a "good" state were typically good under multiple policies) but the estimation errors in $\hat{J}_\mu$ to be uncorrelated and large relative to the differences in cost-to-go between the different heuristics, then the mean $\hat{J}_\mu$ could be a lower-variance estimator for J^* than the minimum $\hat{J}_\mu$. More formally, our approximate cost-to-go can be expressed as

$$\hat{J}_\mu(i) = J^*(i) + A(i, \mu(i)) + e(i, \mu)$$

where

$$A(i,u) = J_\mu(i) - J^*(i) \geq 0$$

is the incremental cost of applying policy μ rather than the optimal policy at state i, and $e(i,\mu)$ are error terms. If the magnitude of the $A(i,\mu)$ are small relative to the errors $e(i,\mu)$, then we might expect that the mean $\hat{J}_\mu(i)$ over all μ to be a better estimate of $J^*(i)$ than the minimum over all μ.

4.4.4 Optimization in Policy Space

In all the previous discussion, we have accepted the implicit goal that the purpose of the approximation of our cost-to-go is to correctly estimate the true cost-to-go. The motivation for this goal is that we know that a perfect approximation to our cost-to-go would lead to an optimal control policy. We therefore hope that a "close" approximation to our cost-to-go will therefore lead to a "close-to-optimal" policy. Unfortunately, this assumption might not be valid at all for any specific problem.

What determines our policy is not explicitly the value of our approximate cost-to-go, but rather how these values differ across adjacent states, where we define "adjacent" to mean that two states are both children of the same parent. In this sense, what is really important in our approximation is not the accuracy of the cost-to-go estimates themselves, but the accuracy of the "finite-differences" of estimates between adjacent states, and a function which is optimized to minimize least-square errors may not necessarily correctly estimate these finite-differences.

Since it is policies we are ultimately after, one might suggest that we attempt to train our approximation parameters w in such a way to generate good policies and not worry about the accuracy of the cost-to-go estimation. Indeed, many successful users of ADP have observed that their approximation architectures do a *poor* job of approximating true costs-to-go (see [72, 81]). Such a strategy is known as *Optimization in Policy Space*. Of course, since we do not know *a priori* what good policies are, and there is no direct functional relationship between w and the value of $\mu(w)$, such an optimization is very difficult. This leads to the "expert based" training strategy of Table 4.1: We will refer to any cost-to-go approximation generated in the above manner as an *expert based ADP*, or *e*-ADP.

As an example of the possible advantage of an *e*-ADP, consider the one-stage control problem illustrated in Figure 4-5. We start with equal probability at one of the 5 jobs at salary level 1. We

1. Heuristically create a cost-to-go approximation, with the goal of first estimating finite-differences, and only secondly estimating values.
2. Assemble a battery of archetypal scheduling scenarios, and simulate an ADP policy using the cost-to-go approximation.
3. Observe the simulated controls, and compare control decisions with those of an expert, or off-line optimization tool.
4. Make adjustments to cost-to-go approximation to correct errors in above control decisions
5. Repeat as long as necessary or possible.

Table 4.1: "Expert" led Optimization in Policy Space

are offered a raise to salary level 2, which we can either accept or decline. The game ends with the reward being the utility of our jobs. We choose to solve this problem via approximate dynamic programming, and utilize a linear approximation architecture to map salary levels with reward. Clearly, the optimal linear fit is a line with slope -1, while a line with slope +1 generates an optimal control.

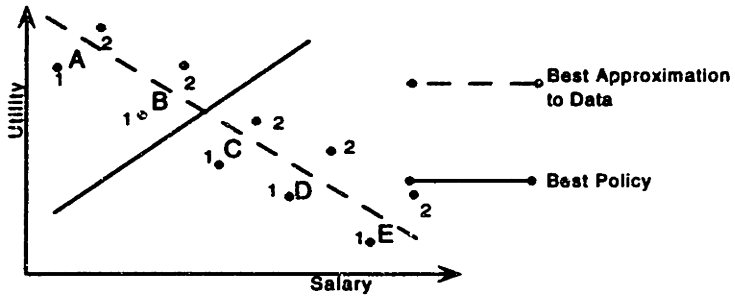


Figure 4-5: Society has 5 jobs, A, B, C, D, and E, and each has two pay levels, 1 and 2. Each job is a data point in the above graph, which plots utility level on the y-axis and pay level on the x-axis. Job A therefore has a lower salary than B, and B is lower than C, etc., all the way to E, which has the highest salary. However, Job A has the highest total utility, followed by B, all the way to E, which is the worst job. Within each job category, however, salary level 2 is higher than salary level 1, and so the utility of salary level 2, for any given job, is also higher.

Of course, this example merely illustrates the dangers of statistical analysis with an under-defined model. Had the user, for example, attempted to estimate utility as a function of both salary and job type, the proper coefficient of +1 for salary would have been derived, and an optimal policy would follow. Unfortunately, the problem we are trying to estimate is one where statistical analysis may by its nature by very difficult.

4.5 Implementation Issues

There are several important implementation issues to be considered, the most important being the numerical solution of our least squares problem and the extension of our formulation to network-wide scheduling problems. We discuss each in turn.

4.5.1 Data Conditioning, Scaling, and Reduction

Solving the equation $w = (X'X)^{-1}X'c$ can be difficult, as the inversion of the symmetric matrix $X'X$ is inaccurate in the presence of ill-conditioning. We therefore have found the need to perform significant scaling of our feature extraction mapping, and to invert $X'X$ by using QR Decomposition and substitution to solve the system of equations

$$X'Xw = X'c. \tag{4.85}$$

Memory is also an issue in our Neural Network training, especially when we take Gauss-Newton steps on large data-blocks. It is for this reason that we developed the two-level "filter" approach of Procedure 5 to training our parameters. Moreover, we had to perform significant data reduction via principal components analysis, although we selected our principal components to maximize correlation with the observed data rather than explanation of total variation in data. Our final approximation is therefore

$$N_w \circ F(i) = w^a \phi(\ w^c \phi(\ w^e (T'F(i) - \bar{X}) + w^f\ ) + w^d\ ) + w^b, \tag{4.86}$$

where T is a matrix which extracts the principal components of F demonstrating the greatest correlation with our observed cost-to-go data.

4.5.2 Network-Wide Scheduling

Another important issue we must address involves the solution of global scheduling problems. The DP formulations we have described above involve only single linehaul dispatcher regions, or single classification or switching terminals. How shall we move from this formulation to generate schedules for an entire network? There are two possibilities. First, we might apply the same type of decomposition to our network that was applied in Chapters 2 and 3, solving individual terminal or line

planning problems independently, then coordinating their solution via a fixed-point method.

An alternative approach would be to incorporate all planning problems into a single large DP formulation, where, given N nodes (lines, terminals, etc.), the state is an N dimensional vector of states for each node, the cost-to-go is the sum of approximate costs-to-go for each node sub-state in our state, and our control is to select among the N dimensional vectors of heuristic controls for each node. Of course, if there are 10 heuristics for each node, then the cardinality of the control space would by 10^N, so this would be infeasible for any realistic networks. We could resolve this issue in several ways. First, we could generate a battery of network-level heuristics to prune the network control space, in the same way we prune the node control space now. Alternatively, we could restrict all nodes to the same heuristic at any time period. While this may seem extremely dissatisfying, remember that this is exactly what we do at the node level. There is no reason to believe, for example, that in a given time period when FIFO is optimal at the hump, a different heuristic might not be optimal at another terminal resource. By constructing a formulation which permits only one active heuristic in a node at a time, we cannot optimally schedule such scenarios.

Our third approach is a hybrid of these two extremes. At any time period, evaluate an optimal heuristic at each node individually by assuming that over the next τ minutes, nothing else changes at all other nodes. Note that this involves N independent ADP-style evaluations, which could be sent to separate processors. After making each such determination, the control vector $\bar{u}$ is assembled by setting each element to the corresponding best control for each independent node evaluation. Under such a scheme, and assuming an architecture that assigns one processor for each node, we'd expect the solution times for network ADP's to be of a similar order of magnitude as those for a single node. Differences in solution times should arise, of course, from communication and synchronization issues. The formal solution approach is described in Procedure 6.

repeat
  Let $\vec{i} = (i_1, i_2, \ldots, i_N)$ be the state of every node in our network.
  for all $k \in \{1, \ldots, N\}$ **do** {Can be sent to parallel processors!}
    for all $h \in H$ **do** {These can also be parallelized}
      Simulate h at node k *only*.
      Form artificial control $u^{k,h} = \{\emptyset, \emptyset, \ldots, h_k, \emptyset, \ldots, \emptyset\}$
      Specify child state $\vec{i}_{k,h} = d(i, u_{k,h})$.
    end for
    $u_k = \arg\min_h g(\vec{i}, \vec{i}_{k,h}) + C_w(\vec{i}_{k,h})$
  end for
  Set network control to be $\bar{u} = (u_1, u_2, \ldots, u_N)$
  Set new global state $\vec{i} \leftarrow d(i, \bar{u})$
until $\vec{i} \in S^N$

Procedure 6: One-Step Network ADP Solution

Chapter 5

ADP Case Study: Hump Yard Switching and Sequencing

Chapter 4 introduced a generic formulation of MNSP* as a dynamic program, and described a comprehensive approximation scheme for its solution. In this chapter, we describe one case study of such an approximation scheme, where we solve YSSM using several different ADP architectures.

5.1 Case Study Environment and Solution Approaches

We start by describing the experimental environment and permutations of solution approaches utilized in this case study.

5.1.1 Experimental Environment

To evaluate our solution procedure, we took YSSM scenario A (the smallest, light congestion scenario described in Section 3.7.3) and generated 100 randomized variations of it by perturbing each inbound train's start time by -6 to 6 time periods. We then set each outbound train f's departure deadline to be

$$d_f = \varepsilon_{f,N_f} + P_d,$$

where P_d is a random integer between 0 and 12, and its cost to be

$$c_f = 10 + P_c,$$

where P_c is a random integer between -6 and 6. We chose scenario A because we wanted to be able to also use the IP formulation of Section 3.4 to generate optimal schedules with which to compare our ADP, and this scenario solved quickly enough for us to generate hundreds of IP solutions at a time. It is important to note, however, that the main attraction of our heuristics and ADP approach is that, unlike the IP, the computation times do not grow exponentially with the size of the problem or planning horizon. Moreover, the ADP, unlike our IP formulation, is rather insensitive to schedule congestion. In fact, we used e-ADP to solve 10 variations Scenario J of Chapter 3, which is the 40 train, tight congestion IP scenario which did not solve in 24 hours of CPU time. e-ADP attained solutions in an average of 3 minutes and 50 seconds. Finally, we note that these computation times give us considerable flexibility to add fidelity to the model without the fear that a small change in detail will turn our problem into an intractable one.

Chapter 4 presented an extensive discussion of all the components of a heuristic-based ADP strategy. In fact, any strategy can really be thought of as a permutation of design decisions, allowing the modeler to mix and match approximation architectures, learning rules, lookahead strategies, and policy update procedures. We discuss a broad set of these permutations in this chapter, and tested forty-two solution approaches on our battery of test problems. In addition to these, we spent considerable time with on-line variations of TD(λ), which, as discussed in Section 4.4, never showed sufficient promise for us to include them in this collection.

5.1.2 Solution Strategies

The solution schemes were:

1. Apply each heuristic itself. The choices are:
   (a) FIFO
   (b) mFIFO
   (c) LIFO
   (d) MD
   (e) WMD

(f) MS

(g) WMS

(h) MRP

(i) WMRP

(j) EDD

2. Apply one heuristic at each stage, chosen randomly. We call this r-ADP, and include it simply as a base for comparison.

3. Apply the 10 Heuristics and choose the best one. We call this a-H.

4. Randomly choose a heuristic at each time period. We perform this only for comparison purposes and do not expect it to do well. We call this r-H.

5. Use Procedure 4 with our ten heuristics, but make decisions using a control-directed cost-to-go function generated by a human scheduling "expert," as described in Section 4.4.4. We call this expert-ADP, or e-ADP.

6. Use approximate H-Rollout (henceforth ARLT), as described in Section 4.4.3 on 10 heuristics. To estimate $J^{\mu}(i)$, we use a polynomial approximation architecture, with weights fitted from a different set of 100 similarly randomized scheduling scenarios. The polynomial architectures we explored were:

   (a) Linear

   (b) Quadratic

   (c) Constrained Linear, where weights are either constrained to be greater than 1, or constrained to be less than minus 1, depending on the feature.

   (d) Constrained Quadratic (constraints just as above).

   We also explored cubic architectures, but found that the cubic term added little explanatory power to our approximation and created computational difficulties due to ill-conditioning of our data, so we abandoned this architecture.

   Moreover, for each approximation architecture, we utilized a min as well as mean control selection strategy (see Section 4.4.3)

7. Use ARLT on 10 heuristics with a neural net approximation architecture, trained with the same training set described above.

8. Implement a k-step lookahead policy (henceforth LKHD-k) under each of the following architectures:

   (a) e-ADP

   (b) linear, constrained linear, quadratic, and constrained quadratic under min-control.

   (c) linear, constrained linear, quadratic, and constrained quadratic under mean-control.

   (d) neural network under min-control

   (e) neural network under mean-control

For each architecture, k was set to 4, except for e-ADP, where both $k = 2$ and $k = 4$ were tried, for comparison purposes.

9. κ-ADP, using the e-ADP.
10. Full rollout (single exact policy iteration) on four most promising heuristics
    (a) MRP
    (b) WMRP
    (c) WMS
    (d) WMD
11. Extended ADP with e-ADP, which we call *xtd*-ADP.

We therefore have a battery of forty-two different solution approaches for YSSM. Our best performers were the rollouts, and all the variations on e-ADP. Moreover, their superiority was rather substantial. As mentioned, we were unsuccessful at initial attempts to implement a TD(λ) or other on-line learning scheme, so did not pursue them for this case study. Our initial attempts at approximate policy iteration on ADP were similarly unsuccessful.

5.1.3 Utilized Features

We employed the following thirty six features, which will henceforth be referenced by the corresponding indices.

1. Work-in-Process (WIP)
2. Time-based Work-in-Process (TWIP)
3. Average Expected Remaining Processing Time (RP)
4. Smallest Expected Remaining Processing Time (sRP)
5. Second Smallest Expected Remaining Processing Time
6. Third Smallest Expected Remaining Processing Time
7. Average Minimum Remaining Processing Time (MRP)
8. Smallest Minimum Remaining Processing Time (sMRP)
9. Second Smallest Minimum Remaining Processing Time
10. Third Smallest Minimum Remaining Processing Time
11. Average Weighted Remaining Processing Time (WRP)
12. Weighted Smallest Remaining Processing Time (sWRP)
13. Weighted Second Smallest Remaining Processing Time

14. Weighted Third Smallest Remaining Processing Time
15. Average Weighted Minimum Remaining Processing Time (WMRP)
16. Smallest Minimum Remaining Processing Time (sWMRP)
17. Second Smallest Minimum Remaining Processing Time
18. Third Smallest Minimum Remaining Processing Time
19. Average Accumulated Expected Tardiness (ET)
20. Average Accumulated Weighted Expected Tardiness (WET)
21. Largest Accumulated Expected Tardiness (lET)
22. Largest Weighted Accumulated Expected Tardiness (lWET)
23. Smallest Accumulated Expected Tardiness (sET)
24. Weighted Smallest Accumulated Expected Tardiness (sWET)
25. Average Accumulated Minimum Tardiness (MT)
26. Average Accumulated Weighted Minimum Tardiness (WMT)
27. Largest Accumulated Minimum Tardiness (lMT)
28. Largest Weighted Accumulated Minimum Tardiness (lWMT)
29. Smallest Accumulated Minimum Tardiness (sMT)
30. Weighted Smallest Accumulated Minimum Tardiness (sWMT)
31. Average Accumulated Expected Earliness (EE)
32. Average Accumulated Weighted Expected Earliness (WEE)
33. Largest Accumulated Expected Earliness (lEE)
34. Largest Weighted Accumulated Expected Earliness (lWEE)
35. Smallest Accumulated Expected Earliness (sEE)
36. Weighted Smallest Accumulated Expected Earliness (SWEE)

Each of these is described in Chapter 4. We did not utilize Minimum Earliness features, since these were almost always zero.

5.2 Summary of Case Study Conclusions

We now summarize the important findings of our computational experiments, and follow this summary with a more detailed discussion of each point.

1. *Value of heuristics and a-H:* Some of our heuristics are quite good individually, and as a group, the battery performed very well. The brute force solution defined by a-H is certainly a valid approach.

2. *"Expert-motivated" Optimization in Policy Space:* "Expert" approaches focused on generating good policies beat data-driven approaches focused on generating approximations in almost all cases.

3. *Best Approaches:* Our rollouts and e-ADP all beat a-H, but a-H did better than all our approximate rollouts, which compared favorably to each heuristic individually. In fact, several of our approximation architectures beat every individual heuristic on average.

4. *Value of Constraints:* For ARLT in the polynomial case, our constrained architectures outperformed their unconstrained counterparts even though they generated inferior approximations, further demonstrating that "expert" intuition can be more powerful than exclusively data-driven approximation.

5. *Value of Neural Network Approximation:* Neural network approximations did a better job of approximating costs-to-go (at least on training data), and did lead to better policies, but were much harder to train.

6. *Mean vs. Min Control:* Mean-based ARLT policies did almost always perform better than min-based policies, consistent with our hypothesis that heuristic specific costs-to-go are highly correlated.

7. *Rollouts:* Our best performers were our rollouts, demonstrating that exact policy iteration, for even one iteration, can lead to dramatically improved policies.

8. *Value of Extended Computation:* Adding "brute-force" computation to the solution procedure can be powerfully beneficial, as in the case of rollouts, but not necessarily so. In fact, LKHD, surprisingly, performed worse than non-lookahead policies under every single scenario, although LKHD-4 did better than LKHD-2 for our ADP.

   However, κ-ADP did quite often beat e-ADP (it always does at least as well), and even *xtd*-ADP.

9. *Random Controller:* r-H performed worse than even the worst heuristics individually. This was surprising; we expected r-H to perform, on average, as well as the average heuristic. This suggests that there is some structure within heuristics which favor consistent application over haphazard selection of control decisions. In this sense, we find it an encouraging result.

5.2.1 Performance of Heuristics

Our first observation is that the performance of our heuristics was quite good. Although mFIFO and LIFO performed poorly, as expected, others such as MRP and WMD were on average within 10% of the optimal schedule. Figure 5-1 plots the relative optimality gap (ROG) for each heuristic over each of the 100 problem instances. ROG is defined to be:

$$\frac{\text{heuristic value} - \text{optimal value}}{\text{optimal value}}.$$

We see that heuristics such as MRP and WMS, which are best on average, also seem to be best rather consistently as well. Figures 5-2 and 5-3 plot the performance of a-H. This is where the value of our

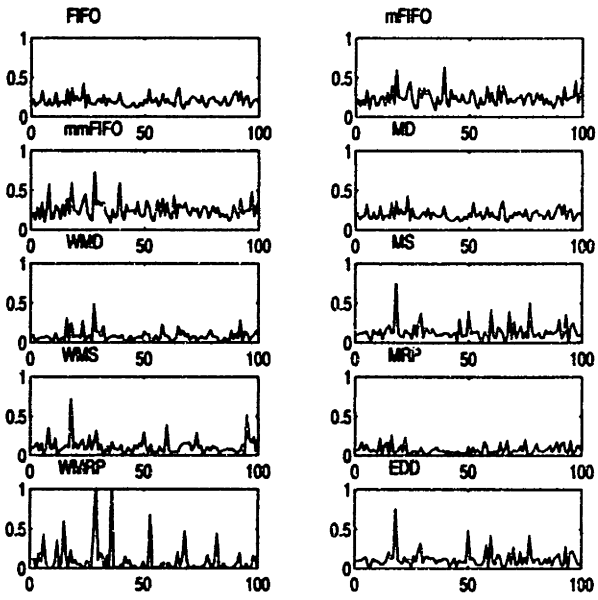


Figure 5-1: Each plot charts the relative optimality gap of each heuristic over our 100 problem instances. Relative optimality gap (ROG) is defined as: (heuristic value - ip value)/(ip value)

heuristics are truly demonstrated. While the best two heuristics (MRP, WMD) had average ROG's

of 7.3% and 8.1%, respectively, a-H had an average ROG of 2.5%, making it extremely difficult to beat. For this reason, we say quite without reservation that a simple policy of assembling a battery of heuristics for this scheduling problem, running all of them, and picking the best one is a legitimate heuristic policy in its own right.

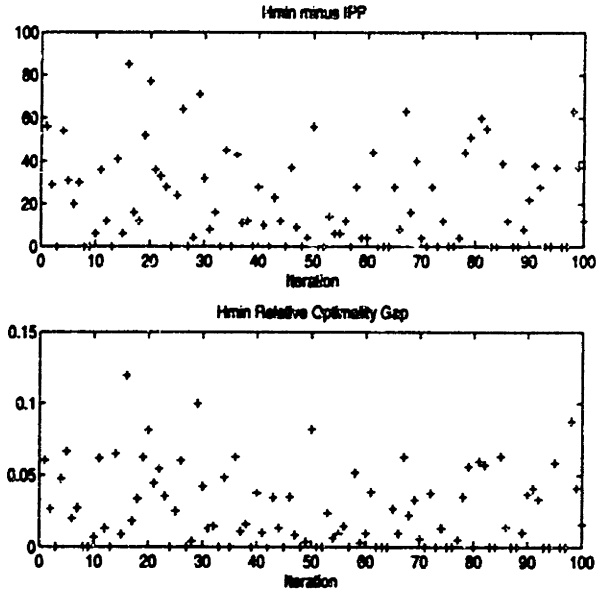


Figure 5-2: The top plot charts the value of a-H minus IPP, while the bottom plots the ROG for a-H.

5.2.2 Performance of e-ADP, xtd-ADP, κ-ADP

We were also pleased with the performance of e-ADP, and its two augmentations, xtd-ADP and κ-ADP. The latter two simply exchange a bit more computational effort for marginal improvements in solution quality. The top right plot of Figure 5-4 demonstrates clearly the success of e-ADP. While our heuristics could generate solutions that were as much as 120% from optimal, and typically as much as 20% from optimal, e-ADP found the optimal schedule in 64% of the instances, and averaged 3.4% from optimal on the other 36 (giving it an overall average ROG of 1.23%). Even compared to a-H, which was itself a strong performer, e-ADP did quite well, as the bottom two plots of Figure 5-4 demonstrate, although there were 17 instances where a-H beat e-ADP.

Should greater solution quality be required, xtd-ADP is always an option. As Table 5.3 demonstrates, xtd-ADP was a solid performer, providing solutions that averaged 0.7% from optimal, and

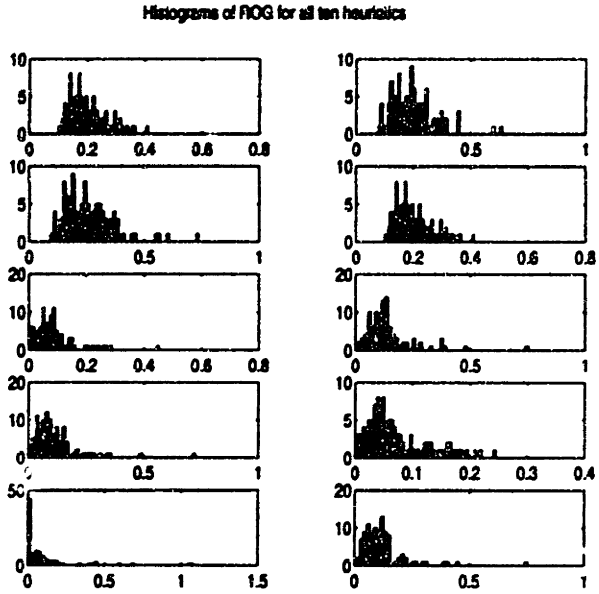


Figure 5-3: We plot histograms of the relative optimality gaps for each heuristic. Note that, while there is a good deal of spread for most heuristics, mean-ROG based comparisons between heuristics correspond to comparisons based on entire distributions as well.

were in fact optimal in 72 of 100 cases. Moreover, even better performance can be generated with κ-ADP, which was optimal in 77 cases and generated solutions that were on average 0.61% from optimal. Figures 5-5 and 5-6 illustrate several comparisons between κ-ADP and e-ADP, a-H, and *xtd*-ADP. Finally, for good measure, we point out that an Extended-κ-ADP, where we solve κ-ADP, then simulate all ten heuristics, and take the best schedule, would have been optimal in 81 instances and yielded schedules that were on average 0.36% from optimal.

5.2.3 Performance of Rollouts

Our star performers were by far our rollout policies. Recall from Section 4.4.3 that a rollout policy on a heuristic corresponds to one iteration of *exact* policy iteration, and that it has been empirically observed that the first policy iteration often yields the largest performance increase.

This was certainly the case in our examples. Our best performing heuristics, WMS, WMD, MRP, and WMRP were optimal 1, 4, 0, and 26 times, respectively. Their average ROG's were 11.3%, 8.1%, 7.3%, and 10.1%. A single policy iteration on each of these heuristics generated new policies that were optimal in 87, 83, 94, and 91 cases, respectively, and had ROG's of 0.2%, 0.5%, 0.1%, and 0.3%. There was not a single instance of our 100 scenarios where some rollout policy was

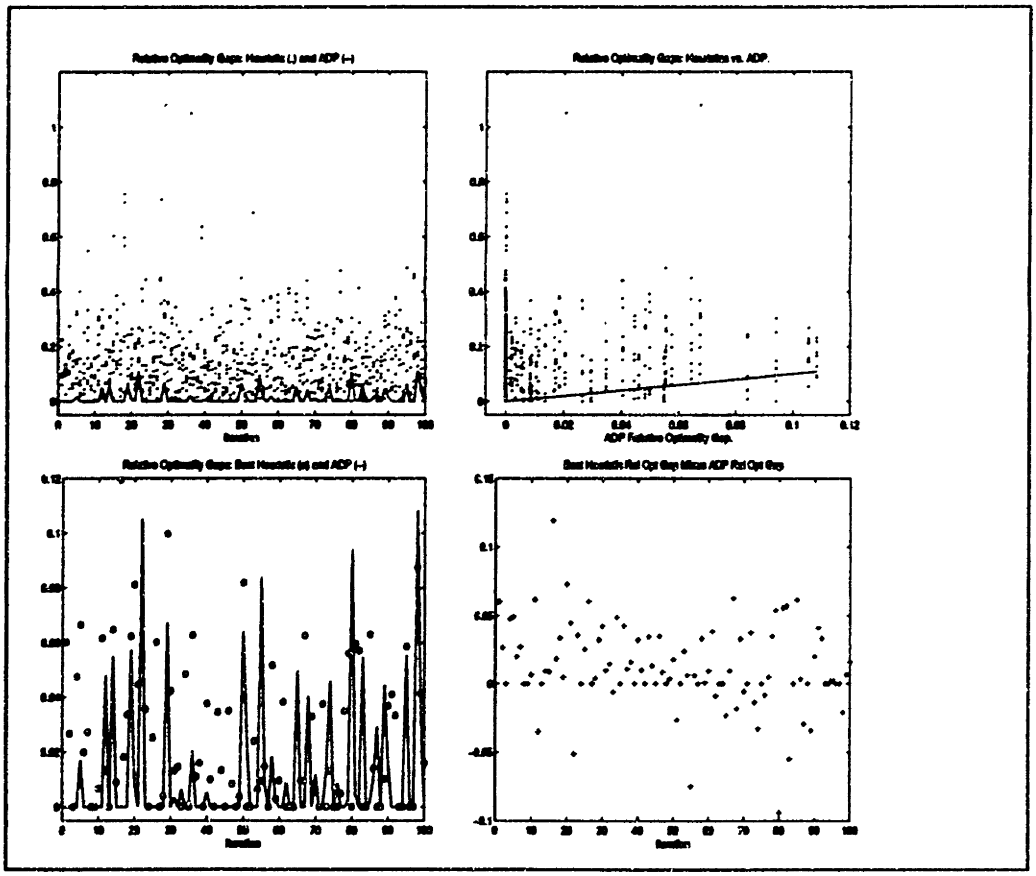


Figure 5-4: These plots compare e-ADP with the ROG's of our heuristics. The first plot compares e-ADP with all heuristic ROG's. We can see that e-ADP does quite well over the group. The second plot (clockwise) demonstrates that same information, but plotted versus the ROG of e-ADP itself, rather than instance number. The third plot is just of the *best* heuristic ROG at each iteration, minus the ROG of e-ADP, and the final plot plots the best heuristic ROG and the e-ADP ROG vs instance number.

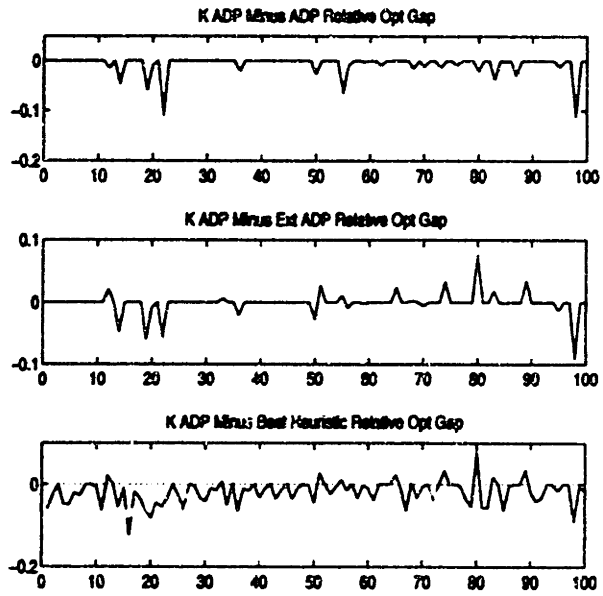


Figure 5-5: The charts above plot the ROG's of κ-ADP minus those for e-ADP, xtd-ADP, and a-H. We see that κ-ADP adds marginal improvement over e-ADP, although the computation time is several times as great.

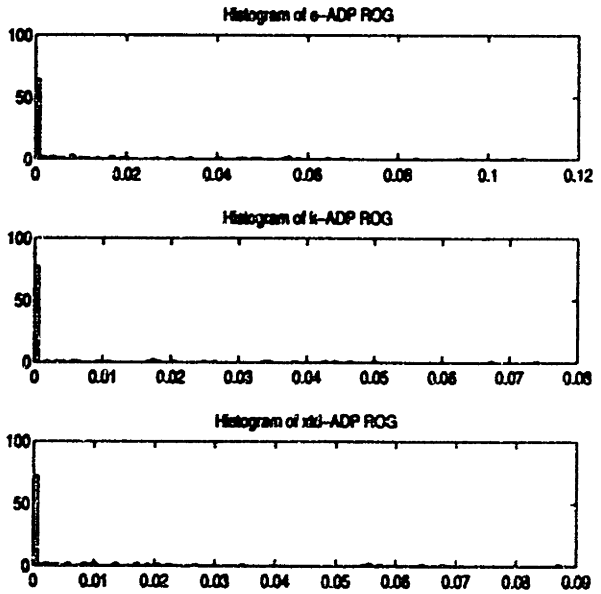


Figure 5-6: We plot histograms of ROG for κ-ADP, e-ADP, xtd-ADP, and a-H. Note that each method generates solutions that are usually optimal, and compare to Figure 5-3.

not optimal.

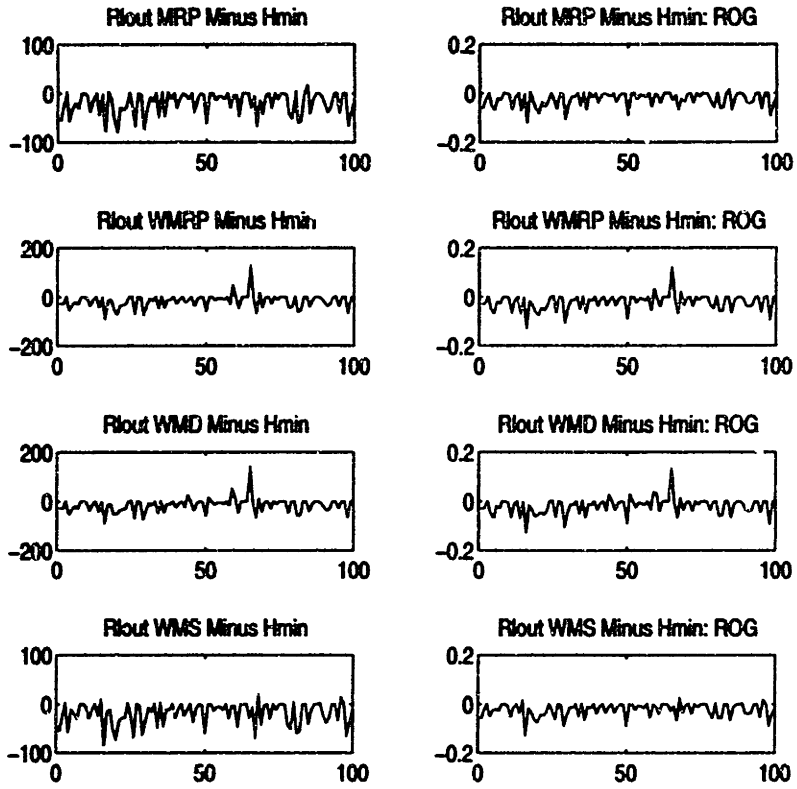


Figure 5-7: Individual Rollouts minus the best heuristic: absolute values and ROG's.

Figure 5-7 plots the performance of each rollout policy against a-H for every problem instance. MRP and WMS rollout beat a-H in all but 3 and four instances, respectively, and even in those cases, the solutions are extremely close. WMD and WMRP rollouts do a bit worse (winning in all but 7 and 4 cases), but in one instance (instance 65), both had ROG's that were 12 points higher than a-H.

Figure 5-8 plots the performance of each rollout policy against e-ADP. The results are similar to those of Figure 5-7. The MRP, WMRP, WMD, and WMS rollouts beat e-ADP in all but 6, 6, 12, and 7 instances, respectively, and no rollout was beaten by e-ADP by a significant margin, with the single exception of instance 65, where both WMRP and WMD performed poorly and were beaten by e-ADP by 9 and 10 points, respectively.

Finally, Figures 5-9 and 5-10 provide a graphical representation of a point we have already made, which is that all rollouts found optimal solutions almost always, and were within a few percentage

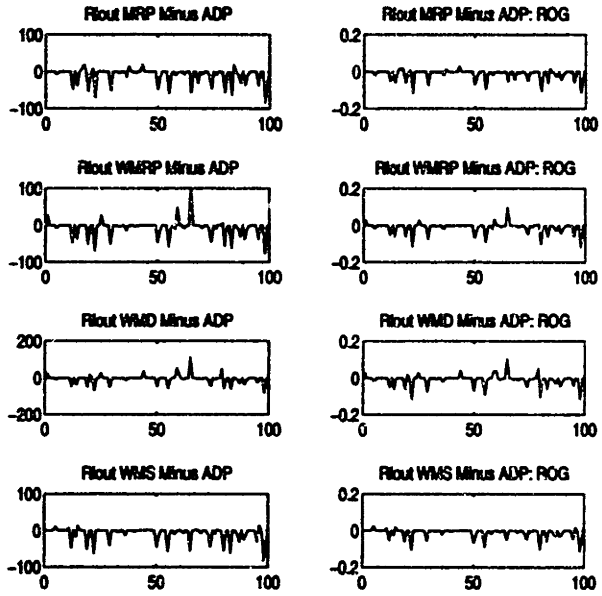


Figure 5-8: Individual Rollouts minus e-ADP, absolute values and ROG's.

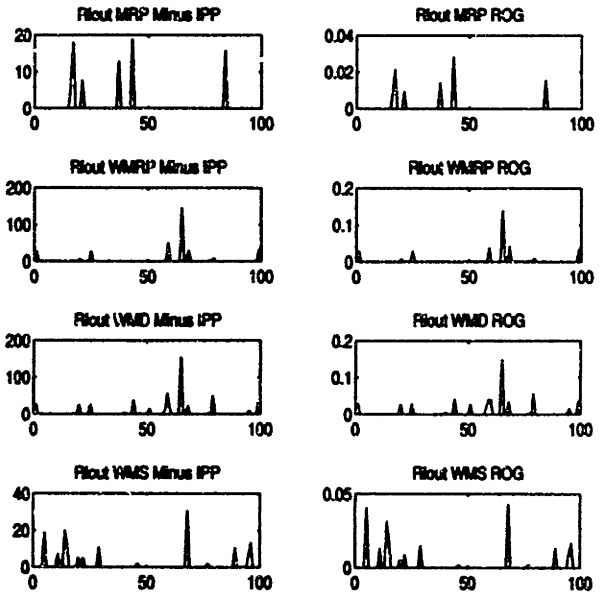


Figure 5-9: Individual Rollouts minus IP: absolute values and ROG's.

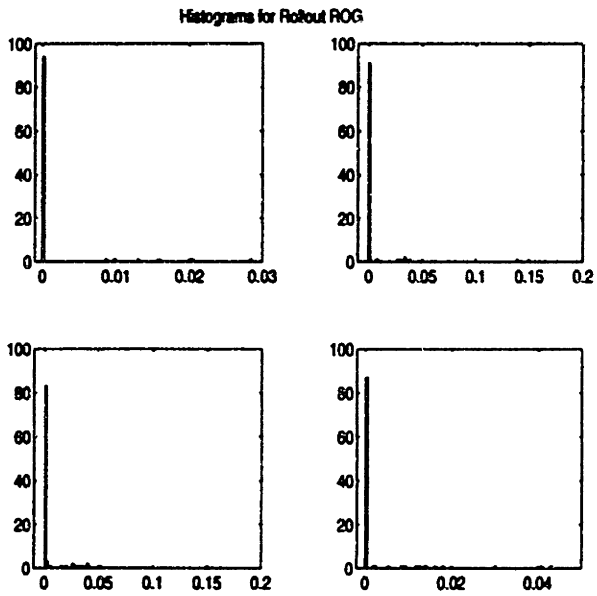


Figure 5-10: We plot histograms of rollout ROG. Note that all our rollout policies give solutions which are essentially optimal.

points of optimal otherwise.

5.2.4 Performance of Approximate Rollout

The performance of our approximate rollout (ARLT) schemes was less impressive. Our linear architecture, under min and mean control, managed to beat e-ADP only twice and once, respectively. Our square min and mean architectures did a bit better, beating e-ADP 10 and 6 times, respectively, and our min and mean neural network architectures did marginally better than that, beating e-ADP 12 and 13 times. In fact, our neural network architecture with mean control was the only one of these six to beat every heuristic in an expected value sense. Figure 5-11 plots the performance of our two neural network architectures against e-ADP.

5.2.5 Mean vs. Min based control policies

One interesting observation is that mean based controls often outperformed min controls. In fact, of the ten architectures which featured both min and mean controls, the mean control did better for six architectures, and often by a very significant margin (7.8% ROG vs. 11.4% ROG, 3.5% vs. 4.0%, 7.4% vs. 12.9%, etc.). By contrast, in the four cases where min control beat mean, the difference

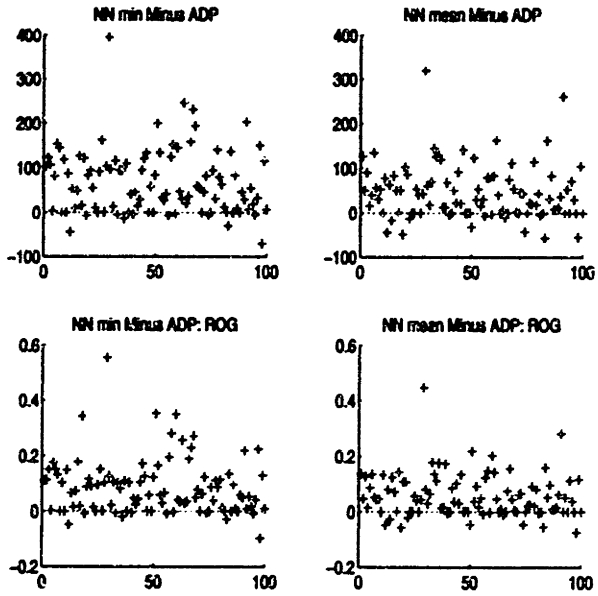


Figure 5-11: We plot the values of ARLT with Neural Net architecture minus ϵ-ADP. We report results for both min and mean control, and report both absolute and ROG differences.

was always less than one point. This somewhat confirms our hypothesis that the rollout costs-to-go are highly correlated, so that a state which is "good" under one heuristic is likely to be good under others as well. It also leads us to believe that our estimates of the rollout cost-to-go do in fact contain some information; otherwise, there should be no systematic difference between min and mean policies.

Figures 5-12, 5-13, and 5-14 plot the differences between min and mean controls for various solution architectures.

5.2.6 Value of Limited Lookahead

The one issue that surprised and dismayed us, even more than the disappointing performance of the ARLT schemes, was the fact that limited lookahead offered no benefit over no lookahead policies, at least for a small number of lookahead stages. In fact, in every single architecture which employed lookahead, the 4 stage lookahead version did significantly worse than its non-lookahead counterpart.

Our explanation for the poor performance of lookahead is that our objective function is trained to appropriately price differences between adjacent states (see the discussion of "finite differences" in Section 4.4.4), but not necessarily between states which are k steps "apart." Note that this

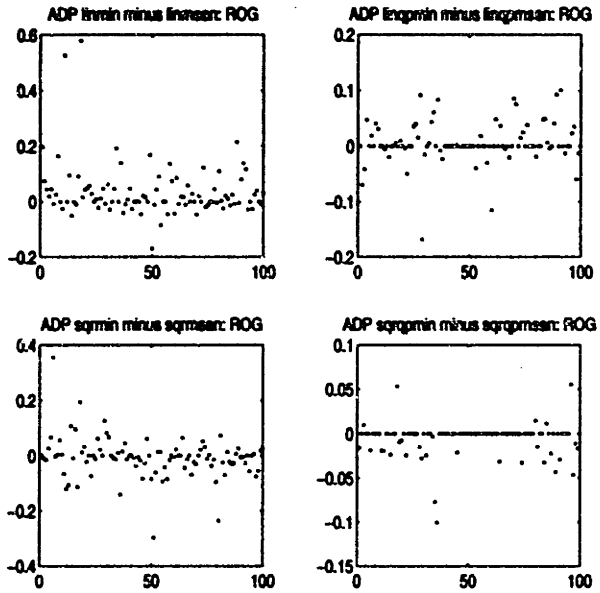


Figure 5-12: Polynomial ADP with no lookahead: min minus mean Relative Optimality Gap

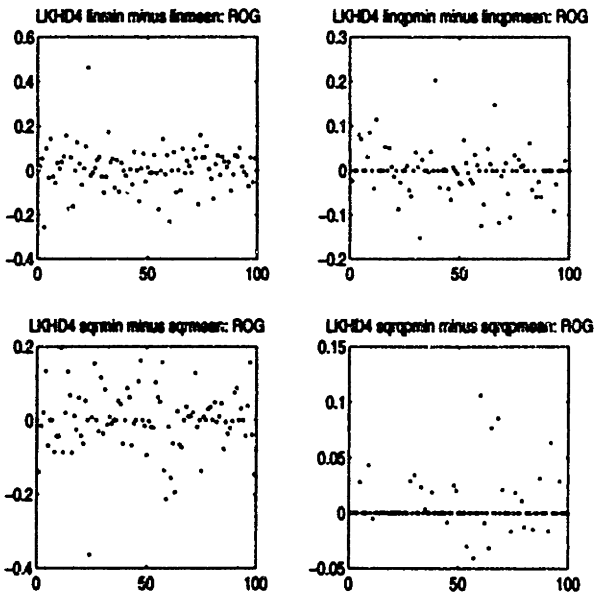


Figure 5-13: Polynomial ADP with lookahead: min minus mean Relative Optimality Gap

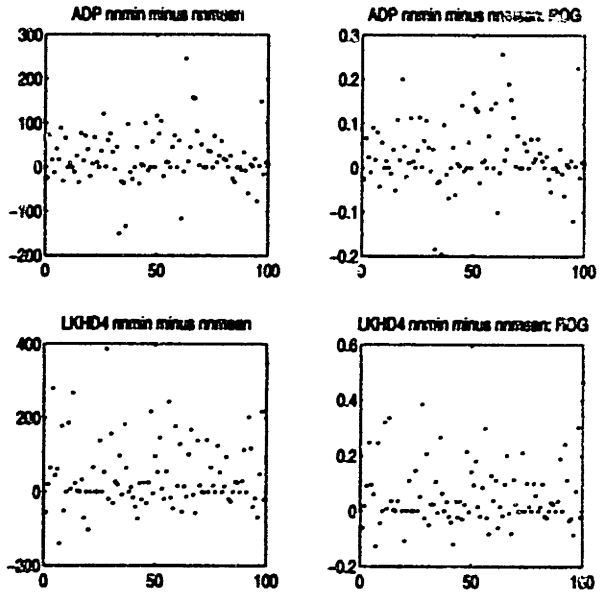


Figure 5-14: Neural Net ADP's with and without lookahead: min minus mean Relative Optimality Gap

"training," involves more than simply parameter weights. The selection and definition of features themselves are tuned to maximize explanatory power between 1-step adjacent states. It is therefore not at all surprising that simply adding lookahead to this same architecture adds no value. On the other had, we certainly believe that had we applied the "Optimization in Policy Space" described in Table 4.1 in a k-lookahead setting from the very beginning, we should easily generate more powerful ADP implementations than our 0-lookahead architecture. Given the already strong performance of 0-lookahead, however, we saw little need to do this.

One interesting observation is that the lookahead "penalty," which we define to be the difference between the value of the lookahead version of a particular ARLT architecture on a scheduling instance and the value of its no-lookahead counterpart, is smaller under the neural network architectures then either the polynomial ones. This phenomenon is illustrated in Figure 5-15. If our conjecture above is correct, this might suggest that the neural networks are doing a better job of estimating finite differences than polynomial architectures.

Moreover, we did observe that 4 step lookahead fairly consistently beat 2 step lookahead with ϵ-ADP, as shown in Figure 5-16.

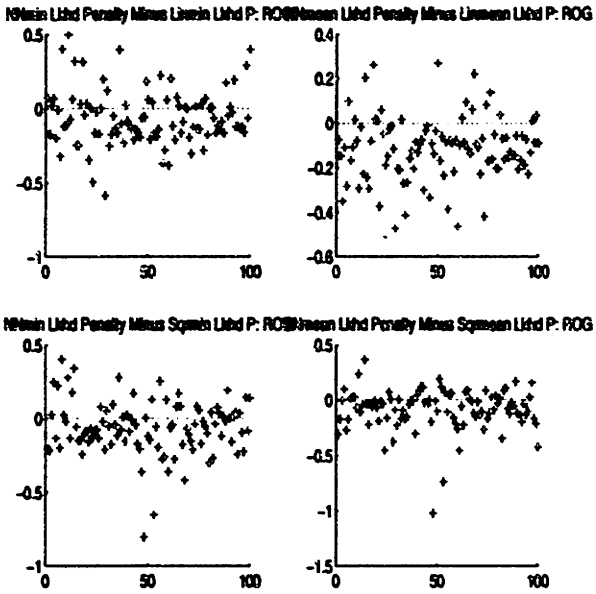


Figure 5-15: We plotted the lookahead "penalty" under the neural net approximations minus the similar penalty under linear approximations. As we suspected, the penalty is less under the neural net approximation, perhaps due to the superior explanatory power of the neural network.

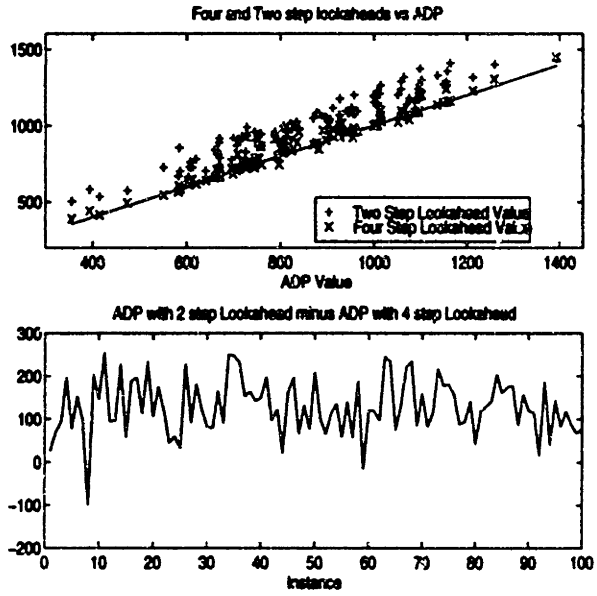


Figure 5-16: We compare the performance of LKHD-2 and LKHD-4 with e-ADP. The latter beats both lookahead policies, which is a phenomenon we saw consistently repeated in all other solution strategies. This surprised and disturbed us. On the other hand, we were pleased to see that LKHD-4 outperformed LKHD-2.

5.2.7 Value of Constraining Polynomial Architecture

Figure 5-17 demonstrates that our constrained polynomial architectures (both linear and quadratic, and min and mean control) fairly consistently beat our unconstrained polynomial ones. In fact, all of our constrained polynomial architectures (with no lookahead) beat every one of our heuristics in an expected value sense by a fairly strong margin (7.4% ROG for best heuristic, vs. 5.3% for worst constrained polynomial).

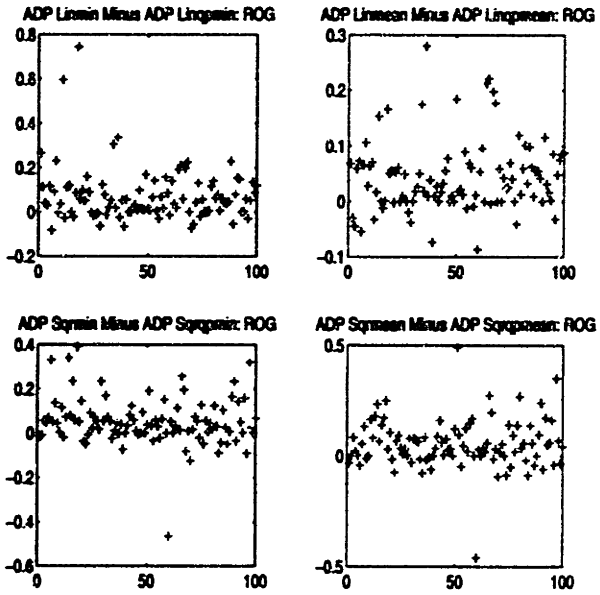


Figure 5-17: We plot the ROG of different ARLT under our four polynomial architectures, and subtract from them the ROG of ARLT under the same architectures, but with constrained coefficients. We notice that constraining coefficients almost universally improved performance.

5.2.8 Fitting Approximation Models

We found that our hypothesis of high correlation between cost-to-go under different heuristic policies for any given state to be qualitatively supported by examination of the optimal polynomial weights, as presented in Figure 5-19. Notice that constraining weights to be less than or equal -1 in the case of earliness features, and greater than or equal to 1 in the case of all others, radically altered the approximation architecture, and had the effect of almost "turning off" a large number of features, especially in the quadratic case.

We also tried "augmented" architectures which added a nonzero intercept, but these made little

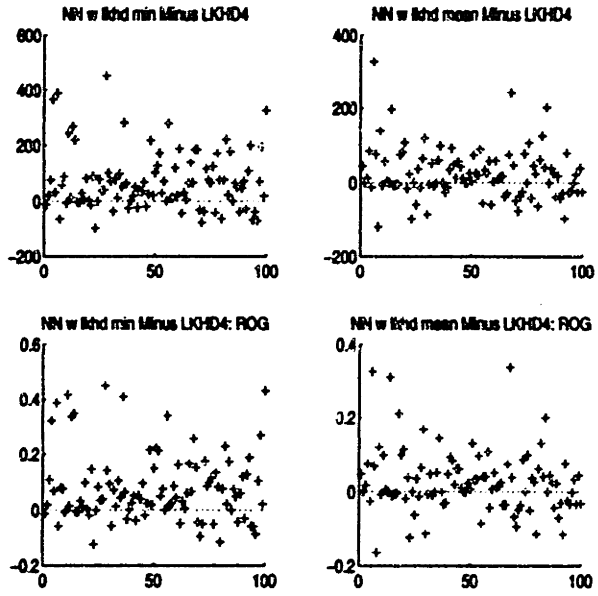


Figure 5-18: We plot the values of ARLT with Neural Net architecture and 4-step lookahead minus LKHD-4. We report results for both min and mean control, and report both absolute and ROG differences

Heuristic	Linear	Quad	Cubic	Constr L	Constr Q
FIFO	20440	10167	7414	27449	33375
	49.656	31.112	30.117	74.267	67.01
mFIFO	23060	11152	8098	31885	37347
	25.369	16.305	13.268	31.389	28.098
LIFO	23060	11152	8098	31884	37347
	25.369	16.305	13.268	31.389	28.098
MD	20501	10164	7402	27473	33536
	48.937	31.174	30.457	74.177	66.787
WMD	20329	9526	6978	27446	38737
	49.325	31.062	24.429	76.407	77.117
MS	18683	9984	7627	24583	32565
	56.628	46.320	49.999	107.42	85.088
WMS	16408	8323	6196	21752	31862
	49.585	40.739	31.839	110.29	89.445
MRP	15336	7482	5423	21169	30709
	56.106	52.481	47.974	129.36	110.66
WMRP	26697	14816	11626	35407	46010
	71.896	45.011	38.873	112.66	118.26
EDD	17798	8975	6682	23582	31679
	57.276	46.152	50.097	111.75	81.090

Table 5.1: We tabulate the prediction quality of each approximation architecture for each heuristic-specific cost-to-go. In each cell, we present both the average squared error and average relative error of our approximations. Average relative error is defined as $\sum_k \frac{|prediction-observation|}{\max(observation,1)\cdot N}$, where N is the number of data observations.

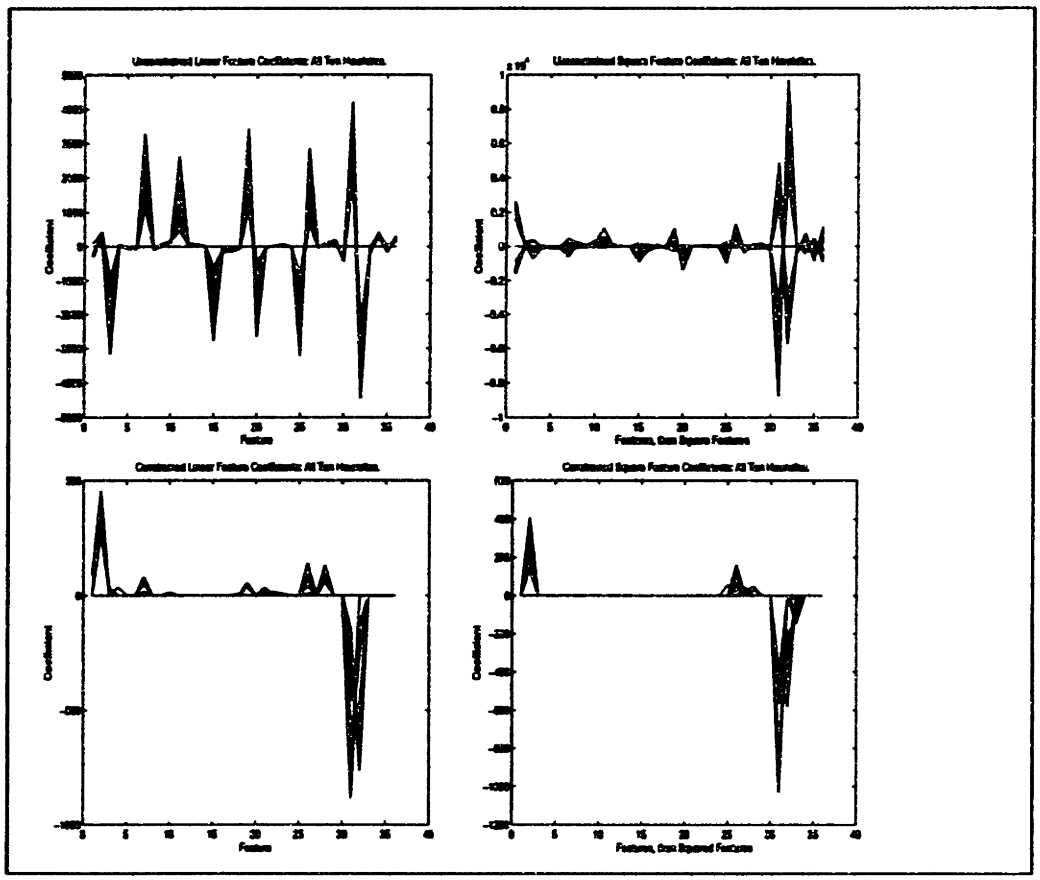


Figure 5-19: We plot the linear and square coefficients for all then heuristics on the top plots. On the bottom plots, we plot the same statistics for the constrained case. Notice that constraining coefficients significantly alters the solution, turning only a few heuristics into "predictor" heuristics. Notice also that the coefficients demonstrate significant correlation between heuristic costs-to-go.

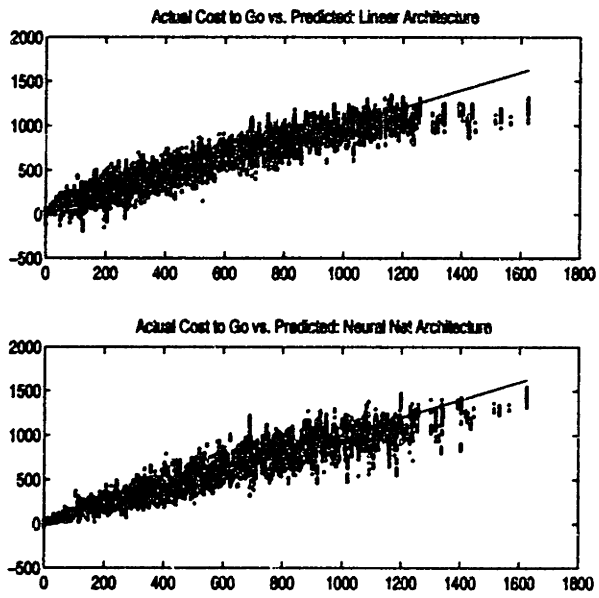


Figure 5-20: We examine the predictive quality of our linear and neural network architectures for FIFO cost-to-go data. Each predicted cost-to-go is plotted as a function of the underlying cost-to-go of its state.

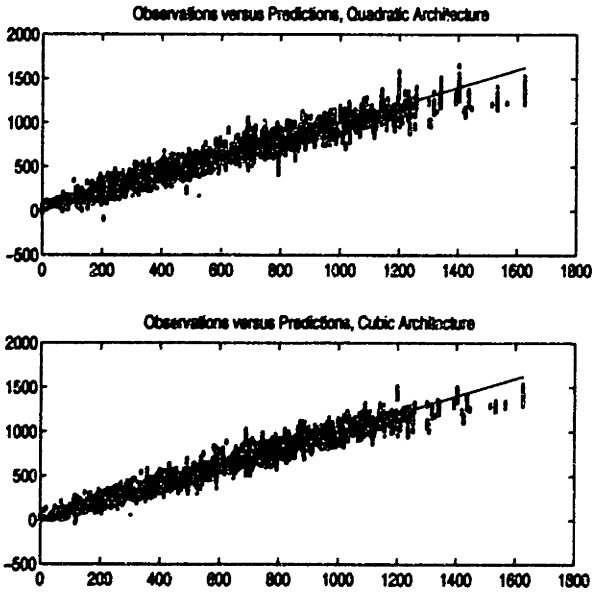


Figure 5-21: We examine the predictive quality of our square and cubic architectures for FIFO cost-to-go data. Each predicted cost-to-go is plotted as a function of the underlying cost-to-go of its state.

Heuristic Rollout	Ave Squared Error	Ave Relative % Error
FIFO	19264	25.749
mFIFO	12749	19.962
LIFO	15450	16.185
MD	18983	25.671
WMD	15118	38.963
MS	12717	25.838
WMS	12258	37.218
MRP	19899	61.257
WMRP	24126	51.397
EDD	17055	44.167

Table 5.2: We list the average squared error and average relative error of our neural net approximations for each heuristic policy. Average relative error is defined as $\sum_k \frac{|prediction-observation|}{\max(observation,1)\cdot N}$, where N is the number of data observations.

difference in the other weights or the power of approximation. Tables 5.1 and 5.2 and Figures 5-20 and 5-21 summarize the quality of our predictions.

5.2.9 Training Neural Networks

We used the principal components analysis of Section 4.4.2 to reduce the dimension of our input vector from 36 to 14. Table 5.2 summarizes the power of our approximation, and Figures 5-22, 5-23, and 5-24 summarize some typical solution histories for the neural networks. As we can see, the neural networks provided better approximations which lead to slightly better policies, but the training of the neural networks was significantly more difficult, taking several hours per heuristic.

5.3 Comparison of all solution methods

We assemble a table indicating important statistics for each of our forty-two solution approaches. For each, we calculated the mean ROG over the 100 problem instances, as well as the standard deviation and range of these statistics, where range is defined to be the value of the largest minus the smallest ROG. Finally, we also indicate the number of instances of the 100 in which each approach found the optimal schedule. These data are presented in Table 5.3. Note that r-H performed significantly worse than every single other approach, which was gratifying to see. A comparison of r-H with other methods is provided in Figure 5-25.

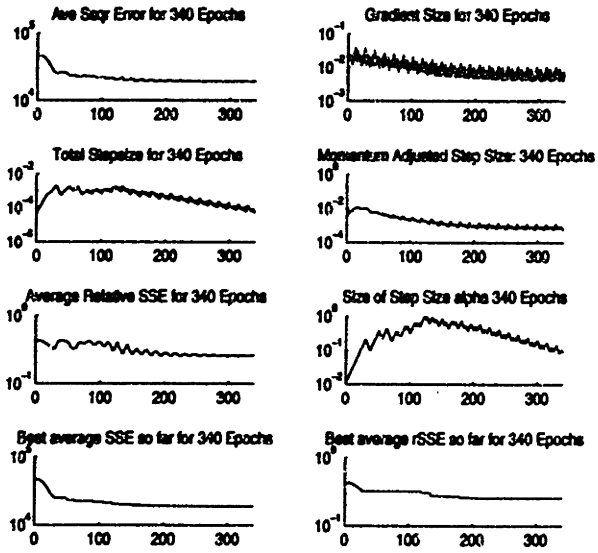


Figure 5-22: We display a record of Procedure 5 for WMD features. An "epoch" here corresponds to one iteration of the outermost loop of Procedure 5. Moreover, normal gradient descent was utilized.

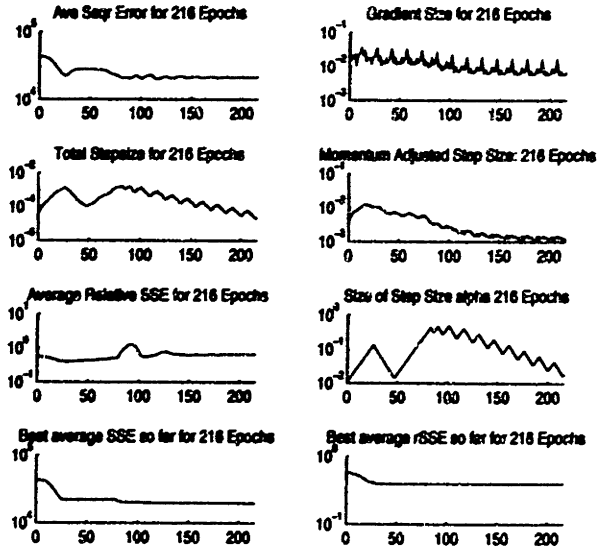


Figure 5-23: We display a record of Procedure 5 for MRP features. An "epoch" here corresponds to one iteration of the outermost loop of Procedure 5. Moreover, normal gradient descent was utilized.

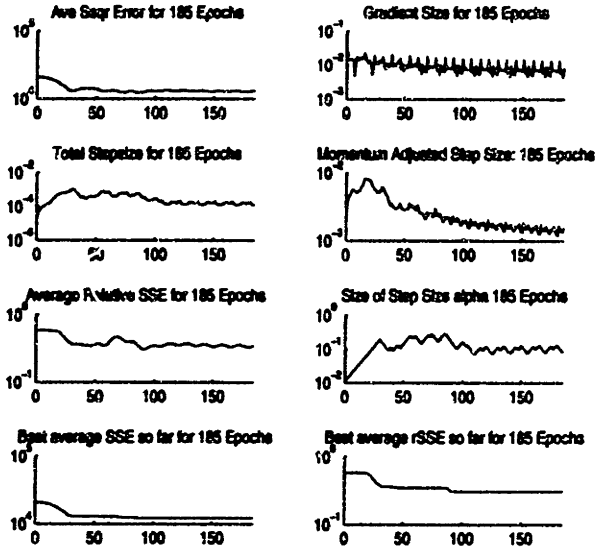


Figure 5-24: We display a record of Procedure 5 for WMS features. An "epoch" here corresponds to one iteration of the outermost loop of Procedure 5. Moreover, normal gradient descent was utilized.

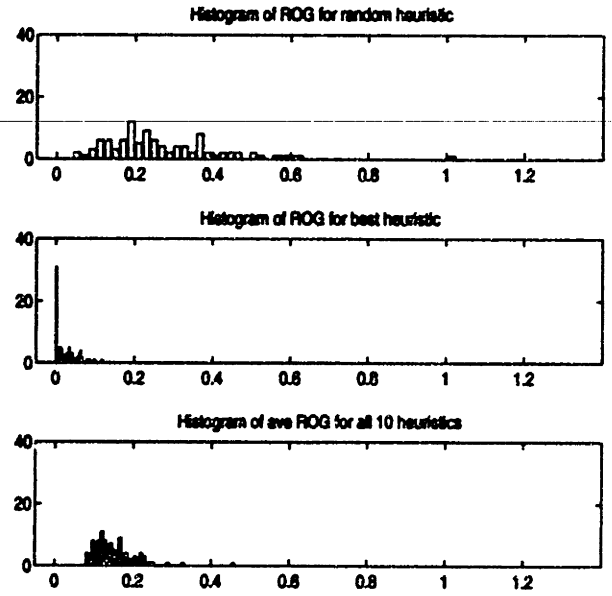


Figure 5-25: We plot histograms of ROG for r-H, the average ROG over all h, and for a-H. Notice how powerful a-H is. Notice also how poorly r-H performs, even compared to the average over all h. This suggests that each heuristic has an internal "consistency" which is destroyed when mixing and matching them in an unintelligent manner.

Solution Method	Average ROG	STD ROG	Range ROG	% Optimal
FIFO	0.1999	0.06443	0.3118	0
mFIFO	0.2423	0.09778	0.5451	0
LIFO	0.2562	0.1106	0.6439	0
MD	0.1992	0.06418	0.3118	0
WMD	0.08107	0.06915	0.4516	4
MS	0.1402	0.1057	0.7487	0
WMS	0.1135	0.101	0.7247	1
MRP	0.07339	0.0543	0.2429	0
WMRP	0.1008	0.1871	1.08	26
EDD	0.1222	0.1024	0.7487	0
r-H	0.2685	0.1499	0.9816	0
a-H	0.02456	0.02701	0.1194	31
e-ADP	0.01234	0.02466	0.1083	64
κ-ADP	0.006142	0.01483	0.07428	77
ADP-Lin Mean	0.0786	0.07781	0.3422	13
ADP-Lin Min	0.1141	0.1268	0.7444	1
ADP-Lin Constr Mean	0.03485	0.0541	0.3399	39
ADP-Lin Constr Min	0.04048	0.0483	0.2043	35
ADP-NN Mean	0.06953	0.0793	0.5154	35
ADP-NN Min	0.09726	0.1023	0.6222	36
ADP-Sqr Mean	0.1075	0.1093	0.5291	8
ADP-Sqr Min	0.1003	0.1085	0.486	11
ADP-Sqr Constr Mean	0.05291	0.07215	0.4651	35
ADP-Sqr Constr Min	0.04745	0.06932	0.4651	36
xtd-ADP	0.00696	0.01698	0.0875	72
LKHD-2	0.2096	0.08425	0.4488	0
LKHD-4	0.04648	0.05048	0.2769	27
LKHD-4-Lin Mean	0.198	0.1048	0.4719	2
LKHD-4-Lin Min	0.2083	0.1081	0.6142	1
LKHD-4-Lin Constr Mean	0.07029	0.05997	0.3002	16
LKHD-4-Lin Constr Min	0.06887	0.06206	0.3002	16
LKHD-4-NN Mean	0.07444	0.07701	0.3618	20
LKHD-4-NN Min	0.1291	0.1225	0.4741	13
LKHD-4-Sqr Mean	0.1966	0.1681	1.052	0
LKHD-4-Sqr Min	0.1944	0.1586	1.056	4
LKHD-4-Sqr Constr Mean	0.08871	0.07483	0.4058	14
LKHD-4-Sqr Constr Min	0.09352	0.07502	0.4058	12
MRP Rlout	0.0009639	0.004186	0.02857	94
WMRP Rlout	0.003241	0.01564	0.1398	91
WMS Rlout	0.002142	0.007287	0.0433	87
WMD Rlout	0.005467	0.01321	0.1494	83
Best Rlout	0	0	0	100

Table 5.3: We list the average relative optimality gap for 100 instances for each of our solution approaches, as well as two estimates of the spread of this gap: its standard deviation and range.

5.4 Conclusions and Insights

There appear to be three dimensions to a successful formulation of the ADP described in Chapter 4. These three dimensions are, first, the quality of the heuristics defining the control space, second, the quality of the features describing the state space, and finally, the quality of the architecture used to map feature vectors to cost to go approximations.

Our experiments seem to indicate that our heuristics are modestly good, sometimes coming within a few percentage points of the optimal solution. In fact, each heuristic has special features and qualities which might explain some of the results seen here. Moreover, the heuristics themselves seem to adequately "span" the control space, in the sense that, when we've analyzed individual planning scenarios on a move-by-move basis and compared the ADP control decisions with the optimal ones, there was always some heuristic which mirrored the optimal, one-step control at any particular state i. All this leads us to believe that the key bottleneck to ADP in our application is the generation of a cost-to-go approximation that will properly distinguish between adjacent states. We were more successful at this when applying expert intuition to effectively perform *optimization in policy space* (*e*-ADP), or simulation to get *exact* estimates of heuristic-specific costs-to-go (rollouts), then when we attempted a more passive, data-driven cost-to-go approximation.

5.4.1 Information Paradox

We also observed the following interesting phenomenon. Consider the scheduling scenario of Figure 5-26. The optimal control is to send any one of the trains into the available process. However, consider what happens once we do that. Since Outbound Inspection is a process which takes 4 time units, the lateness statistics for each of the other four trains is increased by 4. On the other hand, if we simply hold each train in place and keep the Outbound Inspection crew idle, the lateness statistic for each train increases only by 1, since in principle each train could be the one to begin inspection the following period. The chosen control, therefore, may very well be to hold each train in the departure yard indefinitely! What is happening is that our lateness statistics are consistent *underestimates* of true lateness. When the inspection crew is idle, we assign lateness to each train on the assumption that it will be the next train to be inspected, which is of course accurate for only one train, although we don't yet know which one. Our lateness therefore underestimates the true lateness for each train. Once a train is assigned to outbound inspection, we have new information which we use to

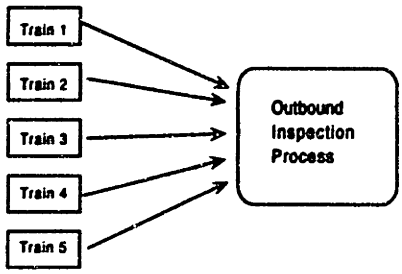


Figure 5-26: Trains 1, 2, 3, 4 and 5 are all available for outbound inspection, which takes 4 time periods. Each train has the same departure deadline and priority.

update the lateness statistics for all other trains by 4, bringing them closer to reality. Even this new statistic, however, is an underestimate, and will be accurate only for the second train to enter inspection. However, in feature space, it appears as though we have increased the lateness of the system.

This, we feel, is a general problem with features that tend to underestimate underlying parameters, and where that underestimate is a function of the amount of "information" in the state. Controls that tend to increase information (such as dedicating processes to jobs) may often increase the estimates of these features. Our ADP will therefore avoid such controls, where possible, which may produce a systematic bias away from optimal decisions.

5.4.2 Stochastic vs. Deterministic Problems

It has been empirically observed that that neuro-dynamic or reinforcement learning methods are much more successful at stochastic control problems (backgammon [73], tetris [76], machine repair [14], or inventory control [78]) than deterministic control problems (such as chess, or LMPM and YSSM). A common explanation for this phenomenon is that stochasticity creates "smoothness" in the expected cost to go function (in feature space), which alleviates the difficulties inherent in trying to globally estimate discontinuous functions with smooth ones while at the same time not losing the important characteristics of these discontinuous functions in the tiny regions that drive the control decisions. We have certainly observed such issues in our own computational research. However, perhaps there is a second issue as well. In stochastic control, the criteria for "success" are different. There, a successful ADP implementation need only beat alternative strategies in an *expected value* sense. Therefore, an ADP which beats a heuristic on 5500 out of 10000 instances would be declared a success if we are dealing with a stochastic control problem, while not necessarily so for a deterministic

one. The reason, of course, is that in a deterministic setting we have the opportunity to apply the heuristic on the 4500 problems in which it did well, and do something else on the other 5500. We can, in other words, observe ω, then choose our heuristic. This is exactly what a-H is, which is what made it so difficult to beat. In the stochastic domain, there is no equivalent to a-H. We can only consider the aggregate performance of each heuristic individually. On that measure, however, most of our architectures did quite well. This is perhaps another sense in which deterministic problems are "harder."

5.4.3 Comparisons with Exact Optimization

While our different execution strategies for ADP have different computational characteristics, they are all significantly superior to the IP. They do not explode with problem size, but grow linearly (or quadratically, in case of rollout) with time horizon. Also, the computation times are relatively insensitive to schedule congestion, while IP times are highly non-robust in this statistic.

Moreover, the ADP formulation is also much more flexible than IP, and can lead to formulations of much higher fidelity. Significant issues such as locomotive maintenance at terminals or the scheduling of M.O.W. crews and helper engines on lines can all be integrated into the ADP formulation without a significant growth in complexity of the model (although not without a growth in data complexity of the state definition!). In the IP, on the other hand, all these issues had to be explicitly excluded from the formulation lest it become intractable.

For all these reasons, we are very optimistic about the applicability of ADP to complex, large scale scheduling problems.

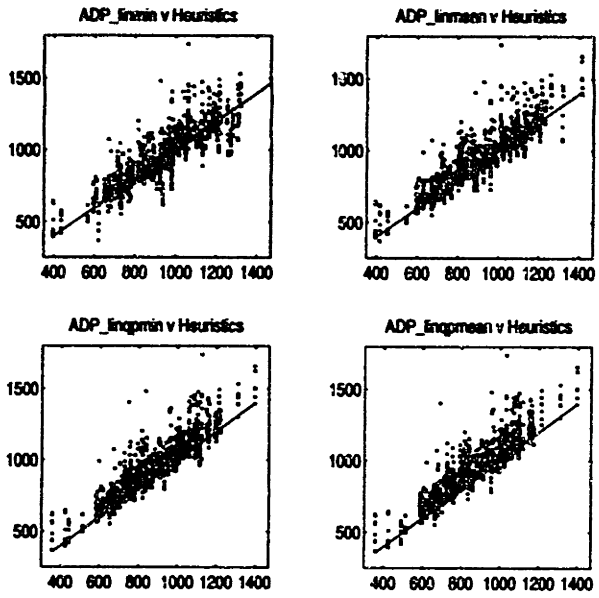


Figure 5-27: ARLT with linear architectures versus heuristics. Notice that our constrained architectures significantly outperform the unconstrained ones.

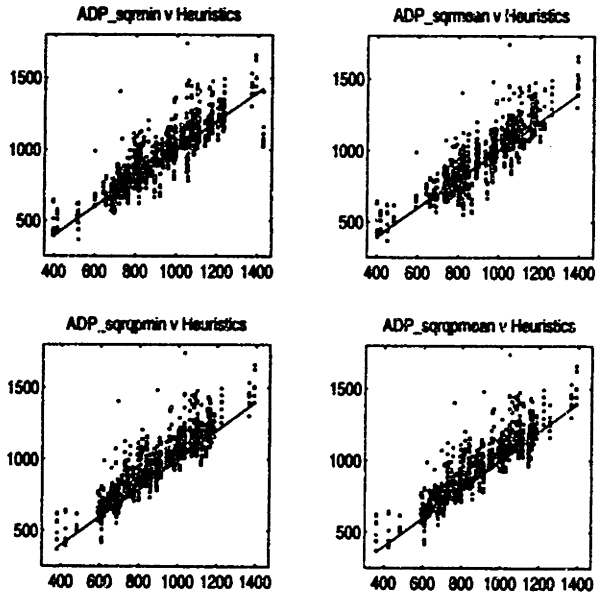


Figure 5-28: ARLT with quadratic architectures versus heuristics. Notice that our constrained architectures significantly outperform the unconstrained ones.

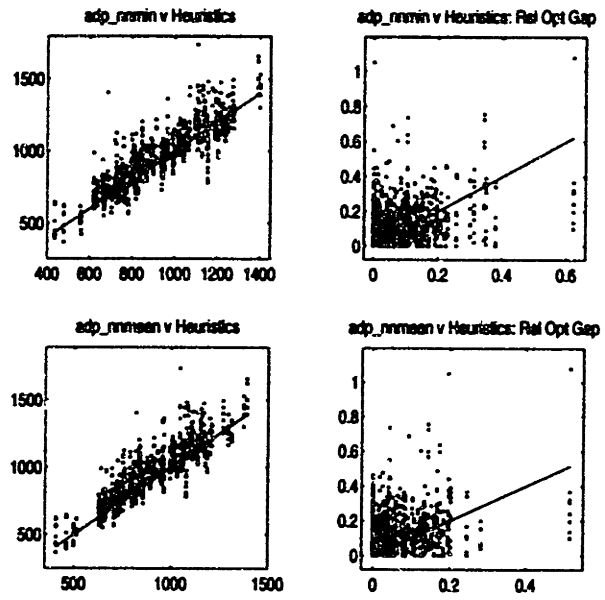


Figure 5-29: We plot values of ARLT with no lookahead utilizing a Neural Network cost to go approximation. We show both absolute and relative optimality gaps, and compare with our battery of heuristics. Both min and mean control policies are utilized.

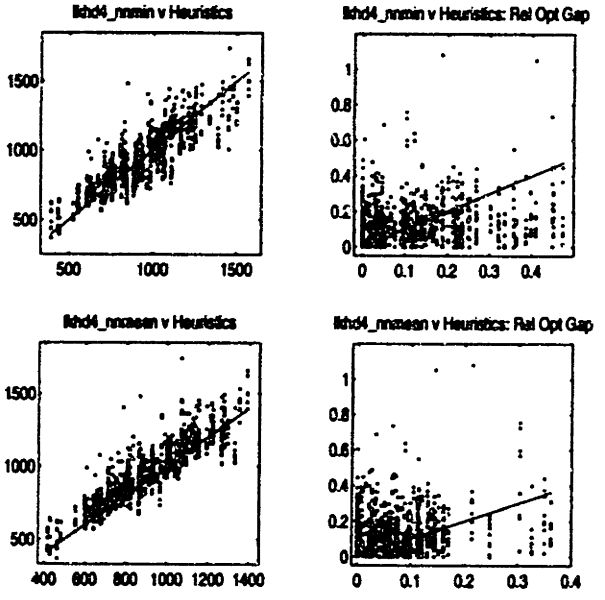


Figure 5-30: We plot values of ARLT with lookahead utilizing a Neural Network cost to go approximation. We show both absolute and relative optimality gaps, and compare with our battery of heuristics. Both min and mean control policies are utilized.

Chapter 6

Conclusion and Summary of Contribution

In this last chapter, we briefly summarize the work presented in this thesis, outline what we see as its major contributions and our accomplishments, and state again some of our most important conclusions.

6.1 Summary of Research

This thesis presented a generalization of the job shop scheduling problem called the Multiprocessor Network Scheduling Problem (MNSP), and a specialization which we called MNSP*.

We saw that using MNSP and MNSP*, we can model a broad set of scheduling problems for the railroad, and introduced RNSP as a special case of MNSP, and three other problems, LMPM, YSSM, and YDM, as special cases of MNSP*.

We saw that in order to make use of RNSP, or any derivation of MNSP, we would need to specify the functional relationship between traffic composition and processor speed at any given node. Chapter 2 discusses how one might use *computer aided dispatching* (CAD) tools in the rail domain to generate such a relationship, and how one can then decompose an instance of MNSP and RNSP into independent planning problems utilizing these planners. This decomposition strategy demonstrated several important features:

1. The fidelity of MNSP could be as high as the fidelity of our node planners, and the performance of our decomposition strategy was independent of this.

2. The decomposition strategy empirically produced feasible, but not necessarily optimal, solutions to MNSP, at least in the case of RNSP.

3. The quality of solutions to MNSP could be affected by suitable use of a "target schedule," which need not itself be a feasible solution to MNSP.

Our next objective was to find a way to get such a target schedule, and our approach was to formulate a lower-fidelity version of RNSP, which happened to correspond to an instance of MNSP in which each node is itself an instance of MNSP*. We did so in two ways.

Our first approach was to borrow a highly successful integer programming model of the Airport Ground Holding Problem (GHP) by Bertsimas and Stock [15]. This formulation was successful because of one very important feature, which is that many of the constraints in the formulation defined facets of the convex hull of integer solutions of its feasible region. The formulation, however, in its essence solves MNSP*, not just GHP. We therefore formulate versions of YSSM, LMPM, and YDM as Bertsimas/Stock variations of MNSP*. We call the original IP formulation of MNSP* the Generic Bertsimas Stock model (GBS).

In solving these problems, we encountered several problems not seen by Bertsimas and Stock in GHP. As discussed in Section 3.6, the linear relaxations of our models were never integral. We therefore had to apply significant effort to generate integer optimal solutions, and this effort generated several contributions of our own to the original GHP model. These are:

1. Observation and exploitation of an underlying network structure of the LP relaxation of GBS.

2. Discovery of a branching strategy which dramatically accelerated the solution speed of GBS. This branching strategy involved the selective relaxation of all but a small subset of integer variables, and the branching of the remaining integer variables in a specified order and direction.

3. Implementation of a decomposition strategy which empirically demonstrates both strong duality, and compatibility with a price-generation rule known as *Everett's Method*.

In Section 3.7, we present computational experiments in which we solve to integer optimality realistically sized instances of YSSM, LMPM, and a network scheduling problem consisting of intercon-

nected instances of YDM. To our knowledge, it is the first time any formulations of such railroad scheduling problems of realistic size are solved to optimality.

While our computational results represent an important step forward of the state of the art for such scheduling problems, they also reveal significant weaknesses of our IP formulation. The most important is that the solution time of our model is highly sensitive to two factors, problem size and schedule congestion. For this reason, we suggest that our IP model is appropriate under conditions which include:

1. Fairly homogeneous line topology, and
2. Low to moderate traffic intensity, and
3. Low expected delay times,

or

1. Off-line, batch scheduling is all that is needed and
2. Computational resources are significant.

Otherwise, we feel that sub-optimal, heuristic approaches are more appropriate, and this leads us to the final stage of our research.

Chapter 4 introduces a battery of heuristics for the solution of MNSP*, from extremely simple heuristics such as FIFO or LIFO, to much more complicated heuristics involving moderate computation to implement. The heuristics are described in Section 4.2. It also introduces the DP formulation of MNSP*, as well as an approximate DP which coordinates the above scheduling heuristics in a very novel way. Namely, we partition time into discrete intervals, and limit our control decisions to apply only one heuristic during each of these intervals. Our new DP therefore involves selecting a sequence of scheduling heuristics to apply over the time-horizon of our scheduling problem. To select an appropriate heuristic at each stage, we simulate each heuristic over the next time period, then "price" the resulting state with an estimation function, as described in Section 4.1.1. This estimation function utilizes a feature extraction mapping which accumulates important features of a state and then uses these features to compare the results of several heuristics and choose one. This approximation architecture is described in Section 4.3, and the features relevant to MNSP* are enumerated and defined in Section 4.3.1. There are many different ways we might use features to

select heuristics, and the rest of Chapter 4 discusses these and other practical issues involved with the implementation of each approach.

Finally, we present in Chapter 5 a computational case study of heuristic-based ADP by solving realistic instances of YSSM. We see in a battery of 100 test problems that heuristic ADP and many of its variants can find solutions that average within 1.2% of optimal within a few minutes of CPU time. These computation times were roughly the same for all instances of YSSM that we tested, even those which the IP formulation of Chapter 3 were completely unable to solve. Moreover, by expending a bit more computational effort, the quality of our solutions can be significantly increased, getting to within an average of 0.1% of optimality for some methods (see Section 5.3). Finally, we point out that, unlike the case of our IP formulation, our DP formulation is amenable to dramatic increases in fidelity and complexity without necessarily worrying about crossing the border of tractability. We ended the thesis by concluding that ADP-based heuristic coordination methods offer an exciting opportunity to solve many of the network-based scheduling problems which have proven so challenging for the last several decades.

6.2 Statement Of Contributions and Conclusions

The first contribution of this thesis is that we've reformulated the MNSP-feasibility problem as a fixed-point problem utilizing "node planners," applied it to RNSP, and so have presented a very generic decomposition approach to solving such problems involving what we call MSA. It is an approach we have not seen before in the literature. We are especially excited about this approach since its main assumption, that we have access to functional node planners, is quite likely to be satisfied, not just in our rail applications, but in other domains as well. In job shop scheduling, for example, single machine scheduling problems, even those which are Strongly $\mathcal{NP}$-Hard, are solved quite effectively in practice. We conclude, therefore, that MSA is an empirically promising approach to generating feasible schedules for RNSP, and potentially for other applications of MNSP as well.

The second contribution is our successful extension of the Bertsimas/Stock model to three different variations of MNSP*, all of which manifested properties that made them more difficult to solve than GHP. In the course of implementing these extensions, we faced and overcame a host of challenges which lead to contributions in their own right: exploiting network structure, implementing Lagrange decomposition and Everett's price update rule, and specification of an effective branching

strategy. These contributions should all, we hope, make it feasible to extend the Bertsimas/Stock model to a much broader set of applications than was previously possible. We conclude that our IP formulation of LMPM, YSSM, and YDP are reasonable solution approaches under the conditions outlined in the previous section, although under different conditions, other, sub-optimal approaches might be more appropriate.

The final set of contributions involve the heuristic solution of MNSP*. First, the heuristics themselves, some of which were quite detailed, are contributions. Secondly, we've implemented, to our knowledge, the first formal heuristic-coordination strategy for scheduling problems using ADP. We feel that the coordination of heuristics is an extremely promising way to solve a diverse range of problems which, because of an excessively large (often uncountable) control space, have often not been considered candidates for ADP-type methods. Finally, our computational experiences demonstrate strongly that ADP techniques, of some form or another, can be quite effective at getting extremely good solutions in computation times which are much smaller than those of more standard optimization methods. Moreover, we hope that our computational experiments, as well accompanying discussion, emphasizes the potential utility of expert-motivated optimization in policy space over the more passive, data-driven approximation methods which are perhaps considered more elegant and have been favored in the research community. Finally, our results with rollout methods reinforce the existing empirical evidence that exact or "almost-exact" single policy iteration via rollouts or other accurate simulation techniques, while computationally expensive, are some of the most powerful tools in the ADP arsenal today.

Bibliography

[1] Dharma Acharya, Operations Research Group, CSX Transportation, Inc., *Personal Communication*, July, 1996.

[2] Joseph Adams, Egon Balas, Danial Zawack, "The Shifting Bottleneck Procedure for Job Shop Scheduling," *Management Science*. Volume 34, No. 3, 1988.

[3] David Applegate, William Cook, "A Computational Study of the Job-Shop Scheduling Problem." *ORSA Journal on Computing*. Volume 3, No. 2, 1991.

[4] Andrew P. Armacost, "Modeling Railroad Terminal Operations: Supporting Real-Time Network Planning and Control", S.M. Thesis, Operations Research Center, Massachusetts Institute of Technology, and C.S. Draper Laboratory, Inc., June, 1995.

[5] Arjang A. Assad, "Models for Rail Transportation," *Transportation Research A*. Volume 14A, pp. 205-220, 1980.

[6] Arjang A. Assad, "Modelling of Rail Networks: Toward a Routing/Makeup Model," *Transportation Research B*. Volume 14B, pp. 101-114, 1980.

[7] Egon Balas, "Machine Sequencing via Disjunctive Graphs: An Implicit Enumeration Algorithm," *Operations Research*. Volume 17, pp. 941-957, 1969.

[8] Egon Balas, "On the Facial Structure of Scheduling Polyhedra," *Mathematical Programming Study 24*. pp. 179-218, 1985

[9] A. G. Barto, S. J. Bradtke, S. P. Singh, "Learning to Act Using Real-Time Dynamic Programming," Artificial Intelligence, Volume 72, pp. 81-138, 1995.

[10] Lawrence D. Bodin, Bruce L. Golden, Allan D. Schuster, and William Romig, "A Model for the Blocking of Trains," *Transportation Research B.* Volume 14B, pp 115-120, 1980.

[11] Dimitri P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*, Academic Press, 1982.

[12] Dimitri P. Bertsekas, *Nonlinear Programming.* Athena Scientific, 1995.

[13] Dimitri P. Bertsekas, John N. Tsitsiklis, *Neuro-Dynamic Programming.* Athena Scientific, 1996.

[14] Dimitri P. Bertsekas, John N. Tsitsiklis, Cynara Wu, "Rollout Algorithms for Combinatorial Optimization," Laboratory for Information and Decision Systems Report LIDS-P-2386, Massachusetts Institute of Technology, 1997.

[15] D. Bertsimas and S. Stock, "The Air Traffic Flow Management Problem with Enroute Capacities," August 1994, M.I.T. Working Paper OR 299-94. To appear in *Operations Research.*

[16] Dimitris Bertsimas, *Personal Communication*, February, 1995.

[17] Dimitris Bertsimas, Ioannis Ch. Paschalidis, John N. Tsitsiklis, "Optimization of Multiclass Queueing Networks: Polyhedral and Nonlinear Characterizations of Achievable Performance," *The Annals of Applied Probability.* Volume 4, No. 1, 1994.

[18] Border, Kim C., *Fixed Point Theorems With Applications to Economics and Game Theory*, Cambridge University Press, 1985.

[19] Peter Brucker, Bernd Jurisch, Bernd Sievers, "A branch and bound algorithm for the job-shop scheduling problem," *Discrete Applied Mathematics.* Volume 49, pp. 107-127, 1994.

[20] X. Cai and C.J. Goh, "A Fast Heuristic for the Train Scheduling Problem," *Computers in Operations Research.* Volume 21, Number 5, 1994.

[21] J. Carlier, E. Pinson, "An Algorithm for Solving the Job-Shop Problem," *Management Science.* Volume 35, No. 2. 1989.

[22] J. Carlier, E. Pinson, "Adjustment of heads and tails for the job-shop problem," *European Journal of Operations Research.* Volume 78, pp. 146-161, 1994.

[23] Bin Tong Chen, Patrick Harker, "Two Moments Estimation of the Delay on Single-Track Rail Line with Scheduled Traffic." *Transportation Science*, Vol. 39, No. 1, 1990.

[24] James Christodouleas, Efthalia Chryssikou, "A Decomposable Integer Programming Formulation for the Line Meet-Pass Planning Problem," C.S. Draper Laboratory Working Paper, July 10, 1994.

[25] Teodor Crainic, Jacques A. Ferland, Jean-Marc Rousseau, "A Tactical Planning Model for Rail Freight Transportation," *Transportation Science*. Volume 18, No. 2, 1984.

[26] Teodor Crainic, Jean-Marc Rousseau, "Multicommodity, Multimode Freight Transportation: A General Modeling and Algorithmic Framework for the Service Network Design Problem," *Transportation Research B*. Volume 20B, No. 3, 1986.

[27] Robert H. Crites, Andrew G. Barto, "Improving Elevator Performance Using Reinforcement Learning," in *Advances in Neural Information Processing Systems 8*. MIT Press, pp. 1017-1023, 1996.

[28] CPLEX Optimization, Inc. Using the CPLEX Callable Library, v. 3.0. 1994.

[29] Carlos F. Daganzo, "Dynamic Blocking For Railyards: Part I. Homogeneous Traffic," *Transportation Research B*. Volume 21B. No. 1, 1987.

[30] Carlos F. Daganzo, "Dynamic Blocking For Railyards: Part II. Heterogeneous Traffic," *Transportation Research B*. Volume 21B. No. 1, 1987.

[31] Carlos F. Daganzo, Richard G. Dowling, Randolph W. Hall, "Railroad Classification Yard Throughput: The Case of Multistage Triangular Sorting," *Transportation Research A*. Volume 17A, No. 2, 1983.

[32] C.S. Draper Laboratory, "PLATO (Planning Automatic Traffic Optimizer): Interface Document." February 17, 1994.

[33] M.A. Duffy, "Statistical Process Control Applied to Rail Freight Terminal Performances," S.M. Thesis, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, 1994.

[34] Robert T. Ferguson, "Mathematical Modeling of the Railroad Switching Process." Ph.D. dissertation, School of Systems Engineering, University of Virginia, 1993.

[35] Gene Golub, Charles Van Loan, *Matrix Computations*. Third Edition, The Johns Hopkins University Press, 1996.

[36] Betsy S. Greenberg, Robert C. Leachman, and Ronald W. Wolff, "Predicting Dispatching Delays on a Low Speed, Single Track Railroad," *Transportation Science*, Vol. 22, No. 1, 1988.

[37] Ali E. Haghani, "Formulation and Solution of a Combined Train Routing and Makeup, and Empty Car Distribution Model," *Transportation Research B*. Volume 23B, No. 6, 1989.

[38] J. M. Harrison, L. M. Wein, "Scheduling Networks of Queues: Heavy Traffic Analysis of a Two-Station Closed Network," *Operations Research*. Number 38, pp. 1052-1064.

[39] Patrick Harker, Sungwook Hong, "Two Moments Estimation of the Delay on a Partially Double Track Rail Line with Scheduled Traffic," *Transportation Research Forum*, 1992.

[40] Randolph W. Hall, "Vehicle Scheduling at a Transportation Terminal with Random Delay en Route," *Transportation Science*, Volume 19, No. 3, 1985.

[41] Susan Fraley Hallowell, "Optimal Dispatching Under Uncertainty: with application to railroad scheduling." Ph.D. dissertation, Department of Decision Sciences, University of Pennsylvania, 1993.

[42] Simon Haykin, *Neural Networks: A Comprehensive Foundation*. Macmillan, New York, NY, 1994.

[43] Dejan Jovanovich, Patrick T. Harker, "Tactical Scheduling of Rail Operations: The SCAN I System," *Transportation Science*, Vol 25, No. 1, February 1991.

[44] Dejan Jovanovich, Operations Research Group, Burlington Northern Railroad, *Personal Communication*, January, 1995.

[45] Mark H. Keaton, "Designing Optimal Railroad Operating Plans: Lagrangian Relaxation and Heuristic Approaches," *Transportation Research B*. Volume 23B, No. 6, 1989.

[46] Mark H. Keaton, "Designing Railroad Operating Plans: A Dual Adjustment Method for Implementing Lagrangian Relaxation," *Transportation Science*. Volume 26, No. 4, 1992.

[47] David R. Kraay, "Learning Heuristics for Repetitive Combinatorial Optimization Problems: with an application in train scheduling." Ph.D. dissertation, Department of Decision Sciences, University of Pennsylvania, 1993.

[48] David Kraay, Patrick T. Harker, Bintong Chen, "Optimal Pacing of Trains in Freight Railroads: Model Formulation and Solution," *Operations Research.* Volume 39, No. 1, 1991.

[49] P.R. Kumar, "Scheduling Queueing Networks: Stability, Performance Analysis and Design," in Frank Kelly, Ruth Williams, (eds.), *Stochastic Networks.* Springer-Verlag, 1995.

[50] Oh Kyoung Kwon, Carl D. Martland, Joseph M. Sussman, Patrick Little, "Origin-to-Destination Trip Times and Reliability of Rail Freight Services in North American Railroads," *Transportation Research Record 1489*, 1995.

[51] E. L. Lawler, J. K. Lenstra, A. G. H. Rinnooy Kan, D. B. Shmoys, "Sequencing and Scheduling: Algorithms and Complexity," in S. C. Graves, A. H. G. Rinnooy Kan, P. H. Zipkin, (eds.), *Handbooks in Operations Research and Management Science.* Volume 4, North-Holland, 1993.

[52] Carl Martland, "Railroad Reliability: Overview of Performance in North America," Presentation to AAR, September 1994.

[53] Carl D. Martland, Patrick Little, Michael Duffy, Yang Dong, "Benchmarks for Rail Hump Yard Performance," Affiliated Lab Working Paper 94-3, AAR Affiliated Laboratory, Center for Transportation Studies, Massachusetts Institute of Technology, 1994.

[54] Carl Martland, Patrick Little, Oh Kyoung Kwon, Rajesh Dontula, "Background on Railroad Reliability," Center for Transportation Studies, Massachusetts Institute of Technology, 1991.

[55] J. F. Muth, G. L. Thomson, *Industrial Scheduling.* Prentice Hall, 1963.

[56] K.V. Mardia, J.T. Kent, J.M. Bibby, *Multivariate Analysis.* Academic Press, 1979.

[57] A. I. Mees, "Railway Scheduling by Network Optimization," *Mathematical Computer Modeling*, Vol. 15, No. 1, 1991.

[58] José Niño Mora, "Optimal Resource Allocation in a Dynamic and Stochastic Environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[59] Harry N. Newton, "Network Design Under Budget Constraints with Application to the Railroad Blocking Problem," Ph.D. dissertation, Industrial and Systems Engineering, Auburn University, 1996.

[60] J.M. Ortega and W.C. Rheinbolt, *Iterative Solution of Nonlinear Equations in Several Variables.* Academic Press, 1970

[61] E.R. Peterson, "Over-the-Road Transit Time for a Single Track Railway," *Transportation Science* Vol. 8, No. 7, 65-74, 1974.

[62] E.R. Peterson, "Interference Delays on a Partially Double-Tracked Railway with Intermediate Signals," *Journal of the Transportation Research Forum* Vol. 16, No. 1, 54-62, 1974.

[63] Christos H. Papadimitriou, John N. Tsitsiklis, "The Complexity of Optimal Queueing Network Control," MIT Working Paper, undated.

[64] Michael Pinedo, *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[65] George Pugh, *Value Driven Methods of Decision and Control*, Draft Manuscript, M.I.T., February 1995.

[66] Maurice Queyranne, "Structure of a Simple Scheduling Polyhedron," *Mathematical Programming.* Volume 58, pp. 263-285, 1993.

[67] Michael J. Ricard, "Optimization of Queuing Networks; an Optimal Control Approach," Ph.D. dissertation, Massachusetts Institute of Technology, 1995.

[68] D.R. Smart, *Fixed Point Theorems*, Cambridge University Press, 1974.

[69] Gilbert Strang, *Linear Algebra and its Applications.* Third Edition, Harcourt Brace Jovanovich, 1986.

[70] Richard L. Sauder, William M. Westerman, "Computer Aided Train Dispatching: Decision Support Through Optimization," *Interfaces* 13: December, 1983.

[71] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning.* Volume 3, pp. 9 - 14, 1988.

[72] G. J. Tesauro, "Neurogammon: a neural network backgammon program," *International Joint Conference on Neural Networks Proceedings III.* pp. 33-39, 1990.

[73] G. J. Tesauro, "Practical Issues in Temporal Difference Learning," *Machine Learning.* Volume 8, pp. 257-277, 1992.

[74] G. J. Tesauro, "Temporal Difference Learning of Backgammon Strategy," *Machine Learning: Proceedings of the Ninth International Workshop.* 1992.

[75] G. Tesauro, G. R. Galperin, "On-Line Policy Improvement Using Monte Carlo Search," unpublished report, 1996.

[76] John N. Tsitsiklis, Benjamin Van Roy, "Feature-Based Methods for Large Scale Dynamic Programming," *Machine Learning.* Volume 22, pp. 59-94, 1996.

[77] M. A. Turnquist, M. S. Daskin, "Queuing Models of Classification and Connection Delay in Railyards," *Transportation Science.* Volume 16, pp. 207-230, 1982.

[78] Benjamin Van Roy, Dimitri P. Bertsekas, Yuchun Lee, John N. Tsitsiklis, "A Neuro-Dynamic Programming Approach to Retailer Inventory Management," Laboratory for Information and Decision Systems, Massachusetts of Technology, 1996.

[79] C.J.C.H. Watkins, P. Dayan, "Q-Learning," *Machine Learning.* Volume 8, pp. 279-292, 1992.

[80] S. Yagar and F. F. Saccomanddo, "An Efficient Sequencing Model for Humping in a Rail Yard," *Transportation Research A.* Volume 17A, No. 4, 1983.

[81] W. Zhang, T. G. Dietterich, "A Reinforcement Learning Approach to Job-Shop Scheduling," *Proceedings of the Fourteenth International Joint Confrerence on Artificial Intelligence*, pp. 1114-1120, 1995.

[82] W. Zhang, T. G. Dietterich, "High-Performance Job-Shop Scheduling with a Time-Delay TD(λ) Network," in *Advances in Neural Information Processing Systems 8*, MIT Press, 1996.

[83] Monte Zweben, Brian Daun, Michael Deale, "Scheduling and Rescheduling with Iterative Repair," in Monte Zweben, Mark S. Fox, (eds.), *Intelligent Scheduling.* Morgan Kaufmann Publishers, 1994.