VERY LARGE-SCALE NEIGHBORHOOD SEARCH HEURISTICS FOR COMBINATORIAL OPTIMIZATION PROBLEMS

By

KRISHNA CHANDRA JHA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2004

This work is dedicated to my father

ACKNOWLEDGMENTS

First of all, I would like to thank Dr. Ravindra K. Ahuja, who has been a great supervisor and mentor throughout my four years at the University of Florida. His inspiring enthusiasm and energy have been contagious, and his advice and constant support have been extremely helpful.

I would also like to express my gratitude to Dr. Joseph P. Geunes, Dr. Elif Akcali, and Dr. Anand Rangrajan for serving as my supervisory committee members and giving me thoughtful advice.

My special acknowledgement goes to my collaborators and colleagues at University of Florida and at other places, especially, Dr. Jian Liu, Dr. Claudio Cunha, Arvind Kumar, Guvenc Sahin, Michelle Hanna, Onur Seref and Kamalesh Somani. My special thanks go to Ameetkumar Motwani for the help in editing and proofreading of this dissertation.

My utmost appreciation goes to my father, the source of all my strength, and to my family for their encouragement and eternal support. I would like to thank my wife Nili for her patience, kindness and continuous support during my four years of study at University of Florida. Finally, my thanks go to my daughter Archita for the hardships faced by her during my study and to my son Anish, whose arrival pushed me to complete this dissertation at the earliest.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

VERY LARGE-SCALE NEIGHBORHOOD SEARCH HEURISTICS FOR COMBINATORIAL OPTIMIZATION PROBLEMS

By

Krishna Chandra Jha

May 2004

Chair: Ravindra K. Ahuja
Major Department: Industrial and Systems Engineering

Combinatorial optimization plays an important role in decision-making since optimal decisions often depend on a nontrivial combination of various factors. Most combinatorial optimization problems are NP-hard and sharp bounds on the optimal value are typically hard to derive. This means that partial enumeration based exact algorithms have a slow convergence rate, and they can solve only small instances to optimality. But real-life combinatorial optimization problems are typically of big size and since exact approaches are inadequate, heuristics are commonly used in practice. Over the last 15 years much of the research effort has been concentrated on the development of local search heuristics. In local search methods, an intensive exploration of the solution space is performed by moving at each step from the current solution to another promising solution in its neighborhood. As a rule of thumb, there is a better probability of finding a promising solution, if the neighborhood size is larger. In this dissertation, we have developed algorithms, which perform intensive searches in a very large size

neighborhood. We have used the concepts of network flow algorithms to make our algorithms computationally efficient.

In Chapter 2, we propose very large-scale neighborhood (VLSN) search algorithms for the quadratic assignment problems (QAP), and we propose a novel search procedure to enumerate good neighbors heuristically. Our search procedure relies on the concept of an improvement graph, which allows us to evaluate neighbors much faster than the existing methods. We present extensive computational results of our algorithms on a standard benchmark QAP instances. In Chapter 3, we suggest linear programming, integer programming, and network flow based lower bounding; using these methods, we obtain several branch and bound algorithms for the weapon-target assignment problem. We also propose a network flow based construction heuristic and a very large-scale neighborhood (VLSN) search algorithm. In Chapter 4, we have developed VLSN search heuristics in conjunction with tabu search to solve a university course-timetabling problem. We have developed a generic model for the problem and have performed computational experiments on randomly generated test instances.

In Chapter 5, we have given integer programming formulations of the block-to-train assignment problem; propose several exact and heuristic algorithms to solve this problem, and present computational results on the data provided by a major US railroad. In chapter 6, we have proposed a decomposition based approach to solve another railroad scheduling problem, called the train schedule design problem. We have developed VLSN search heuristics to solve the sub-problem in first phase and have formulated the problem in second phase as a minimum cost flow problem.

CHAPTER 1
INTRODUCTION

Optimization has been expanding in all directions at an astonishing rate during the last few decades. New algorithmic and theoretical techniques have been developed, the diffusion into other disciplines has proceeded at a rapid pace, and our knowledge of all aspects of the field has grown even more profound. At the same time, one of the most striking trends in optimization is the constantly increasing emphasis on the interdisciplinary nature of the field. Optimization today is a basic research tool in all areas of engineering, medicine and the sciences. The decision making tools based on optimization procedures are successfully applied in a wide range of practical problems arising in virtually any sphere of human activities, including biomedicine, energy management, aerospace research, telecommunications and finance.

The problems of finding the "best" and the "worst" have always been of great interest. For example, given n sites, what is the fastest way to visit all of them consecutively? In manufacturing, how should one cut plates of a material so that the waste is minimized? Some of the first optimization problems were solved in ancient Greece and are regarded among the most significant discoveries of that time. In the first century A.D., Alexandrian mathematician Heron solved the problem of finding the shortest path between two points by way of the mirror. This result, also known as Heron's theorem of the light ray, can be viewed as the origin of the theory of geometrical optics. The problem of finding extreme values gained a special importance in the

seventeenth century when it served as one of motivations in the invention of differential calculus. The soon-after-developed calculus of variations and the theory of stationary values lie in the foundation of the modern mathematical theory of optimization.

Many well-structured problems encountered in business (and other domains) can be defined as optimization problems

P: maximize (minimize) $f(x)$

s.t. $x \in S$

where *S* represents the set of admissible solutions (satisfying all hard constraints) of an underlying search space *X*. The function $f: S \rightarrow \mathbb{R}$, where *R* represents the set of real numbers, is often called goal function or objective function.

A large subclass of these problems is the class of combinatorial optimization problems. An optimization problem is called combinatorial, if *X* and *S* are combinatorial or discrete in some sense. Although there does not seem to be the formal common understanding of what is 'combinatorial', we consider here that *X* and/or *S* are 'combinatorial' if they are discrete sets of finite elements or countably infinite elements (Ibaraki [1987]).

Combinatorial optimization plays an important role in decision-making since optimal decisions often depend on a nontrivial combination of various factors. Most combinatorial optimization problems are NP-hard, and sharp bounds on the optimal value are typically hard to derive. This means that partial enumeration based exact algorithms have a slow convergence rate, and they can solve only small instances to optimality. But real-life combinatorial optimization problems are typically of big size, and since exact approaches are inadequate, heuristics are commonly used in practice. Etymologically, the

word “heuristic” comes from the Greek *heuriskein* (to find). There has been a steady evolution over the past forty years in the development of heuristics, which produce solutions of reasonably good quality in a reasonable amount of time. The first proposed heuristics tried to systemize decision-making processes done by hand. With the help of computers, which can test a huge amount of combinations in a short amount of time, solutions could easily be generated which have turned out to be of much better quality than what an expert in the field could produce by hand. In these early heuristics, much of the emphasis was put on quickly obtaining a feasible solution and possibly applying to it a post-optimization procedure.

The literature devoted to heuristic algorithms often distinguishes between two broad classes: constructive algorithms and improvement algorithms. A constructive algorithm builds a solution from scratch by assigning values to one or more decision variables at a time. An improvement algorithm generally starts with a feasible solution and iteratively tries to obtain a better solution. Neighborhood search algorithms (alternatively called local search algorithms) are a wide class of improvement algorithms where at each iteration an improving solution is found by searching the “neighborhood” of the current solution. A procedure of this type usually terminates when the first local optimum is obtained. Randomization and restarting approaches used to overcome poor quality local solutions are often ineffective.

More general strategies known as meta-heuristics usually combine some heuristic approaches and direct them towards solutions of better quality than those found by local search heuristics. Over the last 15 years much of the research effort has concentrated on the development of meta-heuristics, using mainly two principles: local search and

population search. The term meta-heuristic was coined by Glover[1986] and has come to be widely applied in the literature, both in the titles of comparative studies and in the titles of collected research papers. A meta-heuristic refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality. The heuristics guided by such a strategy may be a high level procedure or may embody nothing more than a description of available moves for transforming one solution to another, together with an associated evaluation rule.

The evolution of meta-heuristics during the past decade has taken an explosive upturn. Meta-heuristics in their modern forms are based on a variety of interpretations of what constitutes "intelligent" search. These interpretations lead to design choices that in turn can be used for classification purposes. However, a rigorous classification of different meta-heuristics is a difficult and risky enterprise, because the leading advocates of alternative methods often differ among themselves about the essential nature of the methods they espouse. Meta-heuristics are often viewed as composed of processes that are intelligent, but in some instances the intelligence belongs more to the underlying design than to the particular character (or behavior) of the method itself.

In local search methods, an intensive exploration of the solution space is performed by moving at each step from the current solution to another promising solution in its neighborhood. Simulated annealing (Kirkpatrick et al. [1983]), Tabu search (Glover [1986]) and variable neighborhood search (Miladenovic and Hanson [1997]) are the most famous local search methods. Population search consists of maintaining a pool of good solutions and combining them in order to produce hopefully better solutions. Classical

examples are genetic algorithms (Holland [1975]) and adaptive memory procedures (Rochat and Taillard [1995]). Meta-heuristics are general combinatorial optimization techniques, which are not dedicated to the solution of a particular problem, but are rather designed with the aim of being flexible enough to handle as many different combinatorial problems as possible. These general techniques have rapidly demonstrated their usefulness and efficiency in solving hard problems. Success stories are reported in many papers. While meta-heuristics can handle in theory any combinatorial optimization problem, it is often the case that an important effort must be put on finding the right way to adapt the general ingredients of these methods to the particular considered problem.

We think that in order to be successful in the adaptation of a meta-heuristic to a combinatorial optimization problem, it is necessary to follow some basic principles. We give in the next sections some guidelines, which may help in producing such successful adaptations of local search and population search methods for the solution of difficult combinatorial optimization problems.

This dissertation concentrates on neighborhood search algorithms where the size of the neighborhood is “very large” with respect to the size of the input data. For large problem instances, it is impractical to search these neighborhoods explicitly, and one must either search a small portion of the neighborhood or else develop efficient algorithms for searching the neighborhood implicitly.

A critical issue in the design of a neighborhood search approach is the choice of the neighborhood structure, that is, the manner in which the neighborhood is defined. This choice largely determines whether the neighborhood search will develop solutions that are highly accurate or whether they will develop solutions with very poor local optima.

As a rule of thumb, the larger the neighborhood, the better is the quality of the locally optimal solutions, and the greater is the accuracy of the final solution that is obtained. At the same time, the larger the neighborhood, the longer it takes to search the neighborhood at each iteration. Since one generally performs many runs of a neighborhood search algorithm with different starting points, longer execution times per iteration lead to fewer runs per unit time. For this reason a larger neighborhood does not necessarily produce a more effective heuristic unless one can search the larger neighborhood in an efficient manner.

Some very successful and widely used methods in operations research can be viewed as very large-scale neighborhood search techniques. For example, if the simplex algorithm for solving linear programs is viewed as a neighborhood search algorithm, then column generation is a very large-scale neighborhood search method. Also, the augmentation techniques used for solving many network Bows problems can be categorized as very large-scale neighborhood search methods. The negative cost cycle canceling algorithm for solving the min cost bow problem and the augmenting path algorithm for solving matching problems are two such examples.

The combinatorial optimization problems, we studied in this dissertation, are very large-scale optimization problems and are impossible to solve to optimality using the state-of-the-art algorithmic approaches. In this dissertation, we focus on developing *heuristic* algorithms for this problem that do not guarantee optimality of the solution but find a nearly optimal solution within a reasonable computational time. The literature devoted to heuristic algorithms often distinguishes between two broad classes: *constructive algorithms* and *neighborhood search algorithms* (alternatively called *local*

search algorithms). A constructive algorithm builds a solution of an optimization problem from scratch by assigning values to one or more decision variables at a time. In contrast, a neighborhood search algorithm starts with a feasible solution x of the problem, defines a set of neighboring solutions $\mathbf{N}(x)$ of the solution x, called the *neighborhood* of x, evaluates neighboring solutions and if it finds a solution $y \in \mathbf{N}(x)$ that is better than x, then it replaces x with y. It next defines the neighborhood with respect to the new solution x and repeats this procedure until the solution x is at least as good as any other solution in $\mathbf{N}(x)$. The solution x is then declared as a *local optimal solution* and the algorithm terminates.

The use of neighborhood search algorithms in solving optimization problems has a long history (Croes [1958], Lin [1965], Lin and Kernighan [1973], Kirkpatrick et al. [1983], Reeves [1993], Osman and Laporte [1996], and Aarts and Lenstra [1997]). However, in the past fifteen years, the fields of operations research, mathematical programming and computer science have all witnessed a strong renewed interest in the development and analysis of neighborhood search based approaches. This interest can be attributed in large part to the intuitive appeal of these algorithms, flexibility, and the ease of implementations of these algorithms to solve many complex real-world problems. Neighborhood search algorithms are now widely regarded as an important tool to solve difficult optimization problems effectively, and in particular, combinatorial optimization problems.

A critical issue in the design of a neighborhood search algorithm is the choice of the *neighborhood structure*; that is, the manner in which the neighborhood is defined. In general, the larger the neighborhood, the better is the quality of locally optimal solutions

and greater is the accuracy of the final solution. At the same time, the larger the neighborhood, the longer it takes to enumerate the neighborhood and evaluate solutions in it. Hence, a larger neighborhood does not necessarily produce a more effective heuristic until one can efficiently search the larger neighborhood.

Recently, numerous efforts are made in developing very large-scale neighborhood (VLSN) search algorithms where the size of the neighborhood is very large. In fact, it is so large that enumerating all neighbors and evaluating them is prohibitively expensive. Using concepts from network flow theory (Ahuja et al. [1993]), a VLSN search algorithm implicitly enumerates the neighborhood to identify an improved neighbor. A VLSN search algorithm can consider neighborhoods with trillions of neighbors while the time to identify improved neighbor is fairly small, often less than a fraction of a second (Ahuja et al. [2002a). Some of the interesting applications of VLSN algorithms are for airline fleet scheduling problems (Ahuja et al. [2001b, 2003b, 2003c]), the locomotive scheduling problem (Ahuja et al. [2002b]), the capacitated minimum spanning tree problem (Ahuja et al. [2001a, 2003a]), the capacitated facility location problem (Ahuja et al. [2002c]).

1.1 Dissertation Summary

As the title suggests, in this dissertation efforts have been made to solve a few hard combinatorial optimization problems by developing VLSN search heuristics. The problems studied are the quadratic assignment problem, the weapon-target assignment problem, the university course-timetabling problem, the block-to-train assignment problem, and the train design problem. We give below a brief summary of our efforts to solve these problems.

1.1.1 Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) consists of assigning n facilities to n locations so as to minimize the total weighted cost of interactions between facilities. In this dissertation, the QAP arises in many diverse settings, is known to be NP-hard, and can be solved to optimality only for fairly small size instances (typically, $n \leq 30$). Neighborhood search algorithms are the most popular heuristic algorithms to solve larger size instances of the QAP. The most extensively used neighborhood structure for the QAP is the 2-exchange neighborhood. This neighborhood is obtained by swapping the locations of two facilities and thus has size $O(n^2)$. Previous efforts to explore larger size neighborhoods (such as 3-exchange or 4-exchange neighborhoods) were not very successful, as it took too long to evaluate the larger set of neighbors. In this dissertation, we propose very large-scale neighborhood (VLSN) search algorithms where the size of the neighborhood is very large and we propose a novel search procedure to enumerate good neighbors heuristically. Our search procedure relies on the concept of an improvement graph, which allows us to evaluate neighbors much faster than the existing methods.

We present extensive computational results of our algorithms on standard benchmark instances available in QAPLIB (http://www.opt.math.tu-graz.ac.at/qaplib). There are two types of quadratic assignment problems: (i) flow and distance matrices are symmetric (for example, distance from location A to B is same as that from B to A), and (ii) flow and distance matrices are asymmetric (for example, distance from location A to location B is different than that from B to A). In this dissertation, we have tested our algorithms on both types of problems. The computational results show that our

algorithms are very robust and produce very good quality solutions for all kinds of problem instances.

1.1.2 Weapon-Target Assignment Problem

The Weapon-Target Assignment (WTA) problem is a fundamental problem arising in defense-related applications of operations research. This problem consists of optimally assigning *n* weapons to *m* targets so that the total expected survival value of the targets after all the engagements is minimum. The WTA problem can be formulated as a nonlinear integer programming problem and is known to be NP-complete. The researchers have studied this problem for more than three decades; however, there do not exist any exact methods for the WTA problem, which can solve even small size problems (for example, with 20 weapons and 20 targets). Though several heuristic methods have been proposed to solve the WTA problem, due to the absence of exact methods, no estimates are available on the quality of solutions produced by such heuristics. In this dissertation, we suggest linear programming, integer programming, and network flow based lower bounding methods using which we obtain several branch and bound algorithms for the WTA problem. We have used piecewise approximation to obtain linear objective function for the problem. We have developed algorithms to determine the bounds on number of weapons assigned to a target. These bounds play a critical role in formulating the problem as a minimum cost flow problem. We also propose a network flow based construction heuristic and a VLSN search algorithm.

The combination of network flow based construction heuristic and VLSN search algorithms is very efficient and produces exceptionally good quality solutions for the problem instances even of size 200 weapons and 200 targets. We present computational results of our algorithms, which indicate that we can solve moderately large size

instances (up to 80 weapons and 80 targets) of the WTA problem optimally and obtain almost optimal solutions of fairly large instances (up to 200 weapons and 200 targets) within a few seconds.

1.1.3 University Course Timetabling Problem

The university C*ourse Timetabling Problem* (CTP) is to develop a weekly course timetable containing the time and the meeting place for each course over a given time horizon (generally, a semester or trimester). The university course-timetabling problem is much different than the school-timetabling problem. In the school timetabling problem, students are generally required to take a fixed set of courses, whereas students have lot of freedom in selecting courses in a university. Also, teachers are mainly engaged in teaching courses at a school, whereas they have dual responsibilities of teaching and research in a university. The objectives of the university course-timetabling problem are to maximize the preferences of teachers for the periods at which they want to teach, and to minimize the time clash among the courses a student wants to take. Generally, teachers give their preferences at the beginning of the planning period, whereas the students' interest data are collected from the past enrollment. The course timetable problem is a highly constrained optimization problem, in which a constraints set widely varies from one case to another. Previous efforts to solve the problem are mainly case specific and are limited to small instances. In this dissertation, we propose a generic model of the problem, which can be applied to a wide variety of real-life instances. In our model, the periods are grouped into slots and we assign the courses to the slots to optimize the objective function. We have formulated the course-timetabling problem as an integer-programming problem. We have developed a construction heuristic and have developed a VLSN search algorithm to solve the problem. We have used a popular meta-heuristic,

tabu search, in conjunction with VLSN search algorithms. This approach permits us to search for the better solution in wide solution space. We report the performance of our algorithm when compared to an optimal solution on randomly generated test instances.

1.1.4 Block-to-Train Assignment Problem

Railroads classify shipments into *blocks* to reduce intermediate handlings of cars. Once a shipment is assigned to a block, it is not reclassified at the intermediate yards on the route from the origin of the block to the destination of the block. The *block-to-train assignment* (BTA) problem consists of assigning these blocks to trains so that the shipments reach their destinations such that the transportation cost and transit time are minimum. The transportation cost of a block consists of distance traveled by the block and cost of swapping whenever it changes the train on the route. The transit time for a block is the time taken in traveling from its origin to its destination. The solution of the BTA problem must honor the capacity constraints on the trains. This is a large size hard combinatorial optimization problem. In the past very few efforts have been made to solve the BTA problem. In this dissertation, we formulate this problem as an integer programming problem on a space-time network. We have developed label enumerating algorithms to enumerate possible routes for a block in the space-time network. Our label correcting algorithm is a VLSN search technique, which very efficiently produces a set of best routes out of enormous possible routes for a block. We also propose several exact and heuristic algorithms to solve this problem. We have tested our algorithms on a real life BTA problem of a major US railroad company.

1.1.5 Train Design Scheduling Problem

The train design scheduling (TDS) problem is one of the most difficult and complex scheduling problems in the railroad. Due to the ever changing nature of the transportation demand, railroads are required to come up with a completely new train schedule or will have to change the existing train schedule. The train schedule design problem is to design a zero-based (do not consider existing solution) train schedule, in which decisions are made with respect to trains to be made, their origins and destinations, their routes and frequency, the time schedule, and the block-to-train assignment. The objectives of the TDS problem are to minimize the overall transportation cost and transit time of shipments. There are many logical and practical constraints, which a solution of the TDS problem is required to honor. For a real-life instance of any major US railroad, the TDS problem consists of millions of integral decision variables and constraints. With the current age computational capability, it is not possible to solve a problem of this magnitude using any of the state-of-art optimization techniques. In the past, few efforts were made to solve this problem heuristically, however, they were implemented on a much simplified version of the problem or to small instances. In this dissertation, we have developed a decomposition based approach to solve the problem. Our approach consists of solving the problem sequentially in three phases. We have developed algorithms to solve the sub-problems in Phase I and Phase II. The problem in Phase I is a network design problem and we have developed a VLSN search algorithm to solve this problem, which sequentially solve the problem in this phase. We have formulated the problem in Phase II as a minimum cost flow problem on a suitably constructed graph, in which we define the cost of arcs in minimum cost flow formulation in accordance with the objectives of the problem in this phase. The computational experiment on a real life

instance indicates that the problems in Phase I and II can be solved very efficiently using a normal computational facility.

1.2 Contribution Summary

We summarize below the contributions made in this dissertation.

1.2.1 Quadratic Assignment Problem

- We have developed a VLSN search algorithm for the quadratic assignment problem, a non-partitioning optimization problem. The application of VLSN search algorithm is different than that of partitioning problems, to which this algorithm was applied earlier.

- We have an approximated an improvement graph, in which the cost of the cycle is a very good approximation of the cost of the multi-exchange, and allows us to enumerate good neighbors quickly. Typically, evaluating a k-exchange neighbor for the QAP takes $O(nk)$ time; but using the improvement graph we can do it in $O(k)$ average time per neighbor.

- We have developed a generic search procedure to enumerate neighbors using improvement graphs. We have also developed several implementations of the generic search procedure, which enumerate the neighborhoods exactly as well as heuristically.

- We present a detailed computational investigation of local improvement algorithms based on our neighborhood search structures. In our experimentation, we have considered 98 symmetric instances and 34 asymmetric test instances available in the QAPLIB (QAP Library).

1.2.2 Weapon-Target Assignment Problem

- We formulate the WTA problem as an integer linear programming problem, that is, as a generalized integer network flow problem on an appropriately defined network. The linear programming relaxation of this formulation gives a lower bound on the optimal solution of the WTA problem.

- We propose a minimum cost flow formulation that yields a different lower bound on the optimal solution of the WTA problem. This lower bound is, in general, not as tight as the bound obtained by the linear programming formulation described above but it can be obtained in much less computational time.

- We propose a third lower bounding scheme, which is based on simple combinatorial arguments and uses a greedy approach to obtain a lower bound.

- We develop branch and bound algorithms to solve the WTA problem employing each of the three bounds described above.

- We propose a very large-scale neighborhood (VLSN) search algorithm to solve the WTA problem. The VLSN search algorithm is based on formulating the WTA problem as a partition problem. The VLSN search starts with a feasible solution of the WTA problem and performs a sequence of *"cyclic and path exchanges"* to improve the solution. We describe a heuristic method that obtains an excellent feasible solution of the WTA problem by solving a sequence of minimum cost flow problems, and then uses a VLSN search algorithm to iteratively improve this solution.

- We perform extensive computational investigations of our algorithms and report these results. Our algorithms solve moderately large size instances (up to 80 weapons and 80 targets) of the WTA problem optimally and obtain almost optimal solutions of fairly large instances (up to 200 weapons and 200 targets) within a few seconds.

1.2.3 University Course Timetabling Problem

- We have developed a generic model of the university course timetabling problem. We expect that this model can be applied to many real-life course timetabling problems.

- We formulate the course timetabling problem as an integer programming problem. We have been able to solve moderate size problems to optimality using CPLEX optimizer.

- The course timetabling problem is a partitioning problem with side constraints. We develop an improvement graph based VLSN search algorithm to solve this problem.

- Due to the constraints in course timetabling problem, solution space is very sparse. We developed an algorithm in conjunction with tabu search, which permits the algorithm to search for the good quality solution in wide solution space.

- We generated the test instances randomly and performed computational experiment of the algorithms. Our heuristic algorithms perform extremely well on the test instances and ware able to produce optimal solution in almost all the cases.

1.2.4 Block-to-Train Assignment Problem

- We develop a space-time network, which allows us to formulate the BTA problem as a multi-commodity flow problem with additional side constraints.

- We develop two integer programming formulations of the BTA problem. The first formulation formulates the BTA problem as a node-arc version of multi-commodity flow problem in the space-time network. The second formulation conceives the BTA problem as a path-flow version of the multi-commodity flow problem in the space-time network.

- The integer programming formulations can also be used to solve the BTA problem approximately. Using these formulations, we suggest a Lagrangian relaxation algorithm that can solve the BTA problem to near-optimality within a few seconds.

- We also propose a greedy construction algorithm that proceeds by assigning blocks to trains one by one until all blocks are assigning. By choosing different rules for selecting blocks and assigning them to trains, we can obtain different implementations of this generic approach.

- We perform extensive computational investigations of these algorithms. Our integer programming formulations can solve the BTA problem to optimality within a few minutes of computer time using CPLEX 8.1. The Lagrangian relaxation algorithm can solve the BTA problem to near-optimality within a few seconds. The computational performance of greedy construction algorithms is also very attractive.

1.2.5 Train Schedule Design problem

- We have developed a decomposition based approach to solve real-life train design problem.

- The sub problem in phase I is a network design problem and we have developed a VLSN search heuristic to solve the problem in this phase.

- We have formulated the problem in Phase II as a minimum cost flow problem on a suitably constructed graph, which can be solved very efficiently using network flow optimization technique.

- We perform the computational experiment on the algorithms using real-life problem instance. Our results show that our approach can produce substantial saving in terms of number of trains required.

To summarize, our contributions in this dissertation are in developing very large scale neighborhood search heuristics, which use efficient network flow algorithms to obtain good quality solutions of different combinatorial optimization problems. Our strength lies in proper formulations of the problems and state-of-art implementation of

algorithms. We have focused on developing algorithms such that near optimal solution can be obtained very quickly using a normal computational facility. We have also tried to develop robust algorithms, which can be applied across different real-life scenarios.

CHAPTER 2
QUADRATIC ASSIGNMENT PROBLEM

2.1 Introduction

The Quadratic Assignment Problem (QAP) is a classical combinatorial optimization problem and is widely regarded as one of the most difficult problems in this class. Given a set $N = \{1, 2, ..., n\}$, and nxn matrices $F = \{f_{ij}\}$, $D = \{d_{ij}\}$, and $B = \{b_{ij}\}$, the QAP is to find a permutation ϕ of the set N which minimizes:

$$z(\phi) = \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{\phi(i)\phi(j)} + \sum_{i=1}^{n} b_{i\phi(i)} \qquad 2.1$$

The QAP arises as a natural problem in facility layout. In this context, the set N represents a set of n facilities (numbered 1 through n) that need to be assigned to locations (numbered 1 through n). The matrix $F = \{f_{ij}\}$ represents the flow between different facilities, and the matrix $D = \{d_{ij}\}$ represents the distance between locations. For example, if the facilities are departments in a campus, then the flow f_{ij} could be the average number of people walking daily from department i to department j. The decision variable $\phi(i)$, $1 \leq i \leq n$, represents the location assigned to facility i. Since there are n facilities and n locations and a facility can be assigned to exactly one location, there is a one-to-one correspondence between feasible solutions of QAP and permutations ϕ.

Observe that Equation 2.1 consists of two terms. The first term is the sum of n^2 flow costs between n facilities (the term $f_{ij}d_{\phi(i)\phi(j)}$ represents the cost of flow from facility i to facility j). The second term considers the cost of erecting facilities, which may be

location-dependent. The matrix $B = \{b_{ij}\}$ represents the cost of creating facility i at location j. Hence, the QAP is to find an assignment of facilities to locations so as to minimize the total cost of flow between the facilities and the cost of erecting the facilities. The matrices F and D are typically symmetric matrices but are not required to be so. In our algorithms, we allow asymmetric instances and thus do not assume that $f_{ij} = f_{ji}$ or $d_{ij} = d_{ji}$. However, for the sake of simplicity, we will assume in Sections 2.2 through 2.6 that we are working with symmetric instances. In Section 2.7, we will study asymmetric instances.

In addition to the facility layout, the QAP arises in many other applications, such as the allocation of plants to candidate locations, the backboard wiring problem, design of control panels and typewriter keyboards, turbine balancing, ordering of interrelated data on a magnetic tape, and others. The details and references for these and additional applications can be found in Malucelli [1993], Pardalos, Rendl and Wolkowicz [1994], Burkard et al. [1998], and Cela [1998]. Given the wide range of applications and the difficulty of solving the problem, the QAP has been investigated extensively by the research community. The QAP is known to be NP-hard, and a variety of exact and heuristic algorithms have been proposed. Exact algorithms for solving QAP include approaches based on (i) dynamic programming (Christofides and Benavent [1989]); (ii) cutting planes (Bazaraa and Sherali [1980]); and (iii) branch and bound (Lawler [1963], Pardalos and Crouse [1989]). Among these, the branch and bound algorithms are the most successful, but they are generally unable to solve problems of size larger than 25 facilites.

Since the applications of the QAP have often given rise to problems of size far greater than 25, there is a need for good heuristics for QAP that can solve larger size problems. A wide variety of heuristic approaches have been developed for the QAP. These can be classified into the following categories: (i) *construction methods* (Buffa, Armour and Vollmann [1964], Muller-Merbach [1970]); (ii) *limited enumeration methods* (West [1983], Burkard and Bonniger [1983]); (iii) *GRASP* (greedy randomized adaptive search procedure) (Li, Pardalos, and Resende [1994]); (iv) *simulated annealing methods* (Wilhelm and Ward [1987]); (v) *tabu search methods* (Skorin-Kapov [1990], Taillard [1991]); (vi) *genetic algorithms* (Fleurent and Ferland [1994], Tate and Smith [1985], Ahuja, Orlin, and Tewari [1998], Drezner [2003]); and (vii) *ant systems* (Maniezzo, Colorni, and Dorigo [1994]). The tabu search method of Taillard [1991], the GRASP method of Li, Pardalos, and Resende [1994], and the genetic algorithm by Drezner [2003] are the most accurate heuristics among these methods.

As observed in the survey paper of Burkard et al. [1998], the current neighborhood search meta-heuristic (tabu search and simulated annealing) algorithms for the QAP use the *two-exchange* neighborhood structure in the search. A permutation ϕ' is called a *2-exchange neighbor* of the permutation ϕ if it can be obtained from ϕ by switching the values of two entries in the permutation ϕ. It is easy to see that the number of two-exchange neighbors of a permutation is $O(n^2)$. There has been very limited effort in the past to explore larger neighborhood structures for the QAP as the time needed to identify an improved neighbor becomes too high. In this chapter, we investigate the neighborhood structure based on *multi-exchanges*, which is a natural generalization of the 2-exchanges. A multi-exchange is specified by a cyclic sequence $C = i_1 - i_2 - \ldots - i_k - i_1$ of facilities

such that $i_p \neq i_q$ for $p \neq q$. This multi-exchange implies that facility i_1 is assigned to the location $\phi(i_2)$, facility i_2 to $\phi(i_3)$, and so on, and finally facility i_k is assigned to $\phi(i_1)$. The location of all other facilities is not changed. We denote by ϕ^C, the permutation obtained by applying the multi-exchange C to the permutation ϕ. In other words,

$\phi^C(i) = \phi(i)$ for $i \in N \setminus \{i_1, \ldots, i_k\}$, 2.2

$\phi^C(i_p) = \phi(i_{p+1})$ for $p = 1,\ldots, k\text{-}1$, and

$\phi^C(i_k) = \phi(i_1)$.

We define the length of a multi-exchange as the number of facilities involved in the corresponding cyclic sequence. For example, the cyclic sequence $C = i_1 - i_2 - \ldots - i_k - i_1$ has length k. We also refer to a multi-exchange of length k as a *k-exchange*. Figure 2.1 illustrates an example of a *3-exchange*. We note that a k-exchange can be generated by k different cyclic sequences. For example, the 3-exchange shown in Figure 2.1 can be generated by any of the sequences *3-7-6-3, 7-6-3-7*, and *6-3-7-6*.

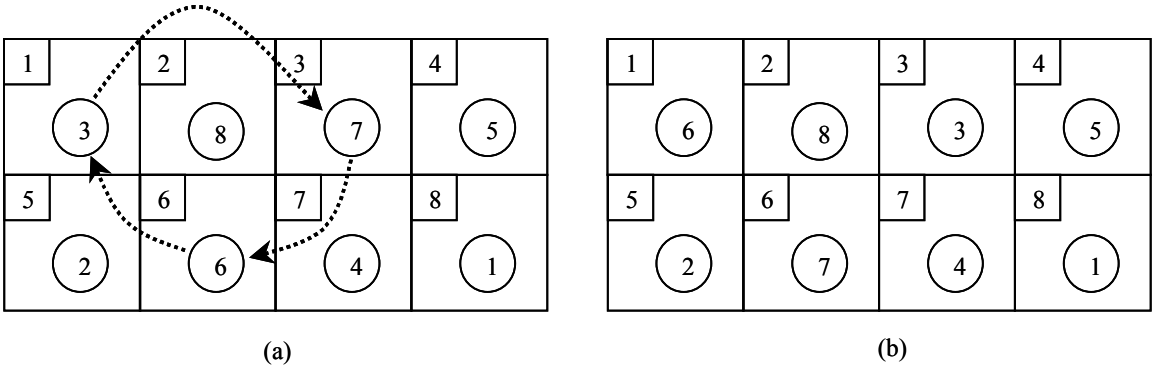


Figure 2.1 Facilities Locations: (a) Initial assignment and (b) Assignment after the cyclic exchange 3-7-6-3.

Given a positive integer $2 \leq K \leq n$, the *K-exchange neighborhood structure* consists of all the neighbors of a permutation obtained by using multi-exchanges of length at most

K. We note that the two-exchange neighborhood structure is contained in the *K*-exchange neighborhood structure. The number of neighbors in the *K*-exchange neighborhood structure is $\Omega(\binom{n}{K}(K-1)!)$. This number is very large even for moderate values of *K*. For example, if $n = 100$ and $K = 10$, then the K-exchange neighborhood may contain as many as $6x10^{18}$ neighbors. This neighborhood structure falls under the category of very large-scale neighborhood (VLSN) structures where the size of the neighborhood is too large to be searched explicitly and we use implicit enumeration methods to identify improved neighbors.

Algorithms based on very large-scale neighborhood structures have been successfully used in the context of several combinatorial optimization problems (see Ahuja et al. [2002a], and Deineko and Woeginger [1999] for surveys on this type). One of the tools used in performing search over very large-scale neighborhood structures is the concept of the *improvement graph*. In this technique, we associate a graph, called the improvement graph $G(\phi)$, with each feasible solution ϕ of the combinatorial optimization problem. The improvement graph $G(\phi)$ is constructed such that there is a one-to-one correspondence between every neighbor of ϕ to some directed cycle (possibly satisfying certain constraints) in the improvement graph $G(\phi)$. We also define arc costs in the improvement graph so that the difference in the objective function value of a neighboring solution and the solution ϕ is equal to the cost of the constrained cycle corresponding to the neighbor. This transforms the problem of finding an improved neighbor into the problem of finding a negative cost constrained cycle in the improvement graph (assuming that the combinatorial optimization problem is a minimization problem). The concept of

improvement graph was first proposed by Thompson and Orlin [1989] for a partitioning problem, where a set of elements is partitioned into several subsets of elements so as to minimize the sum of the objective functions of the subsets. This technique has been used to develop several VLSN search algorithms for specific partitioning problems such as the vehicle routing problem (Thompson and Psaraftis [1993]) and the capacitated minimum spanning tree problem (Ahuja, Orlin, Sharma [2003a, 2001a]). The concept of improvement graph was also used by Talluri [1996] and Ahuja et al. [2001b] to search very large-scale neighborhoods as in fleet assignment problems arising in airline scheduling. Ergun [2001] also proposed several improvement graphs for the vehicle routing problem and machine scheduling problems.

In this chapter, we study the use of the improvement graph for the multi-exchange neighborhood structure for the QAP. However, our current application of the improvement graph is different than previous applications. In previous applications, the improvement graph satisfied the property that the cost of the multi-exchange was equal to the cost of the corresponding (constrained) cycle in the improvement graph. This property is not ensured for the improvement graph for the QAP. Rather, the cost of the cycle is a very good approximation of the cost of the multi-exchange, and allows us to enumerate good neighbors quickly. The improvement graph also allows us to evaluate the cost of a neighbor faster than using a normal method. Typically, evaluating a k-exchange neighbor for the QAP takes $O(nk)$ time; but using the improvement graph we can do it in $O(k)$ average time per neighbor.

We developed a generic search procedure to enumerate neighbors using improvement graphs. We also developed several implementations of the generic search

procedure, which enumerate the neighborhoods exactly as well as heuristically. We present a detailed computational investigation of local improvement algorithms based on our neighborhood search structures. Our investigations yield the following conclusions: (i) locally optimal solutions obtained using multi-exchange neighborhood search algorithms are superior to those obtained using 2-exchange neighborhood search algorithms; (ii) generally, increasing the size of the neighborhood structure improves the quality of local optimal solutions but after a certain point there are diminishing returns; and (iii) enumerating a restricted subset of neighbors is much faster than enumerating entire neighborhood and can develop improvements that are almost as good.

This chapter is organized as follows. In Section 2.2, we describe the improvement graph data structure for the QAP. We present a generic heuristic search procedure for the *K*-exchange neighborhood structure for the QAP in Section 2.3. In Section 2.4, we describe several specific implementations of the generic search procedure. In Section 2.5, we describe the neighborhood search algorithm based on the generic search procedure. Section 2.6 describes an acceleration technique we use to speed up the performance of the algorithm. For clarity of concepts, in all theses sections we analyze and discuss the algorithms for symmetric cases only, which can be easily generalized for asymmetric cases. To keep the expressions simple, we assume that diagonal elements are zero either in flow matrix or cost matrix, and hence do not account completely the interaction among diagonal elements. In Section 2.7, we present the corresponding expressions for the general case (both symmetric and asymmetric instances). We provide and analyze the computational results from our implementations in Section 2.8. Section 2.9 summarizes our contributions.

2.2 Improvement Graph

One of the main contributions of this chapter is the development of the improvement graph to enumerate multi-exchanges for the QAP. In this section, we describe how to construct the improvement graph, and how it may help us in evaluating multi-exchanges quickly. This section as well as the following sections requires some network notations, such as cycles and paths. We will use the graph notation given in the book by Ahuja, Magnanti, and Orlin [1993] and refer the reader to this book for the same.

Given a permutation ϕ and a k-exchange C, we denote the *cost* of the cyclic exchange by $Cost(\phi, C)$. This cost term represents the difference between the objective function values of ϕ^C and ϕ, that is,

$$Cost(\phi, C) = z(\phi^C) - z(\phi) = 2 * \sum_{i \in C} \sum_{j=1}^{n} f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right) \qquad 2.3$$

Clearly, the cost of the k-exchange C can be computed in time $O(kn)$. We will show that using improvement graphs, the cost of C can be computed in $O(k^2)$ time. This time can be further reduced to an average of $O(k)$ time. Since we choose k to be much smaller than n, the improvement graph allows us to evaluate multi-exchange substantially faster than standard methods. In fact, it also leads to dramatic improvements in the running time to identify traditional 2-exchanges.

We associate an improvement graph $G(\phi) = (N, A)$ with ϕ, which is a directed graph comprising of the node set N and arc set A. The node set N contains a node i for every facility i, and the arc set A contains an arc (i, j) for every ordered pair of nodes i and j in N. Each multi-exchange $C = i_1 - i_2 - \ldots - i_k - i_1$ defines a (directed) cycle $i_1 - i_2$ -

…- $i_k - i_1$ in $G(\phi)$ and, similarly, each (directed) cycle $i_1 - i_2$ -…- $i_k - i_1$ in $G(\phi)$ defines a multi-exchange $i_1 - i_2$ - … - $i_k - i_1$ with respect to ϕ. Thus, there is one-to-one correspondence between multi-exchanges with respect to ϕ and cycles in $G(\phi)$. We will, henceforth, use C to denote both a multi-exchange and a cycle in $G(\phi)$, and its type will be apparent from the context.

An arc $(i, j) \in A$ signifies that the facility i moves from its current location to the current location of facility j. In view of this interpretation, a cycle $C = i_1 - i_2$ - … - $i_k - i_1$, signifies the following changes: facility i_1 moves from its current location to the location of facility i_2, facility i_2 moves from its current location to the location of facility i_3, and so on. Finally, facility i_k moves from its current location to the location of facility i_1.

We now associate a cost c_{ij}^{ϕ} with each arc $(i, j) \in A$. Ideally, we would like to define arc costs so that the cost of the multi-exchange C with respect to the permutation ϕ is equal to the cost of cycle C in $G(\phi)$. However, such a possibility would imply that P = NP because the multi-exchange neighborhood structure includes all feasible solutions for an instance of the QAP. We will, instead, define arc costs so that the cost of the multi-exchange is "close" to the cost of the corresponding cycle. We define c_{ij}^{ϕ} as follows: it is the change in the cost of the solution ϕ when facility i moves from its current location to the location of facility j and all other facilities do not move. Observe that this change indicates that after the change there is no facility at location $\phi(i)$ and the location $\phi(j)$ has two facilities. Thus, to determine the cost of the change, we need to take the difference between the costs of interactions between facility i and other facilities, before and after the change. Let ϕ' denote the solution after the change. Then, $\phi'(l) = \phi(l)$ for $l \neq i$ and $\phi'(i)$

$= \phi(j)$. Note that ϕ' is not a permutation because $\phi'(i) = \phi'(j)$. We define $c_{ij}^{\phi} = z(\phi') - z(\phi)$. Then,

$$c_{ij}^{\phi} = z(\phi') - z(\phi) = 2 * \sum_{l=1}^{n} f_{il} \left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \right), \qquad 2.4$$

which captures the change in the cost of interaction *from* facility *i* to other facilities.

The manner in which we define arc costs in the improvement graph does not ensure that the cost of the cycle C in $G(\phi)$, given by $\sum_{(i,j)\in C} c_{ij}^{\phi}$, will equal $Cost(\phi, C)$. The discrepancy in these two cost terms arises because when defining the arc cost c_{ij}^{ϕ} we assume that the facility *i* moves from its current location to the location of facility *j* but all other facilities do not move. But in the multi-exchange C several facilities move and we do not correctly account for the cost of flow between facilities in C. We, however, correctly account for the cost of flow between any two facilities if one of the two facilities is not in C. We show next that the cost term $Cost(\phi, C)$ can be computed by adding a corrective term to $\sum_{(i,j)\in C} c_{ij}^{\phi}$.

$$Cost(\phi, C) = z(\phi^C) - z(\phi) = 2 * \sum_{i\in C} \sum_{j=1}^{n} f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right)$$

$$= 2 * \sum_{i\in C} \sum_{j\notin C} f_{ij} \left(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)} \right) + 2 * \sum_{i\in C} \sum_{j\in C} f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right)$$

$$= 2 * \sum_{(i,j)\in C} c_{ij}^{\phi} - 2 * \sum_{i\in C} \sum_{j\in C} f_{ij} \left(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)} \right)$$

$$+ 2 * \sum_{i\in C} \sum_{j\in C} f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right). \qquad 2.5$$

The equation 2.5 shows that we can determine the cost of a multi-exchange C by first determining the cost of the cycle C in $G(\phi)$, which is $\sum_{(i,j)\in C} c_{ij}^{\phi}$, and then correcting

it using the second term given in equation 2.5. This corrective term can be computed in time $O(k^2)$ if C is a k-exchange.

Let us now remark on the usefulness of the improvement graph. First, it allows us to determine the approximate cost of a multi-exchange C quickly. The cost term $\sum_{(i,j)\in C} c_{ij}^{\phi}$ is a reasonable estimate of the cost of the multi-exchange C. To see this, observe from equation 2.5 that the corrective term $Cost(\phi, C) - \sum_{(i,j)\in C} c_{ij}^{\phi}$ contains $O(k^2)$ interactions between facilities. However, n facilities have $O(n^2)$ interactions between them. If we choose k to be a relatively small fraction of n, then the corrective term (on the average) will be substantially smaller than the total cost and the cost of the cycle C in $G(\phi)$ will be a good estimate of the cost of the multi-exchange C. For example, if $n = 100$ and $k = 5$, then there are *9,900* interactions between facilities and only *20* of them are counted incorrectly. If we use $k = 10$, then about *100* of them are counted incorrectly which is only 1% of the total interactions between facilities. Thus, the improvement graph allows us to enumerate extremely large set of neighbors quickly using approximate costs, and the approximation in costs is quite small.

The improvement graph also allows us to determine the correct cost of a multi-exchange faster than it normally takes to compute its cost. Normally, to compute the cost of a multi-exchange takes $O(kn)$ time as we would need to update the cost interactions between k facilities (that move) with other facilities. However, using equation 2.5 we can compute the cost of a multi-exchange in $O(k^2)$ time. For example, if $n = 100$ and $k = 10$, then we can compute the cost of a multi-exchange about *10* times faster which can make substantial difference in an algorithm's performance.

The benefits we derive from the use of improvement graph come at a cost; we need to construct the improvement graph and calculate arc costs. It follows from equation 2.4 that we can construct the improvement graph from scratch in $O(n^3)$ time. But we need to compute the improvement graph from scratch just once. In all subsequent steps, we only *update* the improvement graph as we perform multi-exchanges. We show in the next lemma that updating the improvement graph following a k-exchange takes only $O(kn^2)$ time. We also show in Section 2.8 that our neighborhood search algorithms need to use k *(4* and *5*) only as on the benchmark tests higher values do not add extra benefit. Hence, it takes $O(n^2)$ time to update the improvement graph, which is quite efficient in practice. Thus, the time needed to construct and update the improvement graph is relatively small, and is well justified by the savings we obtain in enumerating and evaluating multi-exchanges.

Lemma 1: *Given the improvement graph $G(\phi)$ and a k-exchange C with respect to ϕ, the improvement graph $G(\phi^C)$ can be constructed in $O(kn^2)$ time.*

Proof: The improvement graphs $G(\phi)$ and $G(\phi^C)$ have the same set of nodes and arcs. They differ only in arc costs. Each arc $(i, j) \in G(\phi^C)$ is one of the following two types: (i) either $i \in C$ or $j \in C$, and (ii) $i \notin$ C and $j \notin$ C. There are $2k(n - k) = O(nk)$ arcs of type (i), and $O(n^2)$ arcs of type (ii). Using equation 2.5, we can determine the cost of a type (i) arc in $O(n)$ time, thus giving a total time of (n^2k) to compute the cost of all type (i) arcs. We show next that we can determine the cost of a type (ii) arc in $O(k)$ time, which also yields a total time of $O(n^2k)$ to compute the costs of all type (ii) arcs.

$$c_{ij}^{\phi^C} = 2 * \sum_{l=1}^{n} f_{il} \left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)} \right)$$

$$= 2 * \sum_{l \notin C} f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right) + 2 * \sum_{l \in C} f_{il}\left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}\right)$$

$$= 2 * \sum_{l=1}^{n} f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right) - 2 * \sum_{l \in C} f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right)$$

$$+ 2 * \sum_{l \in C} f_{il}\left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}\right)$$

$$= 2 * c_{ij}^{\phi} - 2 * \sum_{l \in C} f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right) + 2 * \sum_{l \in C} f_{il}\left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}\right). \qquad 2.6$$

Since we already know c_{ij}^{ϕ} and C is a k-exchange, we can evaluate equation 2.6 in O(k) time, which establishes the lemma.

2.3 Identifying Profitable Multi-Exchanges

Our algorithm for the QAP is a neighborhood search algorithm and proceeds by performing profitable multi-exchanges. To keep the number of multi-exchanges enumerated manageable, we first enumerate 2-exchanges, followed by 3-exchanges, and so on, until we reach a specified value of k, denoted by K, which is the largest size of the multi-exchanges we wish to perform. This enumeration scheme is motivated by the consideration that we look for larger size multi-exchanges when smaller size multi-exchanges cannot be found. In this section, we describe a generic search procedure for enumerating and identifying multi-exchanges using improvement graphs.

Our method for enumerating multi-exchanges with respect to a solution ϕ proceeds by enumerating directed paths of increasing lengths in the improvement graph $G(\phi)$, where the length of a path is the number of nodes in the path. Observe that each path $P = i_1 - i_2 - \ldots - i_k$ in the improvement graph has a corresponding cycle in the improvement graph $i_1 - i_2 - \ldots - i_k - i_1$ obtained by joining the last node of the path with

the first node in the path; this cycle also defines a multi-exchange with respect to ϕ. Let $C(P)$ denote the multi-exchange defined by the path P.

Our method for enumerating cycles of increasing lengths performs the following three steps repeatedly for increasing values of k, starting with $k = 2$. Let S^k denote a set of some paths of length k in $G(\phi)$. We start with $S^1 = \{1, 2, \ldots, n\}$, which is the set of n paths of length 1, each consisting of a singleton node.

Path extension: We consider each path $P \in S^{k-1}$ one by one and "extend" it by adding one node to it. To extend a path $P = i_1 - i_2 - \ldots - i_{k-1}$, we add the arc (i_{k-1}, i_k) for each $i_k \in N \backslash \{i_1, i_2, \ldots, i_k\}$, and obtain several paths of length k. Let $E(P)$ denote the set of all paths obtained by extending the path P. Further, let $\Pi^k = \bigcup_{P \in S^{k-1}} E(P)$.

Cycle evaluation: Each path $P \in \Pi^k$ yields a corresponding multi-exchange $C(P)$. We evaluate each of these multi-exchanges and determine whether any of them is a profitable multi-exchange. If yes, we return the best multi-exchange and stop; otherwise we proceed further.

Path pruning: In this step, we prune several paths in the set Π^k, which are less likely to lead to profitable multi-exchanges. We call a procedure, *PathSelect*(Π^k), that takes as an input the set of paths Π^k enumerated in the previous step and selects a subset S^k of it. This subset of paths will be extended further in the next iteration for the next higher value of k. We describe in Section 2.4 several ways to implement the *PathSelect* procedure. Path pruning is critical to keep the number of paths enumerated manageable.

The following algorithmic description summarizes the steps of our heuristic search procedure, which we call the *K-exchange search* procedure.

procedure *K-exchange search*;
begin
$k \leftarrow 1$;
let $S^1 \leftarrow N$ be the set of paths of length 1;
$C^* \leftarrow \phi$ and $W^* \leftarrow 0$;
while S^k is non-empty **and** $k < K$ **and** $W^* \geq 0$ **do**
begin
$k \leftarrow k+1$;
$\Pi^k \leftarrow \bigcup_{P \in S^{k-1}} E(P)$;
let $P_{min} \in S^k$ be the path such that $Cost(\phi, C(P_{min})) = \min\{Cost(\phi, C(P)): P \in \Pi^k\}$;
if $W^* > Cost(\phi, C(P_{min}))$ **then** $W^* \leftarrow Cost(\phi, C(P_{min})$ and $C^* \leftarrow C(P_{min})$;
$S^k \leftarrow PathSelect(\Pi^k)$;
end;
return C^*;
end.

Figure 2.2 The generic search procedure for identifying profitable multi-exchanges.

Observe that in this procedure, the value of K is a parameter and can be specified by the user. Increasing the value of K may in general improve the quality of local optimal solutions obtained, but our computational investigations show that there are diminishing returns after $K = 4$; hence $K = 4$ is a good value to be used in the search procedure. For another implementation (Implementation 4) of *PathSelect* as discussed in Section 2.4, we keep the value of $K = 5$. Also observe that the algorithm terminates in two ways: C^* is empty or C^* is nonempty. If C^* is empty, then it implies that the algorithm has failed to find a profitable multi-exchange and the current solution ϕ is locally optimal. If C^* is nonempty, then it implies that the algorithm found a profitable multi-exchange C^*.

We now analyze the complexity of the algorithm. Let p denote the maximum number of paths in any S^k. The *while* loop executes at most K times. In each execution of the while loop, it takes $O(pn)$ time to compute the set Π^k and it may contain as many as pn paths. Since computing the cost of k-exchange for each $P \in \Pi^k$ takes $O(k^2)$ time, we require $O(k^2pn)$ time to find a profitable k-exchange, if any. We shall show in Section 2.4 that the subroutine *PathSelect* takes $O(pn \log(pn))$ time. Since for most situations considered by us $\log(pn) < k^2$, the running time of the algorithm is $O(k^2pn)$.

It is easy to see that if we ignore the time taken by the procedure *PathSelect*, then the bottleneck operation in the generic search procedure is to evaluate the cost $Cost(\phi, C(P))$ of each path $P \in \Pi^k$. Since $C(P)$ is a k-exchange with respect to the solution ϕ, using equation 2.6 we can determine its cost in $O(k^2)$ time. We will next show that we can determine the cost of k-exchange $C(P)$ in $O(k)$ time.

The generic search procedure proceeds by enumerating paths in $G(\phi)$. Each path $P = i_1 - i_2 - \ldots - i_k$ in $G(\phi)$ defines a "path exchange" with respect to the solution ϕ in an obvious manner, which is the same as the k-exchange $C = i_1 - i_2 - \ldots - i_k - i_1$ except that we do not perform the last move of shifting facility i_k from its current location to the location of facility i_1. Alternatively, $\phi^P(i_l) = \phi(i_{l+1})$ for all $l = 1, 2, \ldots, k\text{-}1$, and $\phi^P(i) = \phi(i)$ for all $i \in N \setminus \{i_1, i_2, \ldots, i_{k-1}\}$. We denote the cost of the path exchange P with respect to the solution ϕ by $Cost(\phi, P)$. Hence,

$$Cost(\phi, P) = z(\phi^P) - z(\phi) = 2 * \sum_{i \in P} \sum_{j=1}^{n} f_{ij} \left(d_{\phi^P(i)\phi^P(j)} - d_{\phi(i)\phi(j)} \right). \qquad 2.7$$

Observe that ϕ^P and $\phi^{C(P)}$ differ only in the location of the facility i_k. This observation allows us to compute the cost of the cyclic exchange $C(P)$ from the cost of the path exchange P in O(k) time using the following expression:

$$Cost(\phi, C(P)) - Cost(\phi, P) = 2 * c^{\phi}_{i_k i_1} + 2 * \sum_{j \in C} f_{i_k j} \left(\left(d_{\phi(i_1)\phi^C(j)} - d_{\phi(i_k)\phi^P(j)} \right) - \left(d_{\phi(i_1)\phi(j)} - d_{\phi(i_k)\phi(j)} \right) \right). \quad 2.8$$

Now suppose that we extend the path P to $P' = i_1 - i_2 - \ldots - i_k - i_{k+1}$ by adding the node i_{k+1}. Then, we can determine the cost of the path P' from the cost of the path P in O(k) time using the following expression:

$$Cost(\phi, P') - Cost(\phi, P) = 2 * c^{\phi}_{i_k i_{k+1}} + 2 * \sum_{j \in P} f_{i_k j} \left(\left(d_{\phi(i_{k+1})\phi^{P'}(j)} - d_{\phi(i_k)\phi^P(j)} \right) - \left(d_{\phi(i_{k+1})\phi(j)} - d_{\phi(i_k)\phi(j)} \right) \right). \quad 2.9$$

In our enhanced version, we maintain the cost of each path P enumerated by the algorithm. Given the cost of path P, we can determine the cost of the cycle $C(P)$ in O(k) time. Further, when we extend any path P, then the cost of the extended path too can be computed in O(k) time. Thus, the running time of the generic search procedure is O($K \sum_{k=2}^{K} |P^k|$), plus the time taken by the subroutine *PathSelect*.

2.4 Specific Implementations

In Section 2.3, we presented a generic search algorithm to identify a profitable multi-exchange. We can derive several specific implementations of the generic version by implementing the procedure *PathSelect*(Π^k) differently. The procedure *PathSelect*(Π^k)

accepts as an input a set of paths Π^k and returns a subset S^k of these paths. We describe next several ways in which *PathSelect* can be implemented.

Implementation 1 (all paths): In this version, we define *PathSelect*(Π^k) to be Π^k itself; that is, we select all the paths to be taken to the next stage. This version guarantees that we will always find a profitable multi-exchange if it exists. However, the number of paths enumerated by the algorithm increase exponentially with k and it takes too long to find profitable k-exchanges for $k \geq 6$ even for $n = 25$.

Implementation 2 (negative paths): In this version, the subroutine *PathSelect*(Π^k) returns only those paths which have negative cost; that is, *PathSelect*(Π^k) = $\{P \in \Pi^k$: *Cost*(ϕ, P) < 0$\}$ where ϕ is the current solution. This version is motivated by the intuition that if there is a profitable multi-exchange $C = i_1 - i_2$ - … - $i_k - i_1$, then there should exist a node in this sequence, say node i_l, so that each of the paths $i_l - i_{l+1}$, $i_l - i_{l+1} - i_{l+2}$, …, $i_l - i_{l+1} - i_{l+2}$ - … - i_{l+k} has a negative cost. Though results of this type are valid for many combinatorial optimization problems, it is *not* true for the QAP. However, it is a reasonable heuristic to eliminate paths that are less likely to yield profitable multi-exchanges.

Implementation 3 (best αn^2 paths): In this version, we sort all the paths in Π^k in the non-decreasing order of path costs, and select the first αn^2 paths, where α is a specified constant. For example, if $\alpha = 2$, then we select the best $2n^2$ paths. This version is motivated by the intuition that the paths with lower cost are more likely to yield profitable multi-exchanges. The choice of α allows us to strike a right tradeoff between the running time and the solution quality. Higher values of α will increase the chances of finding profitable multi-exchanges but also increase the time needed to find a profitable

multi-exchange. Our computational results presented in Section 2.8 indicate that $\alpha = 1$ is a good choice considering both the running time and solution quality. We have used max heap data structure to keep αn^2 paths in a stage. Hence if there are pn possible paths (as discussed in Section 2.3), it takes $pn\log(pn)$ time to store αn^2 best paths in a heap.

Implementation 4 (best *n* paths): In this implementation, we select the best path in Π^k starting at node i for each $1 \leq i \leq n$. Therefore, the set S^k contains at most one path starting at each node in N. Note that in Implementation 3, it is possible that many low cost paths contain the same set of arcs making the search less diverse. Allowing each node to be the starting point of a different path can add some diversity to the heuristic search process.

We implemented Implementation 3 and 4 of *PathSelect* on the test problems as our experiment on the test problems indicates that it is difficult to use Implementation 1 even for moderate size problems and in all cases Implementation 3 gives better result than Implementation 2. We present in Section 2.8 the computational results of these implementations.

2.5 The Neighborhood Search Algorithm

In this section, we describe our neighborhood search algorithm (Figure 2.3) for the QAP. Our algorithm starts with a random permutation (obtained by generating pseudorandom numbers between 1 and n and rejecting the numbers already generated) and successively improves it by performing profitable multi-exchanges obtained by using the K-exchange search procedure, until the procedure fails to produce a profitable multi-exchange.

```
algorithm QAP-neighborhood-search;
begin
    generate an initial random permutation ϕ;
    construct the improvement graph G(ϕ);
    while K-exchange search returns a non-empty multi-exchange C do
    begin
        replace the permutation ϕ by the permutation ϕ^C;
        update the improvement graph;
    end;
    return the permutation ϕ;
end;
```

Figure 2.3 The neighborhood search algorithm for the QAP.

Let us perform the running time analysis of the algorithm. The initial construction of the improvement graph takes $O(n^3)$ time. The time needed by the procedure *K*-exchange search is $O(K^2p)$, where p is the maximum number of paths maintained by the procedure during any iteration (see Section 2.3). For Implementation 3 of *PathSelect*, $p \leq \alpha n^2$ and this procedure requires $O(K^2n^2)$ time per iteration (that is, per improvement). For the Implementation 4 of the *PathSelect*, $p \leq n^2$, and the procedure again takes $O(K^2n^2)$ time. Updating the improvement graph takes $O(n^2K)$ time (see Section 2.2).

Each execution of the *QAP-neighborhood-search* algorithm yields a locally optimal solution of the QAP with respect to the neighborhood defined by the *K-exchange search* procedure. The solution obtained depends upon the initial random permutation ϕ and the version of the *PathSelect* procedure we use. We refer to one execution of the algorithm as one *run*. Our computational investigations revealed that if we apply only one run of the algorithm, then the solution method is not very robust. The QAP in general has an extremely large number of locally optimal solutions even if the size of the neighborhood is very large. Each run produces a locally optimal solution, which is a

random sample in the solution space of locally optimal solutions. To obtain a robust locally optimal solution, we need to perform several runs of the algorithm and use the best locally optimal solution found in these runs.

2.6 Accelerating the Search Algorithm

In this section, we describe a method to speedup the performance of the generic search algorithm and also its specific implementations. The speedup uses the fact that several paths give the same multi-exchange. For example, all the paths $i_1 - i_2 - i_3 - i_4$, $i_2 - i_3 - i_4 - i_1$, $i_3 - i_4 - i_1 - i_2$, and $i_4 - i_1 - i_2 - i_3$ imply the same multi-exchange $i_1 - i_2 - i_3 - i_4 - i_1$ when we connect the last node of these paths to the first node of the path. In general, a k-exchange can be represented by k different paths. Since our generic search algorithm enumerates k-exchanges by enumerating paths, we may obtain the same k-exchange several times during the search process through different paths. To avoid repeated enumeration of the multi-exchanges, our search algorithm maintains certain kinds of paths, called *valid paths*, defined as follows:

Valid Paths: *A path $i_1 - i_2$ -...- i_k is a valid path if $i_1 \leq i_j$ for every $2 \leq j \leq k$.*

Our generic search algorithm enumerates only valid paths. The following lemma shows that we do not miss any multi-exchanges by maintaining valid paths only.

Lemma 2. *Any multi-exchange can be enumerated by maintaining only valid paths.*

Proof: Consider a multi-exchange $j_1 - j_2$ -…- j_k. Let $j_l = \min \{j_h: 1 \leq h \leq k\}$. Now define $i_1 = j_l$, $i_2 = j_{l+1}$, …, $i_k = j_{l+k}$, where all subscript mathematics is modulo (k+1). It follows from the definition of j_l that each of the paths i_1, $i_1 - i_2$, $i_1 - i_2 - i_3$, … , $i_1 - i_2$ -…- i_k is a valid path. Hence starting at node i_1 we can gradually build $i_1 - i_2$ -…- i_k by

maintaining only valid paths, and joining node i_k to node i_1 gives us the desired multi-exchange. ♦

We can easily modify the generic search algorithm so that it only enumerates valid paths. In this modified algorithm, when we consider adding the arc (i_k, i_{k+1}) to the path $i_1 - i_2$ -…- i_k, we compare i_1 with i_{k+1}. If $i_1 \leq i_{k+1}$, we add the arc; otherwise we do not add it. It can be noted that above lemma holds if we enumerate all paths. However, as we keep only αn^2 paths in each stage, there may be the cases when we might miss a profitable multi-exchange. Our experiment shows that losses in missed improvements are well compensated by the gain in time. The computational results presented in Section 2.8 show that enumerating only valid paths decreases the running time of the generic search algorithm substantially.

2.7 Expressions for the Asymmetric QAP

In the previous sections, we gave expressions for calculating various cost terms for symmetric instances of QAP. In this section, we give expressions for the asymmetric QAO. We state the expressions without proof since the logic is similar to those for the symmetric case.

For the asymmetric case, we will replace the expressions 2.3 - 2.9 by the following expressions 2.10a – 2.10g respectively.

$Cost(\phi, C) = z(\phi^C) - z(\phi)$

$$= \sum_{i \in C} \sum_{j=1}^{n} \left(f_{ij} \left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)} \right) + f_{ji} \left(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)} \right) \right) \qquad 2.10a$$

$$c_{ij}^{\phi} = z(\phi') - z(\phi) = \sum_{l=1}^{n} \left(f_{il} \left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)} \right) + f_{li} \left(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)} \right) \right). \qquad 2.10b$$

$$Cost(\phi, C) = \sum_{(i,j)\in C} c_{ij}^{\phi} - \sum_{i\in C}\sum_{j\in C}\left(f_{ij}\left(d_{\phi^C(i)\phi(j)} - d_{\phi(i)\phi(j)}\right) + f_{ji}\left(d_{\phi(j)\phi^C(i)} - d_{\phi(j)\phi(i)}\right)\right) +$$

$$\sum_{i\in C}\sum_{j\in C} f_{ij}\left(d_{\phi^C(i)\phi^C(j)} - d_{\phi(i)\phi(j)}\right) + \sum_{i\in C}\sum_{j\in C} f_{ji}\left(d_{\phi^C(j)\phi^C(i)} - d_{\phi(j)\phi(i)}\right) \quad 2.10c$$

$$c_{ij}^{\phi^C} = c_{ij}^{\phi} - \sum_{l\in C}\left(f_{il}\left(d_{\phi(j)\phi(l)} - d_{\phi(i)\phi(l)}\right) + f_{li}\left(d_{\phi(l)\phi(j)} - d_{\phi(l)\phi(i)}\right)\right) +$$

$$\sum_{l\in C}\left(f_{il}\left(d_{\phi(j)\phi^C(l)} - d_{\phi(i)\phi^C(l)}\right) + f_{li}\left(d_{\phi^C(l)\phi(j)} - d_{\phi^C(l)\phi(i)}\right)\right) \quad 2.10d$$

$$Cost(\phi, P) = z(\phi^P) - z(\phi) = \sum_{i\in P}\sum_{j=1}^{n}\left(f_{ij}\left(d_{\phi^P(i)\phi^P(j)} - d_{\phi(i)\phi(j)}\right) + f_{ji}\left(d_{\phi^P(j)\phi^P(i)} - d_{\phi(j)\phi(i)}\right)\right) \quad 2.10e$$

$$Cost(\phi, C(P)) - Cost(\phi, P) = c_{i_k i_1}^{\phi} + \sum_{j\in C} f_{i_k j}\left(\left(d_{\phi(i_1)\phi^C(j)} - d_{\phi(i_k)\phi^P(j)}\right) - \left(d_{\phi(i_1)\phi(j)} - d_{\phi(i_k)\phi(j)}\right)\right) +$$

$$\sum_{j\in C} f_{ji_k}\left(\left(d_{\phi^C(j)\phi(i_1)} - d_{\phi^P(j)\phi(i_k)}\right) - \left(d_{\phi(j)\phi(i_1)} - d_{\phi(j)\phi(i_k)}\right)\right) \quad 2.10f$$

$$Cost(\phi, P') - Cost(\phi, P) = c_{i_k i_{k+1}}^{\phi} + \sum_{j\in P} f_{i_k j}\left(\left(d_{\phi(i_1)\phi^{P'}(j)} - d_{\phi(i_k)\phi^P(j)}\right) - \left(d_{\phi(i_1)\phi(j)} - d_{\phi(i_k)\phi(j)}\right)\right) +$$

$$\sum_{j\in P} f_{ji_k}\left(\left(d_{\phi^{P'}(j)\phi(i_1)} - d_{\phi^P(j)\phi(i_k)}\right) - \left(d_{\phi(j)\phi(i_1)} - d_{\phi(j)\phi(i_k)}\right)\right) \quad 2.10g$$

2.8 Computational Testing

In this section, we describe computational results of the neighborhood search algorithms developed by us. We implemented all of our algorithms in C and ran them on IBM SP machine (model RS6000) with a processor speed of 333 MHz. We tested the algorithms on 132 benchmark instances available at the QAPLIB, the library of QAP instances maintained by the Institute of Mathematics, Graz University of Technology (http://www.opt.math.tu-graz.ac.at/qaplib/). Our computational results include analyzing the

CPU times taken by our algorithms, quality of the solutions obtained by them as well as understanding the behavior of the VLSN search algorithms.

Neighborhood search algorithms need some feasible solution as the starting solution. We generated random permutations of n numbers and used them as starting solutions. Further we implemented a multi-start version of the neighborhood search algorithm, where we apply the neighborhood search algorithm multiple times with different starting solutions, called different *runs,* and select the best solution found in these runs that the number of runs depend on the size of the problem instance.

In Section 2.4, we proposed four implementations of the generic VLSN search algorithm for the QAP. The first implementation maintains all the paths enumerated in the search process. We found that the number of paths grows very quickly with k and the algorithm runs very slowly even when we go up to k-exchanges with $k = 6$. For example to solve a QAP with $n = 42$ (problem sko42), each run of this implementation takes about 8 seconds for $k = 4$ whereas Implementation 3 takes only 0.025 seconds per run. Additional preliminary tests yielded that this implementation is not as competitive as other implementations and we decided not to perform a thorough testing of the algorithm.

In the second implementation of the VLSN search algorithm, we maintain only those paths, which have negative costs. For many combinatorial optimization problems, maintaining only negative cost paths is sufficient to enumerate negative cost cycles (improved neighbors), but this is not true for the QAP due to the non-linearity in the cost structure. Our computational testing revealed that maintaining only negative cost paths is not a good heuristic to enumerate negative cost cycles. Thus, we did not perform a thorough testing of this implementation.

Our preliminary testing revealed that Implementation 3 and 4 exhibited the best overall behavior and deserved a thorough testing. The following details of implementation 3 are worth mentioning. Recall from Section 2.4 that we keep only αn^2 best paths in Π^k. We used the Max Heap data structure (Cormen et al. [2001]) to store these paths. We found that $\alpha = 1$ gives fairly good results and hence used this value. In addition, we used only those paths whose path cost is not more than 0.5% of the best objective function value of the QAP found so far. We found that using higher cost paths rarely leads to negative cost cycles. Finally, when we examined paths in Π^k to enumerate cycles of length k and find several negative cost cycles, we use the least cost negative cycle to obtain the next solution. As far as Implementation 4 is concerned, we implemented it in the straightforward fashion but before enumerating paths, we eliminate all negative cycles of length 2 by performing 2-exchanges.

2.8.1 Accuracy of the Solution

We applied Implementation 3 and 4 to 132 benchmark instances in QAPLIB, of these 98 were instances of symmetric QAP and the remaining were for the asymmetric case. We applied multiple runs of each implementation and ran them for a specified amount of time. For the symmetric instances, we ran our algorithm for 1 hour for $n \leq 40$ and for 2 hours for $n > 40$. The running times for the asymmetric instances were 1.5 hours for $n \leq 40$ and for 3 hours for $n > 40$. Tables 1 & 2 in Appendix - 1, respectively, give the results of these algorithms for symmetric and asymmetric instances and compare our solutions with the solutions obtained by the 2-exchange algorithm (2OPT) and the best known solutions (BKS). The columns titled BestGap, AvgGap, nRuns, %Best, respectively, give the percent deviation of the best solution found in all runs with respect

to the best known solution, average deviation over solutions found in all runs, number of runs, and the percentage of the solutions found were best known solutions. User can derive the following conclusions from these tables.

- Implementation 3 exhibited the best overall performance. It obtained the best known solutions in 74 out of 98 symmetric instances and in 24 out of 34 asymmetric instances. Its average error was the lowest and it found the best known solutions with the maximum frequency.

- Implementation 3 is found to exhibit superior performance compared to 2OPT in terms of gap of best solution found by algorithm with the best known solution. For 25 symmetric instances implementation 3 obtained better solutions than 2OPT, and for only 2 symmetric instances 2OPT obtained better solutions than Implementation 3. Similarly, for 10 asymmetric instances implementation 3 obtained better solutions than 2OPT, and for only 1 asymmetric instance 2OPT obtained better solutions than Implementation 3.

- Implementation 3 was also found to be better than 2OPT in terms of average gap and frequency of finding best known solution. The average of AvgGap of Implementation 3 was 7.6%, whereas this number for 2OPT was 11.05% for symmetric instances and these numbers were 6.42% and 7.49% respectively for asymmetric instances. Finally, whereas Implementation 3 found best known solution with an average frequency of 17.13% in symmetric case, this number for 2OPT was 11.85% in symmetric case. For asymmetric case, Implementation 3 found best known solution with an average frequency of 1.97%, whereas this number for 2OPT is 0.46%.

- Implementation 4 also exhibited superior performance with respect to 2OPT, but its overall performance was worse than Implementation 3. Implementation 4 runs very fast and it terminates in a fraction of second for most problem sizes, but the solutions obtained using this method are not as robust as those obtained using Implementation 3.

Above results seem to suggest that very large-scale neighborhood is overall more effective than the traditional 2-exchange neighborhood. When both the algorithms are run for the same time, the 2OPT performs many more runs but still the best solution found is, on the average, not as good as found by VLSN search in fewer number of runs. Hence the extra time taken by VLSN search algorithm is more than justified by the better quality of the solutions obtained.

We will now describe some computational investigations we performed to understand the behavior of our implementations.

2.8.2 Effect of Neighborhood Size

In our approach, the size of the neighborhood critically depends upon (i) the maximum cycle length, and (ii) the number of paths, of a given length, maintained. The larger the cycle length and the number of paths maintained, larger is the neighborhood, more is the running time, and better is the quality of the solution obtained (in general). Hence it is worthwhile to examine the effect of these two parameters on the running time and the solution quality.

In our first experiment, we considered six problems of the same size; sko100a, sko100b, sko100c, sko100d, sko100e, sko100f, and applied 100 runs of Implementation 3 with cycle lengths varying from 2 to 7 and noted the average running time taken by the algorithm (per run) and the average gap (per run). We kept the number of paths maintained by the algorithm as fixed at n^2. Figure 2.4 plots these two values as a function of cycle length. It can be seen that the average gap decreases significantly with the increase in cycle length until cycle length is 4, and after that the average gap does not vary much. We also observe that the running time of the algorithm increases linearly with the increase in the cycle length. We think that the cycle length of 4 strikes a right balance between the solution accuracy and solution time and hence we used this value in the computational results presented earlier.

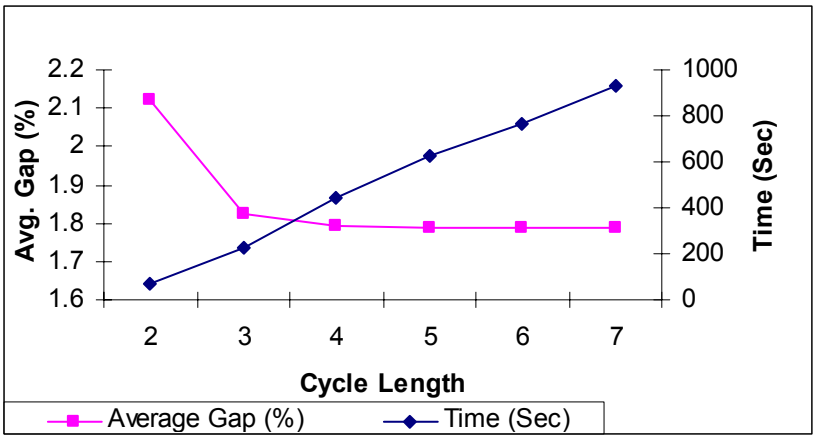


Figure 2.4 Effect of cycle length on time taken and solution quality for 100 runs on problem sko100a-f

Our second experiment was similar to the first experiment but we varied the number of paths maintained by the algorithm while keeping the cycle length fixed at 4. Figure 2.5 gives a plot of the average gap and average time per run when we performed 100 runs of Implementation 3 on the six problems sko100a-f. We observe that the solution accuracy gradually improves as the number of paths increase as well as the running time of algorithm increases linearly with the number of paths maintained. We believe that maintaining n^2 paths is a good compromise between solution quality and solution time and we used this value in our experiments.

In another experiment, we counted the number of improvement iterations with cycle length 2, 3 and 4. Recall that our algorithm performs a 3-exchange when it fails to find 2-exchange, and performs a 4-exchange when it fails to find a 3-exchange. Table 2.1 gives these values for 10 benchmark instances on which we apply 100 runs of Implementation 3. We observe that there are many more iterations with 2-exchanges compared to 3-exchanges, and many more 3-exchanges compared to 4-exchanges.

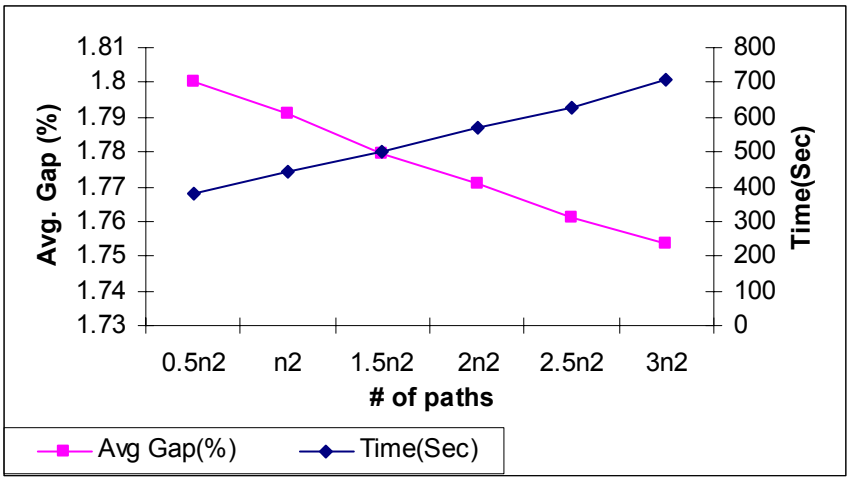


Figure 2.5 Effect of number of paths in each stage on time taken and solution quality for 100 runs on problem sko100a-f

Table 2.1 Number of iterations with different implemented cycle length

	# of iterations of Cycle Length		
Problem	**2**	**3**	**4**
chr22a	1614	151	49
kra30a	2283	123	30
kra30b	2306	125	32
nug30	2580	104	42
ste36a	3537	142	44
tho40	3839	124	35
wil50	5298	81	38
sko42	4246	150	64
sko100a	13232	270	70
tai100a	7267	274	75

2.8.3 Effect of the Speedup Technique

The reader may recall from Section 2.6 that we used a speedup technique to reduce redundant enumeration of cycles. In this technique, we maintain only those valid paths $i_1, i_2, \ldots, i_k$ for which $i_k > i_1$. Lemma 2 showed that we would not miss any negative cycles even if we only maintain valid paths. This proof relied on the assumption that we maintain *all* valid paths. Since our algorithm maintains only n^2 paths, we might miss

some negative cycles and the speedup technique may deteriorate the quality of the solutions obtained. We performed an experiment to assess the effect of the speedup technique on the solution quality and solution time. Table 2.2 gives these values for 10 benchmark instances. We applied 100 runs on each benchmark instance and obtained the average values. We observe that speedup technique not only decreases the running time substantially but also worsens the solution quality. We believe that overall it is advantageous to use the speedup technique since the saved time can be used to perform more runs of the algorithm and improve the overall performance of the algorithm.

Table 2.2 Effect of accelerated path enumeration scheme

	Using Speedup Technique		**Without Speedup Technique**	
Problem	**Average Gap**	**Time in Seconds**	**Average Gap**	**Time in Seconds**
Chr22a	10.00	1	9.13	5
Kra30a	6.44	5	6.27	12
Kra30b	4.26	5	4.05	12
Nug30	3.19	5	2.92	12
Ste36a	9.34	10	8.37	27
Tho40	3.87	12	3.76	28
Wil50	1.54	24	1.41	70
Sko42	2.68	17	2.57	40
Sko100a	1.87	283	1.78	962
tai100a	2.88	280	2.48	1191

2.9 Conclusions

In this chapter, we develop a very large-scale neighborhood structure for the QAP. We show that using the concept of improvement graph, we can easily and quickly enumerate multi-exchange neighbors of a given solution. We develop a generic search procedure to enumerate and evaluate neighbors and propose several specific implementations of the generic procedure. We perform extensive computational

investigations of our implementations and they suggest that multi-exchange neighborhoods add value over the commonly used 2-exchange neighborhoods.

Our implementations of multi-exchange neighborhood search algorithms are local improvement methods. We wanted the focus of our research effort more on neighborhood structure and less on specific implementations. Further possibilities for improvement could possibly be obtained using ideas from tabu search (Glover and Laguna [1997]). We leave it as a topic of future research. Neighborhood search algorithms have also been used in genetic algorithms to improve the quality of the individuals in the population (Ahuja, Orlin, and Tiwari [2000], and Drezner [2003]). Our neighborhood structure may be useful in these genetic algorithms too.

CHAPTER 3
WEAPON-TARGET ASSIGNMENT PROBLEM

3.1 Introduction

The *Weapon-Target Assignment* (WTA) problem is a fundamental problem arising in defense-related applications of operations research. The problem consists of optimally assigning weapons to the enemy-targets so that the total expected survival value of the targets after all the engagements is minimized. There are two versions of the WTA problem: *static* and *dynamic*. In the static version, all the inputs to the problem are fixed; that is, all targets are known, all weapons are known, and all weapons engage targets in a single stage. The dynamic version of the problem is a multi-stage problem where some weapons are engaged at the targets at a stage, the outcome of this engagement is assessed and strategy for the next stage is decided. In this chapter, we study the static WTA problem; however, our algorithms can be used as important subroutines to solve the dynamic WTA problem.

We now give a mathematical formulation of the WTA problem. Let there be n targets, numbered 1, 2, …, n, and m weapon types, numbered 1, 2, …, m. Let V_j denote the value of the target j, and W_i denote the number of weapons of type i available to be assigned to targets. Let p_{ij} denote the probability of destroying target j by a single weapon of type i. Hence $q_{ij}=1 - p_{ij}$ denotes the probability of survival of target j if a single weapon of type i is assigned to it. Observe that if we assign x_{ij} number of weapons of type i to target j, then the survival probability of target j is given by $q_{ij}^{x_{ij}}$. A target may be

assigned weapons of different types. The WTA problem is to determine the number of weapons x_{ij} of type i to be assigned to target j to minimize the total expected survival value of all targets. This problem can be formulated as the following nonlinear integer-programming problem:

$$\text{Minimize } \sum_{j=1}^{n} V_j \left(\prod_{i=1}^{m} q_{ij}^{x_{ij}} \right) \qquad 3.1a$$

Subject to

$$\sum_{j=1}^{n} x_{ij} \le W_i, \quad \text{for all } i = 1, 2, \ldots, m, \qquad 3.1b$$

$$x_{ij} \ge 0 \text{ and integer, for all } i = 1, 2, \ldots, m, \text{ and for all } j = 1, 2, \ldots, n. \qquad 3.1c$$

In the above formulation, we minimize the expected survival value of the targets while ensuring that the total number of weapons used is no more than those available. This formulation presents a simplified version of the WTA problem. In more practical versions, we may consider adding additional constraints, such as (i) lower and/or upper bounds on the number of weapons of type i assigned to a target j; (ii) lower and/or upper bounds on the total number of weapons assigned to target j; or (iii) a lower bound on the survival value of the target j. The algorithms proposed in this chapter can be easily modified to handle these additional constraints.

Research on the WTA problem dates back to the 1950s and 1960s where the modeling issues for the WTA problem was investigated (Manne [1958], Braford [1961], Day [1966]). Lloyd and Witsenhausen [1986] established the NP-completeness of the WTA problem. Exact algorithms have been proposed to solve the WTA problem for the following special cases: (i) when all the weapons are identical (DenBroder et al. [1958] and Katter [1986]) or (ii) when the targets can receive at most one weapon (Chang et al.

[1987] and Orlin [1987]). Some of the heuristics proposed to solve the WTA problem are based on nonlinear network flow (Castanon et al. [1987]), neural networks (Wacholder [1989]), and genetic algorithms (Grant et al. [1993]). Green et al. [1997] applied a goal programming-based approach to the WTA problem. Metler and Preston [1990] have studied a suite of algorithms for solving the WTA problem efficiently, which is critical for real-time applications of the WTA problem. Maltin [1970], Eckler and Burr [1972] and Murphey [1999] provide comprehensive reviews of the literature on the WTA problem. Research to date on the WTA problem either solves the WTA problem for special cases or develops heuristics for the WTA problem. Moreover, since no exact algorithm is available to solve the weapon target assignment problems, it is not known how accurate are the solutions obtained by these heuristic algorithms.

In this chapter, we propose several exact and heuristic algorithms to solve the WTA problem. Our branch and bound algorithms are the first implicit enumeration algorithms that can solve moderate size instances of the WTA problem optimally. We also propose heuristic algorithms, which generate almost optimal solutions within a few seconds. This chapter makes the following contributions:

- We formulate the WTA problem as an integer linear programming problem, that is, as a generalized integer network flow problem on an appropriately defined network. The linear programming relaxation of this formulation gives a lower bound on the optimal solution of the WTA problem. We describe this formulation in Section 3.2.1.

- We propose a minimum cost flow formulation that yields a different lower bound on the optimal solution of the WTA problem. This lower bound is, in general, not as tight as the bound obtained by the linear programming formulation described above but it can be obtained in much less computational time. We describe this formulation in Section 3.2.2.

- We propose a third lower bounding scheme in Section 3.2.3, which is based on simple combinatorial arguments and uses a greedy approach to obtain a lower bound.

- We develop branch and bound algorithms to solve the WTA problem employing each of the three bounds described above. These algorithms are described in Section 3.3.

- We propose a very large-scale neighborhood (VLSN) search algorithm to solve the WTA problem. The VLSN search algorithm is based on formulating the WTA problem as a partition problem. The VLSN search starts with a feasible solution of the WTA problem and performs a sequence of *"cyclic and path exchanges"* to improve the solution. We describe in Section 3.4 a heuristic method that obtains an excellent feasible solution of the WTA problem by solving a sequence of minimum cost flow problems and then uses a VLSN search algorithm to iteratively improve this solution.

- We perform extensive computational investigations of our algorithms and report these results in Section 3.5. Our algorithms solve moderately large size instances (up to 80 weapons and 80 targets) of the WTA problem optimally and obtain almost optimal solutions of fairly large instances (up to 200 weapons and 200 targets) within a few seconds.

3.2 Lower-bounding Schemes

In this section, we describe four lower bounding schemes for the WTA problem, using linear programming, integer programming, minimum cost flow problem and a combinatorial method. These four approaches produce lower bounds with different values and have different running times.

3.2.1 A Lower Bounding Scheme using an Integer Generalized Network Flow Formulation

In this section, we formulate the WTA problem as an integer-programming problem with a convex objective function value. This formulation is based on a result reported by Manne [1958] who attributed it to Dantzig (personal communications).

In formulation 3.1, let $s_j = \prod_{i=1}^{m} q_{ij}^{x_{ij}}$. Taking logarithms on both sides, we obtain, $log(s_j) = \sum_{i=1}^{m} x_{ij} log(q_{ij})$ or $-log(s_j) = \sum_{i=1}^{m} x_{ij} \left(-log(q_{ij})\right)$. Let $y_j = -log(s_j)$ and $d_{ij} = -log(q_{ij})$. Observe that since $0 \le q_{ij} \le 1$, we have $d_{ij} \ge 0$. Then $y_j = \sum_{i=1}^{m} d_{ij} x_{ij}$. Also observe

that $\prod_{i=1}^{m} q_{ij}^{x_{ij}} = 2^{-y_j}$. By introducing the terms d_{ij} and y_j in formulation 3.1, we get the following formulation:

Minimize $\sum_{j=1}^{n} V_j 2^{-y_j}$ 3.2a

Subject to

$\sum_{j=1}^{n} x_{ij} \leq W_i$ for all $i = 1, 2, ..., m$, 3.2b

$\sum_{i=1}^{m} d_{ij} x_{ij} = y_j$ for all $j = 1, 2, ..., n$, 3.2c

$x_{ij} \geq 0$ and integer for all $i = 1,..., m$ and for all $j = 1,..., n$, 3.2d

$y_j \geq 0$ for all $j = 1, 2, ..., n$. 3.2e

Observe that formulation 3.2 is an integer-programming problem with separable convex objective function. This integer program can also be viewed as an integer generalized network flow problem with convex flow costs. Generalized network flow problems are flow problems where flow entering an arc may be different than the flow leaving the arc (see, for example, Ahuja, Magnanti, and Orlin [1993]). In a generalized network flow problem, each arc (i,j) has an associated multiplier γ_{ij} and the flow x_{ij} becomes $\gamma_{ij}x_{ij}$ as it travels from node i to node j. The formulation 3.2 is a generalized network flow problem on the network shown in the Figure 3.1. We give next some explanations of this formulation.

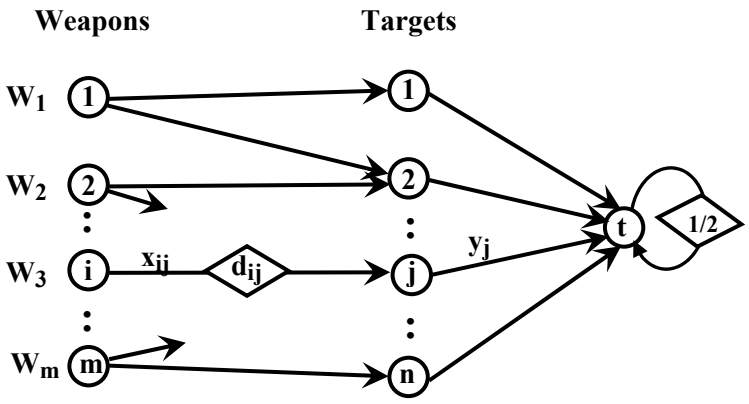


Figure 3.1 Formulating the WTA problem as Integer generalized network flow problem

The network contains m weapon nodes, one node corresponding to each weapon type. The supply at node i is equal to the number of weapons available, W_i for the weapon type i. The network contains n target nodes, one node corresponding to each target. Further, there is one sink node t whose demand equals the sum of all supplies. The supplies/demands of target nodes are zero. We now describe the arcs in the network. The network contains an arc connecting each weapon node to each target node. The flows on these arcs are given by x_{ij}, representing the number of weapons of type i assigned to the target j. The multipliers for these arcs are d_{ij}'s. Since there is no cost coefficient for x_{ij}'s in the objective function, the cost of flow on these arcs is zero. The network contains an arc from each of the target nodes to the sink node t. The flow on arc (j, t) is given by y_j and the cost of flow on this arc is $V_j 2^{-y_j}$. Finally, there is a loop arc (t, t) incident on node t with multiplier ½. An appropriate flow on this arc is sent so as to satisfy the mass balance constraints at node t.

In formulation 3.2, the cost of the flow in the network equals the objective function 3.2a; the mass balance constraints of weapon nodes are equivalent to the constraint 3.2b;

and mass balance constraints of target nodes are equivalent to the constraint 3.2c. It follows that an optimal solution of the above generalized network flow problem will be an optimal solution of the WTA problem.

The generalized network flow formulation 3.2 is substantially more difficult than the standard generalized network flow problem (see Ahuja et al. [1993]) since the flow values x_{ij}'s are required to be integer numbers (instead of real numbers) and the costs of flows on some arcs is a convex function (instead of a linear function). We will approximate each convex function by a piecewise linear convex function and relax the integer flows by real-valued flows so that the optimal solution of the modified formulation gives a lower bound on the optimal solution of the generalized formulation 3.2.

We consider the cost function $V_j 2^{-y_j}$ at values y_j that are integer multiples of a parameter p > 0, and draw tangents of $V_j 2^{-y_j}$ at these values. Let $F_j(p, y_j)$ denote the upper envelope of these tangents. It is easy to see that the function $F_j(p, y_j)$ approximates $V_j 2^{-y_j}$ from below and for every value of y_j provides a lower bound on $V_j 2^{-y_j}$. Figure 3.2 shows an illustration of this approximation.

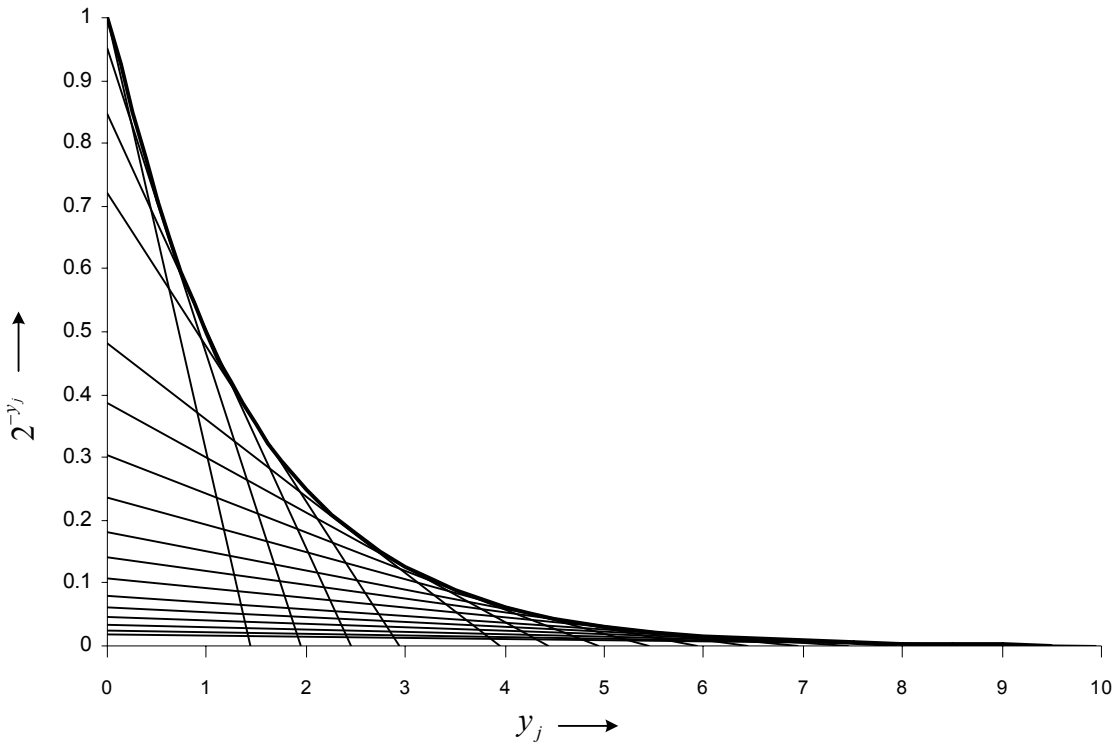


Figure 3.2 Approximating a convex function by a lower envelope of linear segments.

Thus, in formulation 3.2 if we replace the objective function 3.2a by the following objective function:

$$\sum_{j=1}^{n} F_j(p, y_j), \qquad 3.2a'$$

we obtain a lower bound on the optimal objective function of 3.2a. Using this modified formulation, we can derive lower bounds in two ways:

LP based lower bounding scheme: Observe that the preceding formulation is still an integer programming problem because are flows x_{ij}'s are required to be integer valued. By relaxing the integrality of the x_{ij}'s, we obtain a mathematical programming problem with linear constraints and piecewise linear convex objective functions. It is well-known (see, Murty [1976]) that linear programs with piecewise linear convex functions can be

transformed to linear programs by introducing a variable for every linear segment. We can solve this linear programming problem to obtain a lower bound for the WTA problem. Our computational results indicate the lower bounds generated by this scheme are not very tight.

MIP based lower bounding scheme: In this scheme, we do not relax the integrality of the x_{ij}'s, which keeps the formulation to be an integer programming formulation. We, however, transform the piecewise linear convex functions to linear cost functions by introducing a variable for every linear segment. We then use cutting plane methods to obtain a lower bound on the optimal objective function value. We have used the built-in routines in the software CPLEX 8.0 to generate Gomory and mixed integer rounding cuts to generate fairly tight lower bounds for the WTA problem.

We summarize the discussion in this section as follows:

Theorem 1. *Both the LP and MIP based lower bounding schemes give a lower bound on the optimal objective function value for the WTA problem.*

3.2.2 A Minimum Cost Flow Based Lower Bounding Scheme

The objective function of the WTA problem can also be interpreted as maximizing the expected damage to the targets. In this section, we develop an upper bound on the expected damage to the targets. Subtracting this upper bound on the expected damage from the total value of the targets (that is, $\sum_{j=1}^{n} V_j$) will give us a lower bound on the minimum survival value. We will formulate the problem of maximizing the damage to targets as a maximum cost flow problem. We show the underlying network *G* for the maximum cost flow formulation in Figure 3.3.

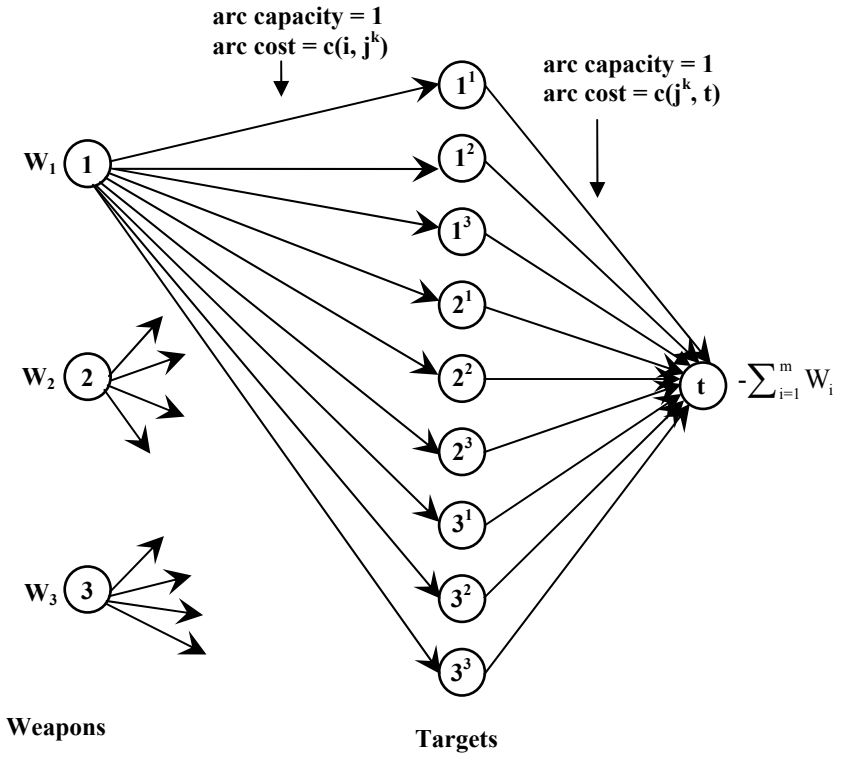


Figure 3.3 A network flow formulation of the WTA problem

This network has three layers of nodes. The first layer contains a supply node i for every weapon type i with supply equal to W_i. We denote these supply nodes by the set N_1. The second layer of nodes, denoted by the set N_2, contains nodes corresponding to targets, but each target j is represented by several nodes j^1, j^2,…, j^k, where k is the maximum number of weapons that can be assigned to target j. A node j^p represents the p^{th} weapon striking the target j. For example, the node labeled 3^1 represents the event of the first weapon being assigned to target 3, the node labeled 3^2 represents the event of the second weapon being assigned to target 3, and so on. All nodes in the second layer have

zero supplies/demands. Finally, the third layer contains a singleton node t with demand equal to $\sum_{i=1}^{m} W_i$.

We now describe the arcs in this network. The network contains an arc (i, j^k) for each node $i \in N^1$ and each node $j^k \in N_2$; this arc represents the assignment of a weapon of type i to target j as the k^{th} weapon. This arc has a unit capacity. This network also contains an arc (j^k, t) with unit capacity for each node $j^k \in N^2$.

We call a flow x in this network a *"contiguous flow"* if it satisfies the property that if $x(i,j^k) = 1$, then $x(i,j^l) = 1$ for all l=1,2,…,k-1. In other words, the contiguous flow implies that a weapon i is assigned to target j as the k^{th} weapon provided that (k-1) weapons have already been assigned to it. The following result directly follows from the manner we have constructed the network G:

Observation 1. *There is one-to-one correspondence between feasible solutions of the WTA problem and contiguous flows in G.*

While there is a one-to-one correspondence between feasible solutions, it is not a cost preserving correspondence if we require costs to be linear. We instead provide linear costs that will overestimate the true non-linear costs. We define our approximate costs next.

The arc (i, j^k) represents the assignment of a weapon of type i to target j as the k^{th} weapon. If k = 1, then the cost of this arc is the damage caused to the target:

$$c(i, j^1) = V_j(1-q_{ij}) \qquad 3.3$$

which is the difference between the survival value of the target before strike (V_j) and the survival value of the target after strike ($V_j q_{ij}$). Next consider the cost $c(i, j^2)$ of the arc (i, j^2) which denotes the change in the survival value of target j when weapon i is

assigned to it as the second weapon. To determine this, we need to know the survival value of target j before weapon i is assigned to it. But this cost depends upon which weapon was assigned to it as the first weapon. The first weapon striking target j can be of any weapon type 1,2,…,m and we do not know its type a priori. Therefore, we cannot determine the cost of the arc (i, j^2). However, we can determine an upper bound on the cost of the arc (i, j^2). We will next derive the expression for the cost of the arc (i, j^k), which as a special case includes (i, j^2).

Suppose that the first (k-1) weapons assigned to target j are of weapon types i_1, i_2, ..., i_{k-1}, and suppose that the type of the k^{th} assigned weapon is of type i. Then, the survival value of target j after the first (k-1) weapons is $V_j q_{i_1 j} q_{i_2 j} ... q_{i_{k-1} j}$ and the survival value of the target j after k weapons is $V_j q_{i_1 j} q_{i_2 j} ... q_{i_{k-1} j} q_{ij}$. Hence, the cost of the arc (i, j^k) is the difference between the two terms, which is

$$c(i, j^k) = V_j q_{i_1 j} q_{i_2 j} ... q_{i_{k-1} j} (1 - q_{ij}) . \quad 3.4$$

Let $q_j^{max} = \max\{q_{ij}: 1, 2, ..., m\}$.Then, we can obtain an upper bound on $c(i, j^k)$ by replacing each q_{ij} by q_j^{max}. Hence, if we set

$$c(i, j^k) = V_j (q_j^{\max})^{k-1} (1 - q_{ij}) , \quad 3.5$$

we get an upper bound on the total destruction on assigning weapons to targets. It directly follows from equation 3.5 that

$$c(i, j^1) > c(i, j^2) > . . . > c(i, j^{k-1}) > c(i, j^k), \quad 3.6$$

which implies that the optimal maximum cost flow in the network G will be a contiguous flow. It should be noted here that since this is a maximization problem, we solve it by first multiplying all arc costs by -1 and then using any minimum cost flow algorithm. Let z* represent the upper bound on destruction caused to targets after all the

assignments obtained by solving this maximum cost flow problem. Then, the lower bound on the objective function of formulation 3.1 is $\sum_{j=1}^{n} V_j$ - z*.

We can summarize the preceding discussion as follows:

Theorem 2. *If z* is the optimal objective function value of the maximum cost flow problem in the network G, then* $\sum_{j=1}^{n} V_j$ *- z* is a lower bound for the weapon target assignment problem.*

3.2.3 Maximum Marginal Return Based Lower Bounding Method

In this section, we describe a different relaxation that provides a valid lower bound for the WTA problem. This approach is based on underestimation of the survival of a target when hit by a weapon as we assume that each target is hit by the best of the weapons.

Let q_j^{min} be the survival probability for target j when hit by the weapon with the smallest survival probability, i.e., $q_j^{min} = \min\{q_{ij}: i = 1, 2, \ldots, m\}$. Replacing the term q_{ij} in formulation 3.1 by q_j^{min}, we can formulate the WTA problem as follows:

Minimize $\sum_{j=1}^{n} V_j \prod_{i=1}^{m} (q_j^{\min})^{x_{ij}}$ 3.7a

Subject to

$\sum_{j=1}^{n} x_{ij} \le W_i$, for all $i = 1, 2, \ldots, m$, 3.7b

$x_{ij} \ge 0$ and integer for all $i = 1, 2, \ldots, m$ and for all $j = 1, 2, \ldots, n$. 3.7c

Let $x_j = \sum_{i=1}^{m} x_{ij}$, and if we let $g_j(x_j) = V_j (q_j^{\min})^{x_j}$, then we can rewrite (3.7) as:

Minimize $\sum_{j=1}^{n} g_j(x_j)$ 3.8a

Subject to

$\sum_{j=1}^{n} x_{ij} \le W_i$, for all $i = 1, 2, \ldots, m$, 3.8b

$\sum_{i=1}^{n} x_{ij} = x_j$ for all $i = 1, 2, \ldots, m$, 3.8c

$x_{ij} \ge 0$ and integer for all $i = 1, 2, \ldots, m$ and for all $j = 1, 2, \ldots, n$. 3.8d

It is also possible to eliminate the variables x_{ij} entirely. If we let $W = \sum_{i=1}^{m} W_i$ then we can rewrite formulation 3.8 as an equivalent integer program 3.9:

Minimize $\sum_{j=1}^{n} g_j(x_j)$ 3.9a

Subject to

$\sum_{j=1}^{n} x_j \le W$, 3.9b

$x_j \ge 0$ and integer for all $j = 1, 2, \ldots, n$. 3.9c

It is straightforward to transform a solution for 3.9 into one for 3.8 since all weapon types are identical in formulation 3.8.

Observe that the formulations 3.1 and 3.7 have the same constraints; hence, they have the same set of solutions. However, in the formulation 3.7, we have replaced each q_{ij} by $q_j^{min} = \min\{q_{ij}: i = 1, 2, \ldots, m\}$, where $q_j^{min} < q_{ij}$. Noting that the optimal solution value for problems 3.7, 3.8 and 3.9 are all identical, we get the following result:

Theorem 3. *An optimal solution value for 3.9 is a lower bound for the WTA problem.*

Integer program 3.9 is a special case of the knapsack problem in which the separable costs are monotone decreasing and concave. As such it can be solved using the greedy algorithm. In the following, assigned(j) is the number of weapons assigned to

target j, and value(i, j) is the incremental cost of assigning the next weapon to target j.

```
algorithm combinatorial-lower-bounding;
begin
    for j := 1 to n do
    begin
        assigned(j) := 0;
        value(j) := g_j(assigned(j)+1) – g_j(assigned(j));
    end
    for i = 1 to m do
    begin
        find j corresponding to the minimum value(j);
        assigned(j) := assigned(j) + 1;
        value(j) := g_j(assigned(j)+1) – g_j(assigned(j));
    end;
end.
```

Figure 3.4 Combinatorial lower bounding algorithm

This lower bounding scheme is in fact a variant of a popular algorithm to solve the WTA problem, which is known as the *maximum marginal return algorithm*. In this algorithm, we always assign a weapon with maximum improvement in the objective function value. This algorithm is a heuristic algorithm to solve the WTA problem but is known to give an optimal solution if all weapons are identical.

We now analyze the running time of our lower bounding algorithm. If we store value(j) in a Fibonacci heap. We point that after the initialization of the heap with the initial values, we perform W "find-min" operations and W decrease-key steps, for a total running time of O(W) time. In our implementation, we used binary heaps, which run in O(W log W) time but are comparably fast for the problem sizes that we considered.

3.3 Branch and Bound Algorithm

We developed and implemented four branch and bound algorithms based on the four lower bounding schemes described in the previous section. A branch and bound algorithm is characterized by the branching, lower bounding and search strategies. We now describe these strategies for our approaches.

Branching strategy: To keep the memory requirement low, the only information we store at any node is which variable we branch on at that node; and the lower and upper bounds at the node. To recover the partial solution associated with a node of the branch and bound tree, we trace back to the root of the tree. The branching strategy we have used in our implementation is based on the maximum-marginal return. For each node of the branch and bound tree, we find the weapon-target combination which gives best improvement and set the corresponding variable as the one to be branched on next. Ties are broken arbitrarily.

Lower bounding strategy: We used the three lower bounding strategies described in Section 3.2. We provide a comparative analysis of these bounding schemes in Section 3.5.

Search strategy: We implemented both the breadth-first and depth-first search strategies. We found that for smaller size problems (i.e., up to 10 weapons and 10 targets), breadth-first strategy gave overall better results; but for larger problems, depth-first search had a superior performance. We report the results for the depth-first search in section 3.5.

3.4 A Very Large-Scale Neighborhood Search Algorithm

In the previous two sections, we described branch and bound algorithms for the WTA problem. These algorithms are the first exact algorithms that can solve moderate size instances of the WTA problem in reasonable time. Nevertheless, there is still a need for heuristic algorithms, which can solve large-scale instances of the WTA problems. In this section, we describe a neighborhood search algorithm for the WTA problem, which has exhibited excellent computational results. This algorithm is an application of very large-scale neighborhood (VLSN) search to the WTA problem. A VLSN search algorithm is a neighborhood search algorithm where the size of the neighborhood is very large and we use some implicit enumeration algorithm to identify an improved neighbor. We refer the reader to the paper by Ahuja et al. [2002a] for an overview of VLSN search algorithms.

A neighborhood search algorithm starts with a feasible solution of the optimization problem and successively improves it by replacing it by an improved neighbor until it obtains a locally optimal solution. The quality of the locally optimal solution depends both upon the quality of the starting feasible solution and the structure of the neighborhood, that is how we define the neighborhood of a given solution. We next describe the method we used to construct the starting feasible solution followed by our neighborhood structure.

3.4.1 A Minimum Cost Flow formulation based Construction Heuristic

We developed a construction heuristic, which solves a sequence of minimum cost flow problems to obtain an excellent solution of the WTA problem. This heuristic uses the minimum cost flow formulation shown in Figure 3.3, which we used to determine a lower bound on the optimal solution of the WTA problem. Recall that in this formulation,

we define the arc costs (i, j^1), (i, j^2), …, (i, j^k), which, respectively, denote the cost of assigning the first, second and k^{th} weapon of type i to target j. Also recall that only the cost of the arc (i, j^1) was computed correctly, and for the other arcs, we used a lower bound on the cost. We call the arcs whose costs are computed correctly as *exact-cost arcs*, and the rest of the arcs as *approximate-cost arcs*.

This heuristic works as follows. We first solve the minimum cost flow problem with respect to the arc costs as defined earlier. In the optimal solution of this problem, exact-cost arcs as well as approximate-cost arcs may carry positive flow. We next fix the part of the weapon target assignment corresponding to the flow on the exact-cost arcs and remove those arcs from the network. In other words, we construct a partial solution for weapon-target assignment by assigning weapons only for exact-cost arcs. After fixing this partial assignment, we again compute the cost of each arc. Some of the earlier approximate-cost arcs will now become exact-cost arcs. For example, if we set the flow on arc (i, j^1) equal to 1, we know that that weapon i is the first weapon striking target j, and hence we need to update the costs of the arcs (l, j^k) for all $l = 1, 2, \ldots, m$ and for all $k \geq 2$. Also observe that the arcs (l, j^2) for all $l = 1, 2, \ldots, m$ now become exact cost arcs. We next solve another minimum cost flow problem and again fix the flow on the exact-cost arcs. We re-compute arc costs, make some additional arcs exact-cost, and solve another minimum flow problem. We repeat this process until all weapons are assigned to the targets.

We tried another modification in the minimum cost flow formulation, which gave better computational results. The formulation we described determines the costs of approximate-cost arcs assuming that the worst weapons (with the largest survival

probabilities) are assigned to targets. However, we observed that in any near-optimal solution, the best weapons are assigned to the targets. Keeping this observation in mind, we determine the costs of valid arcs assuming that the best weapons (with the smallest survival probabilities) are assigned to targets. Hence, the cost of the arc (i, j^k), which is $c(i, j^k) = V_j q_{i_1 j} q_{i_2 j} \ldots q_{i_{k-1} j}(1 - q_{ij})$ is approximated by $c(i, j^k) = V_j [q_{min}(j)]^{k-1}(1-q_{ij})$. Our experimental investigation shows that this formulation generates better solutions compared to the previous formulation. We present computational results of this formulation in Section 3.5.

3.4.2 The VLSN Neighborhood Structure

The WTA problem can be conceived of as a partition problem defined as follows. Let $\boldsymbol{S} = \{a_1, a_2, a_3, \ldots, a_n\}$ be a set of n elements. The partition problem is to partition the set $\boldsymbol{S}$ into the subsets $S_1, S_2, S_3, \ldots, S_K$ such that the cost of the partition is minimum, where the cost of the partition is the sum of the cost of each part. The WTA problem is a special case of the partition problem where the set of all weapons is partitioned into n subsets $S_1, S_2, \ldots, S_n$, and subset j is assigned to target j, $1 \le j \le n$. Thompson and Orlin [1989] and Thompson and Psaraftis [1993] proposed a VLSN search approach for partitioning problems which proceeds by performing *cyclic exchanges*. Ahuja et al. [2001a, 2003a] proposed further refinements of this approach and applied it to the capacitated minimum spanning tree problem. We will present a brief overview of this approach when applied to the WTA problem.

Let $\boldsymbol{S} = (S_1, S_2, \ldots, S_n)$ denote a feasible solution of the WTA problem where the subset S_j, $1 \le j \le n$, denotes the set of weapons assigned to target j. Our neighborhood search algorithm defines neighbors of the solution $\boldsymbol{S}$ as those solutions that can be

obtained from $\boldsymbol{S}$ by performing *multi-exchanges*. A *cyclic multi-exchange* is defined by a sequence of weapons i_1- i_2- i_3-…- i_r- i_1 where the weapons i_1, i_2, i_3, …, i_r belong to different subsets S_j's. Let $t(i_1)$, $t(i_2)$, $t(i_3)$, …, $t(i_r)$, respectively, denote the targets to which weapons i_1, i_2, i_3,…, i_r, are assigned. The *cyclic multi-exchange* i_1- i_2- i_3-…- i_r- i_1 represents that weapon i_1 is reassigned from target $t(i_1)$ to target $t(i_2)$, weapon i_2 is reassigned from target $t(i_2)$ to target $t(i_3)$, and so on, and finally weapon t_r is reassigned from target $t(i_r)$ to target $t(i_1)$. We can similarly define a *path multi-exchange* by a sequence of weapons i_1- i_2- i_3-…- i_r which differs from the *cyclic multi-exchange* in the sense that the last weapon i_r is not reassigned and remains assigned to target $t(i_r)$.

The number of neighbors in the multi-exchange neighborhood is too large to be enumerated explicitly. However, using the concept of *improvement graph*, a profitable multi-exchange can be identified using network algorithms. The improvement graph $G(\boldsymbol{S})$ for a given feasible solution $\boldsymbol{S}$ of the WTA problem contains a node r corresponding to each weapon r and contains an arc (r, l) between every pair of nodes r and l with $t(r) \neq t(l)$. The arc (r, l) signifies the fact that weapon r is reassigned to target (say j) to which weapon l is currently assigned and weapon l is unassigned from its current target; the cost of this arc, c_{rl}, is set equal to the change in the survival value of the target. Let V_j' denote the survival value of the target j in the current solution. Then, the cost of the arc (r, l) is $c_{rl} = V_j'\left((q_{rj} / q_{lj}) - 1\right)$. We say that a directed cycle $W = i_1$- i_2- i_3-…- i_k- i_1 in $G(\boldsymbol{S})$ is *subset-disjoint* if each of the weapons i_1, i_2, i_3, …, i_k is assigned to a different target. Thompson and Orlin [1989] showed the following result:

Lemma 1. *There is a one-to-one correspondence between multi-exchanges with respect to* $\boldsymbol{S}$ *and directed subset-disjoint cycles in* $G(\boldsymbol{S})$ *and both have the same cost.*

This lemma allows us to solve the WTA problem using the following neighborhood search algorithm:

```
algorithm WTA-VLSN search;
begin
        obtain a feasible solution S of the WTA problem;
        construct the improvement graph G(S);
        while G(S) contains a negative cost subset-disjoint cycle do
        begin
                obtain a negative cost subset-disjoint cycle W in G(S);
                perform the multi-exchange corresponding to W;
                update S and G(S);
        end;
end;
```

Figure 3.5 The VLSN search algorithm for the WTA problem.

We now give some details of the VLSN search algorithm. We obtain the starting feasible solution $\boldsymbol{S}$ by using the minimum cost flow based heuristic described in Section 3.4.1. The improvement graph $G(\boldsymbol{S})$ contains W nodes and $O(W^2)$ arcs and the cost of all arcs can be computed in $O(W^2)$ time. We use a dynamic programming based algorithm (as described by Ahuja et al. [2003]) to obtain subset-disjoint cycles. This algorithm first looks for profitable two-exchanges involving two targets only; if no profitable two-exchange is found, it looks for profitable three-exchanges involving three targets; and so on. The algorithm either finds a profitable multi-exchange or terminates when it is unable to find a multi-exchange involving k targets (we set $k = 8$). In the former case, we improve the current solution, and in the latter case we declare the current solution to be locally optimal and stop. The running time of the dynamic programming algorithm is $O(W^2\ 2^k)$ per iteration, and is typically much faster since most cyclic exchanges found by the algorithm are swaps.

3.5 Computational Results

We implemented each of the algorithm described in the previous section and extensively tested them. We tested our algorithms on randomly generated instances as data for the real-life instances is classified. We generated the data in the following manner. We generated the target survival values V_j's as uniformly distributed random numbers in the range 25-100. We generated the kill probabilities for weapons engaging with the targets as uniformly distributed random numbers in the range 0.60-0.90. We performed all our tests on a 2.8 GHz Pentium 4 processor computer with 1 GB RAM PC. In this section, we present the results of these investigations.

3.5.1 Comparison of the Lower Bounding Schemes

In our first investigation, we compared the tightness of the lower bounds generated by the lower bounding algorithms developed by us. We tested the three lower bounding schemes described in Section 3.2: (i) LP based lower bounding scheme; (ii) MIP based lower bounding scheme; (iii) the minimum cost based lower bounding scheme; and (iv) the maximum marginal return based lower bounding scheme. We tested an additional lower bounding scheme which is a variant of the LP based scheme.

Table 3.1 gives the computational results of these four lower bounding schemes. For each of these schemes, the first column gives the % gap from the optimal objective function value and the second column gives the time taken to obtain bound. The following observations can be derived from this table: (i) the MIP lower bounding scheme gives the tightest lower bounds but also takes the maximum computational time; (ii) the minimum cost flow based bounding scheme gives fairly tight lower bounds when the number of weapons is less than or equal to the number of targets; and (iii) the

maximum marginal return algorithm takes the least amount of time to obtain lower bounds.

Table 3.1 Comparison of four lower bounding schemes.

# of Weapons	# of Targets	LP Scheme		MIP Scheme		Min Cost Flow Scheme		Max Marginal Return Scheme	
		% Gap	Time (in secs)	% Gap	Time (in secs)	% Gap	Time (in secs)	% Gap	Time (in secs)
5	5	8.03	0.015	0.21	0.016	1.66	<0.001	10.61	<0.001
10	10	3.63	0.015	0.12	0.031	0.00	<0.001	11.01	<0.001
10	20	19.70	0.015	0.04	0.062	0.00	<0.001	1.45	<0.001
20	10	11.88	0.015	0.53	0.156	21.32	<0.001	19.00	<0.001
20	20	7.28	0.031	0.25	0.109	1.32	<0.001	6.40	<0.001
20	40	23.35	0.046	0.04	0.296	0.00	<0.001	1.57	<0.001
40	10	14.79	0.015	2.12	0.609	42.41	<0.001	46.89	<0.001
40	20	7.06	0.031	0.45	0.359	25.52	0.015	13.53	<0.001
40	40	6.83	0.078	0.11	0.703	1.63	0.015	3.05	<0.001
40	80	21.69	0.14	0.03	1.812	0.00	0.046	0.88	<0.001

3.5.2 Comparison of Branch and Bound Algorithms

We developed branch and bound algorithms using the preceding lower bounding schemes. Table 3.2 gives the results of these algorithms. The branch and bound algorithm using the LP based lower bounding scheme did not perform well at all and we do not present its results. We replaced this algorithm by another algorithm, which we call the *hybrid algorithm*. The hybrid algorithm computes lower bounds using both the minimum cost flow based and the maximum marginal return based lower bounding schemes and uses the better of these two bounds. We find that the branch and bound algorithm using the MIP based lower bounding gives the most consistent results and able to solve the largest size problems (containing 80 weapons and 80 targets). We also find that the hybrid algorithm also gives excellent results for those instances where the number of weapons is less than or equal to the number of targets.

Table 3.2 Comparison of branch and bound algorithms.

# of Weapons	# of Targets	MIP Based B&B Algorithm		Min Cost Flow Based B&B Algorithm		Maximum Marginal Return Based B&B Algorithm		Hybrid Algorithm	
		Nodes Visited	Time (in secs)	Nodes Visited	Time (in secs)	Nodes Visited	Time (in secs)	Nodes Visited	Time (in secs)
5	5	15	0.14	11	<0.001	23	<0.001	11	<0.001
10	10	29	0.56	1	<0.001	181	<0.001	1	<0.001
10	20	23	0.83	1	<0.001	83	<0.001	1	0.015
20	10	101	7.27	-	-	2,8611	1.34	20,251	2.52
20	20	109	6.56	2,383	4.39	15936	0.94	1,705	2.50
20	40	105	16.58	1	<0.001	111,603	10.14	1	0.015
40	10	1,285	327.27	-	-	-	-	-	-
40	20	205	35.19	-	-	$\sim 10^8$	13,651.9	$\sim 10^7$	25,868.9
40	40	211	50.96	$\sim 10^6$	10,583.62	$\sim 10^6$	943.03	38,3275	1,891.83
40	80	385	235.41	1	0.031	-	-	1	0.031
80	40	117,227	43,079.55	-	-	-	-	-	-
80	80	44905	58,477.31	-	-	-	-	-	-
80	160	1055	3,670.49	1	0.062	-	-	1	0.062

3.5.3 Performance of the VLSN Search Algorithm

We now present computational results of the minimum cost flow based construction heuristic and the VLSN search algorithm. Table 3.3 gives the objective function values of the solutions obtained by the construction heuristic and the improved values when VLSN search algorithm is applied to these solutions. We observe that the construction heuristic obtained optimal solutions for over 50% of the instances and for the remaining instances the VLSN search algorithm converted them into optimal or almost optimal solutions. The computational times taken by these algorithms are also very small and even fairly large instances are solved within 3 seconds.

Table 3.3 Results of the construction heuristic and the VLSN search algorithm.

# of Weapons	# of Targets	Construction Heuristic		VLSN Algorithm	
		Optimality Gap	Time (in seconds)	Optimality Gap	Time (in seconds)
10	5	0%	<0.001	0%	<0.001
10	10	0%	<0.001	0%	<0.001
10	20	0%	<0.001	0%	<0.001
20	10	0%	<0.001	0%	<0.001
20	20	0%	<0.001	0%	<0.001
20	40	0%	0.015	0%	0.015
20	80	0%	0.015	0%	0.031
40	10	1.79%	0.015	0%	0.031
40	20	0.33%	0.015	0%	0.015
40	40	0%	0.015	0%	0.015
40	80	0%	0.031	0%	0.078
40	120	0%	0.062	0%	0.109
80	20	2.33%	0.109	0%	0.156
80	40	0.10%	0.062	0%	0.109
80	80	0.0003%	0.093	0.0003%	0.156
80	160	0%	0.172	0%	0.219
80	320	0%	0.390	0%	0.625
100	50	0.79%	0.120	0.0015%	0.437
100	100	0.001%	0.187	0.0009%	0.250
100	200	0%	0.375	0%	0.609
200	100	0.01%	0.656	0.0059%	0.828
200	200	0.001%	0.921	0.0008%	1.109
200	400	0%	1.953	0%	2.516

3.6 Conclusions

In this chapter, we consider the weapon target assignment problem, which is considered to be one of the classical operations research problems that has been extensively studied in the literature but still has remained unsolved. Indeed, this problem is considered to be the holy grail of defense-related operations research. Though weapon target assignment problem is a nonlinear integer-programming problem, we use its special structure to develop LP, MIP, network flow, and combinatorial lower bounding

schemes. Using these lower bounding schemes in branch and bound algorithms gives us effective exact algorithms to solve the WTA problem. Our VLSN search algorithm also gives highly impressive results and gives either optimal or almost optimal solutions for all instances it is applied to. To summarize, we can now state that the WTA problem is a well-solved problem and its large-scale instances can also be solved in real-time.

CHAPTER 4
UNIVERSITY COURSE TIMETABLING PROBLEM

4.1 Introduction

Timetabling is an important problem encountered by almost everybody either in terms of using it or making it. One of the major categories of timetabling which is of importance in an academia is university or school timetabling. In practice, the school or university timetable is prepared manually or with the help of simulation software or customized optimization software. Researchers in OR field have proposed number of algorithms for timetabling problem to minimize the conflict and to maximize the preferences in timetable.

A university or school-timetabling problem, in broad sense, can be defined as the problem of finding a sequence of lectures or examinations between teachers and students in a prefixed period of time satisfying a set of constraints of various types. We can categorize timetabling problem in three parts based on use and context, namely,

Course timetabling (in university): Scheduling of courses, i.e., fixing the time and place for lectures, with the objective of optimizing two important goals, (i) minimizing students' interest conflict and (ii) maximizing teachers' preference. Teachers' preference is important as he/she has got dual responsibility of teaching and research and students' interest is important as there are lot of freedom to students in selecting the course. Course timetabling is generally done for a week, which is repeated through out

semester or trimester. The problem may also include assignment of courses to classrooms so that distance covered by students in between the classes can be minimized.

Examination timetabling: Scheduling of examinations over a pre-specified period of time with the objective of eliminating the conflict in students' schedule (i.e. a student cannot be scheduled to take two exams at the same time) over a set of constraints type. Here constraints are mainly in terms of minimum gap between two exams of a student, different requirements for compulsory and elective courses and rooms' availability.

School timetabling: Scheduling of courses, i.e., fixing the place and time for different courses (lectures) with the objective of minimizing teacher' clash, i.e., a teacher cannot be asked to take two lectures at the same time. Although, it looks similar to university course timetabling, the basic difference lies in terms of teachers' assignment and clash of student's interest. The number of lectures taken by a teacher in a school is normally much more than that in university, whereas all students in a class or group follow same schedule and assigned to same room as against there are much more electives and students have to roam in campus to attend different classes.

One can easily notice from above that objectives for timetabling problem are nothing but set of constraints, e.g., we can make the constraints that a teacher should be assigned to teach in most preferred periods and there should not be any conflict in student's interest. Usually, timetabling problems are defined with constraints only, which are further divided into two categories soft and hard. Hard constraints are those, which must be honored by a timetable whereas soft constraints may be violated, which are subsequently made part of the objective with a cost, associated with every violation of soft constraints. For example, in our case we are considering teacher's preference and

student's interest conflict as soft constraints and make it part of objective functions. An ideal timetable should have zero objective function value, i.e., all the constraints are strictly followed.

Here in this chapter, we are restricting our focus to course timetabling in a university and more specifically to the fixing of time for the courses for a week. We are not considering here the assignment of courses to the room, which can be taken separately as a sub-problem with the objective of minimizing the distance traveled between classrooms by a student. Our approach is based on application of a meta-heuristic Very Large Scale Neighborhood Search, which has shown very encouraging results for number of combinatorial optimization problem.

We have structured this chapter in sections with section 4.2 dealing with problem definition, section 4.3 dealing with literature survey, section 4.4 dealing with mathematical formulation, section 4.5 dealing with random instance generation approach, section 4.6 dealing with heuristics for initial solution, section 4.7 dealing with very large scale neighborhood search, section 4.8 dealing with VLSN and tabu search, section 4.9 dealing with implementation and finally section 4.10 dealing with conclusions drawn from the above chapter.

4.2 Problem Definition

As against the available literatures, which are generally specialized to the case specific algorithms for course timetabling problem, in this chapter, our attempt is to develop an algorithm applicable to general cases. The terminology for the course timetabling as used in this chapter are follows:

Period: An interval of time in which a teacher gives lecture to a group of students in a single sitting. Different periods can have different duration and are characterized by starting time. For this chapter we have numbered the periods in increasing order of the days and time for whole week. We have also assumed that number of periods is same everyday, i.e., if there are 5 periods in a day then there will be 25 periods in week.

Slot: Slot is the set of periods, to which a course can be assigned, i.e., if a course is assigned to slot *s* and *s* consist of periods *p1*, *p2* and *p3*, then this course will be taught in periods *p1*, *p2* and *p3* every week. We assume that we know all-possible slots i.e., all possible combinations of periods apriory.

Course: A course is the material, which is taught. We have not categorized the courses here; however, generally this belongs to either compulsory or elective category. For a course we know the teacher and how many times in a week the course is to be taught (i.e., *credit hours*). We also know apriori number of students going to take the course, which is generally derived from past data.

Permissible slots: Set of slots to which a course can be assigned. The number of periods in a permissible slot must be equal to credit hours for course it belongs to. We assume that the permissible slots are given apriori for every course.

Now, we can define the course-timetabling problem for this chapter as: Minimize the sum of the penalty of assigning a teacher (i.e., a courses the teacher is taking) to an undesired slot and penalty for assigning two courses, in both of which a student is interested, to common period; subject to following constraints.

- Each course must be assigned to a slot in the set of permissible slots.
- A teacher cannot teach more than one course in a period.
- The number of courses to be taught in a period must not exceed the available number of rooms in that period.

- All pre-assignments of course to a slot and unavailability of teacher/room in a slot (period) must be honored.

There may be additional constraints like compactness of timetable (i.e., minimum number of periods required), consideration of multiple sections of a course etc., which are not considered here in this chapter.

4.3 Literature Review

For the last three decades researchers have been working on this problem and have produced number of algorithms for the course timetabling problem. The survey of the automated timetabling problem by Schaerf [1999] has very systematically categorized the timetabling problems and has given the mathematical formulation for each one along with a general solution approach. Schaerf has discussed both exact approach as well as heuristics for the problem. Meta-heuristics such as genetic algorithms, tabu search, simulated annealing etc. are successfully applied for timetabling problem. The reduction of timetabling problem to graph coloring problem and application of heuristics for graph coloring problem has been proved very efficient. Another survey by Carter and Laporte [1998] gives excellent reference to different timetabling problems and algorithms available for the same.

The paper by Werra [1997] has represented the problem as a graph coloring problem and has used generic approach to solve the problem. The approach by Stallaert [1997] has used Integer Programming Formulation to solve the timetabling problem for the Anderson School of Management at UCLA. Author has successfully reduced the number of variables by exploiting the features of the problem. Thompson and Downsland [1996] have used a variant of simulated annealing to solve examination timetabling.

Their approach dynamically changes the neighborhood structure and also includes infeasible solution in neighborhood, however, with penalty. The paper by Dimopoulou and Miliotis [2001] have solved the course timetabling and examination-timetabling problem for a specific case and have formulated the problem as integer program.

The work in this chapter is mainly influenced by the application of tabu search for large scale timetabling problem by Hertz [1991] and tabu search approach by Valdes et al. [2002]. Hertz [1991] has also considered the problem of grouping of students in different sections. Author's approach includes three phases, (a) getting initial feasible solution, (b) applying tabu search on timetabling problem and (c) applying tabu search on grouping problem. This paper also explores infeasible solutions in neighborhood structure, however, keeps a separate account of it. Valdes et al. [2002] has solved both course timetabling as well as room assignment for University of Valencia, Spain with the application of tabu search. Distinction of authors' approach lies in considering not only of pairwise exchanges in neighborhood search but that of multi-exchanges as well. Authors also discuss briefly the approach for assigning courses to rooms.

In this chapter we are applying very large scale neighborhood (VLSN) technique, which has shown very good results for capacitated telecommunication network design (Ahuja et al. [2001a]) as well as on other combinatorial optimization problems. VLSN uses very efficiently the concept of improvement graph to find an improving cycle consists of multiple moves.

4.4 Mathematical Formulation

The course-timetabling problem as defined above can be formulated as quadratic integer program as given below.

Parameters

C: set of courses

CT_t: set of courses for teacher t

T: set of teachers

S: set of slots

D: set of periods

d_i: set of slots which contains period i

R: set of room types

CR_r: courses requiring room of type r

m_{rs}: number of rooms of type r available in slot s

S_i: set of permissible slots for course i

PS_t: set of preferred slots for teacher t

α_{ij}: cost of assigning teacher (course) i to slot j, this may be made zero for the preferred slot for a teacher

q_{ij}: number of students taking both the courses i and j

β_{ij}: penalty if courses i and j are assigned to same slots, this may be made very high for the cases when courses i and j are compulsory for a class, i.e., courses i and j cannot be taught in same slot

γ_{ij}: number of periods common in slots i and j

δ_{ij}: 1 if $\gamma_{ij} > 0$, 0 otherwise

F: set of pre-assigned courses

p_i: slot to which course i is pre-assigned

CS_p: set of courses taken by student p

P: set of students

Decision Variable:

x_{is}: $\begin{cases} 1, \text{ if course } i \text{ is assigned to slot } s \in S_i \\ 0, \text{ otherwise} \end{cases}$

Objective:

$$\min \quad \sum_{i \in C} \sum_{s \in S_i} \alpha_{is} x_{is} + \sum_{i \in C} \sum_{\substack{j > i \\ j \in C}} \sum_{s_1 \in S_i} \sum_{s_2 \in S_j} \gamma_{s_1 s_2} q_{ij} \beta_{ij} x_{is_1} x_{js_2} \qquad 4.1a$$

subject to:

Each course i must be assigned to a slot s

$$\sum_{s \in S_i} x_{is} = 1, \ \forall i \in C \qquad 4.1b$$

A teacher cannot be assigned to more than one course at a time

$$x_{is} + \sum_{\substack{j \in CT_t \\ j > i}} \sum_{s' \in S_j} \delta_{ss'} x_{js'} \leq 1, \ \forall i \in CT_t, \ \forall t \in T, \ \forall s \in S_i, \qquad 4.1c$$

Room availability in each slot

$$\sum_{i \in CR_r} \sum_{\substack{s \in S_i \\ s \in d_p}} x_{is} \leq m_{rs}, \ \forall p \in D, \ \forall r \in R \qquad 4.1d$$

Pre-assignment of courses to slots

$$x_{ip_i} = 1, \ \forall i \in F \qquad 4.1e$$

Integrality constraint

$$x_{is} \in \{0,1\}, \ \forall s \in S_i, \ \forall i \in C \qquad 4.1f$$

Although literature suggests many approaches to solve timetabling as integer program, we assume here that when considered in general way, it may not be possible to reduce the number of variables, which is very crucial for the successful implementation

of this approach. In our opinion non-linear terms in objective function and binary variables make the above formulation difficult.

We implemented the integer-programming equivalent with linear objective function value for above formulation by introducing variable y_j for each quadratic term in objective function value and adding following constraints.

$x_{is_1} + x_{js_2} - y_k \leq 1$, for each quadratic term in objective function value for which

$$\gamma_{s_1 s_2} q_{ij} \beta_{ij} > 0 \qquad 4.2a$$

$$0 \leq y_k \leq 1 \qquad 4.2b$$

The experimental results on the test instances show that convergence to optimal solution for bigger size instances are very difficult.

4.5 Random Instance Generation

In literature, applications of algorithms are specific to the problem set up of a university or school and it is very difficult to obtain problem instances to compare the results. Therefore, we have developed here one approach for random instance generation.

For this chapter, we have made following assumptions in problem defined above and before generating instance.

- The durations of all the periods are same; hence a course can be assigned to a slot having number of periods in it equal to credit hours. This can be generalized by making the condition, as total duration of the periods in a slot should be equal to credit hours for the course.
- All rooms are having same capacity and hence a constraint for room availability is limited to total number of rooms available in a period rather than number of rooms available of different capacity.
- The constraint for pre-assignment of courses and unavailability of room is not considered.

In our approach, we take three inputs for instance generation, namely, total number of periods, total number of slots and total number of courses. The number of periods in a week should be multiple of 5 as we have assumed that there are 5 working days in a week and number of periods per day is same. The feasibility of the instance highly depends on the number of courses, which user has to specify judiciously. With these inputs, the algorithm for instance generation works as follows.

1. The number of periods in each slot takes value 2, 3 or 4 randomly. The options are fixed as we have assumed that credit hours for a course can be 2, 3 or 4 only.

2. Assignment of periods to slots has been done as follows:

    - If number of periods is 2 then first period can be anyone but last period of the week, and then, second period can be either in continuation (on same day) with first period with 50 % probability, otherwise must be from other day (days after the day of first period).
    - If number of periods in a slot is 3 then first period can be anyone from the set of periods but those of last day. Second period can be in continuation with first period with 30 % probability. Similarly, if first and second periods are not in continuation than with 30 % probability second and third periods can be in continuation, otherwise belong to different days.
    - If number of periods are 4, then it consists of periods from two days. First two periods are continuous on one day and last two periods are continuous on another day.

3. Each course is assigned credit hours 2, 3 or 4 randomly.

4. Teacher is assigned to the course randomly. The number of courses taught by a teacher is kept random in the range 1 to 2.

5. The number of permissible slots for each course is kept in the range of 25% to 75% of the total number of slots having number of periods equal to credit hour. The set of permissible slot for a course is generated randomly. It can be noted that two courses may have same set of permissible slots.

6. We keep number of rooms equal to 1.5 * (total credit hours/total number of periods). In ideal case the number of rooms should be equal to total credit hours/total number of periods; however, we keep a slack of half of required rooms to take care of different constraints.

7. The penalty for scheduling two courses in same period when a student is interested in both the courses is randomly generated in range (5, 20).

8. The percentage of pairs of courses, which may have common students, is fixed at 10 % (it can be varied by user). We generate such pair of courses randomly. The number of common students in these pair of courses is generated randomly in the range of (0, 40%) of minimum number of students in a course between the two courses.

9. The penalty for assigning a teacher to a slot is kept in the range (10, 50).

In this algorithm number of parameters can be varied based on the specific application. We expect that this procedure of instance generation represent a wide variety of course timetabling setup.

4.6 Building Initial Solution

Getting a feasible solution in case of course timetabling is not a trivial task as in case of many combinatorial optimization problems. We have developed two construction heuristics to obtain initial solution. For course timetabling, construction heuristic has to meet two requirements namely, it should be able to produce feasible solution, and solution should be of good quality. In both the approaches courses are assigned to slots one by one based on greedy criteria to improve the quality of the solution. One case decision of assigning course to a slot is solely based on minimum increase in objective function value, whereas in another approach we consider the gap in increase in objective function value if assigned to best available slot or second best available slot. In every procedure we check, at the time of every assignment, the availability of room in a period and conflict of teacher in terms that a teacher cannot teach two courses at same time.

4.6.1 Greedy Heuristic 1

The procedure for obtaining initial solution, in the case when we consider only the increase in objective function value at each stage can be described as follows.

- Sort the courses in non-decreasing order of number of permissible slots. This ordering gives priority to courses that have less flexibility in assigning to the slots.
- Take courses in above order and calculate the incremental cost in assigning this course to its permissible slots, without violating room availability and teacher's conflict constraint.
- If found such slot(s), assign the course to slot with least incremental cost. Repeat S*tep 2* for all the courses.
- If a course (*c1*) could not get assigned in *Step 2* and *Step 3*, find the slot (*s1*) for this course in its set of permissible slots by shifting course (*c2*) already assigned to s1 to some other feasible slot (*s2*). We explore all such possibilities and assign *c1* to *s1* and *c2* to *s2* with least incremental cost. The incremental cost here is change in cost by assigning *c1* to *s1* (considering *c2* has been removed from *s1*) plus change in cost by assigning *c2* to *s2* minus change in cost by removing *c2* from *s1*.

This procedure does not always guarantee the generation of feasible solution. While applying this heuristic in conjecture with VLSN, we need different initial solutions in different runs for an instance. We obtain this by assigning a course to one among *p* (a parameter) best slots randomly instead of that to the best slot.

4.6.2 Greedy Heuristic 2

As applied in greedy heuristic 1, the selection of a course for next assignment only based on number of available slots may lead to a solution, which is not good in terms of quality. We propose here another approach, which has shown very good result (Kiaer et al. [1992]).

Let the set *S* contains the courses, which are yet to be assigned to a slot. At start it contains all the courses. Let set A_i contains the available slots to which course *i* can be assigned at any stage. It implies, in periods of these slots, there is no conflict of teacher and room is available to teach the course. Initially, set A_i contains all the permissible slots for course *i*.

Algorithm:

1. Create vector ***cost*** and ***fdeg*** for each course in *S*, where *cost(i, j)* represents the change in objective function value if course $i \in S$ is assigned to the slot $j \in A_i$ and *fdeg(i)* is the cardinality of A_i. Initially, *cost(i, j)* is set to the cost , if teacher of course *i* is assigned to slot *j* and *fdeg(j)* is set to the number of permissible slots for course *i*.
2. For each course in S,
   - if *fdeg(i)* = 0, mark the course as unassigned and remove it from set S
   - else if *fdeg(i)* = 1, assign the course *i* to only slot available. Remove *i* from set *S* and update the cost vector for all other courses in *S*.
3. If *fdeg(i)* > 1 for each $i \in S$,
   - calculate *del(i)*, the difference in two lowest *c(i, j)* for each course and select the course *k* for next assignment for which *del(i)* is maximum over all $i \in S$, use lower *fdeg(i)* in case of tie.
   - Assign the slot with minimum *cost(k, j)* to the course *k*. Remove the course *k* from *S* and update the vectors *cost* and *fdeg*.
4. Repeat Step 2 and 3 until $S = \Phi$.

Figure 4.1 Algorithm for assignment of courses using Greedy Heuristic 2

Above algorithm in figure 4.1 assumes that if we will not assign a course with maximum *del(i)* in current stage, then it may result in assigning this course to the slot having high increase in objective function value. For all the courses, which could not be assigned to any slot, we perform backtracking and try to allocate slot to these courses. To randomize the initial solution, we will select randomly one among best 2 or 3 courses, as got in Step 3.

4.7 Very Large Scale Neighborhood Search (VLSN)

We use the partitioning feature of the timetabling problem, which is expected to give a very efficient VLSN approach. In course timetabling problem we can think of slots as different sets and courses assigned to slots as elements. Then the problem becomes the partitioning of all the courses into available slots. Our proposed heuristic for the course-timetabling problem is based on the following structure of neighborhood search.

Cyclic exchange: We call a cycle $c_{(1)} - c_{(2)} - c_{(3)} - \ldots - c_{(k)} - c_{(1)}$ consisting of courses $c_{(1)}, c_{(2)}, c_{(3)}, \ldots, c_{(k)}$, such that course $c_{(1)}$ is assigned to slot of course $c_{(2)}$, course $c_{(2)}$ is assigned to slot of course $c_{(3)}$ and so on and finally $c_{(k)}$ to slot of $c_{(1)}$, as *Cyclic Exchange*. If the change in objective function value is negative due to such exchange, we call it *Negative Cycle*. Please also note that this is a *subset-disjoint cycle* as elements in cycle are assigned to different slots. However, periods in these slots may not be disjoint.

Path exchange: We call a set of courses $c_{(1)}, c_{(2)}, c_{(3)}, \ldots, c_{(k)}$ such that course $c_{(1)}$ can be assigned to slot of course $c_{(2)}$, course $c_{(2)}$ can be assigned to slot of course $c_{(3)}$ and so on, as *path exchange*. It can be noted that in path exchange, cardinality of slot of $c_{(k)}$ increases by one and that of $c_{(1)}$ decreases by one.

Neighborhood function N (x): Set of all feasible solutions that can be reached from *x* (current solution) by making cyclic exchange with maximum cycle length *k*.

Improvement graph G′ (N, A): This is a graph consists of nodes *N,* a set of all courses; arcs *A,* a set of all arcs *(i, j)* such that, (i) course *i* and course *j* are currently assigned to different slots, (ii) assignment of course *i* to the slot of course *j* should not violate any constraint (considering course *j* has been removed from its current slot) and, (iii) the slot of course *j* is in the set of permissible slots for course *i*. One can think of *arc (i, j)* as representing that the course *i* is assigned to current slot of course *j* and course *j* will leave its current slot. We also add dummy nodes and dummy arcs in the Improvement Graph, to consider the possibility of path exchange. The details of this are omitted here, however, one can refer to Ahuja et al.[2001a] for details.

Cost (i, j): This represents the cost of *arc (i, j)* in improvement graph. This can be visualized as the change in objective function if course *i* is assigned to slot of course *j* and

course *j* leaves its current slot. It can be noteed that *cost (i, j)* depends only on the courses assigned to periods in the current slots of courses *i* and *j*.

Above formulation of Improvement Graph and Cyclic exchange is based on the following theorem by Thompson and Orlin [1989].

> *There is one to one corresponding between cyclic exchange with respect to partition S (of courses in slots) and subset disjoint directed cycles in the improvement graph G', and both have the same cost.*

With the above structure, we propose following heuristic for course timetabling problem.

- Start with an initial solution generated with randomized greedy method.
- Construct the Improvement Graph with respect to above initial solution. If the initial solution is infeasible, i.e., some courses are yet to be assigned to a slot, exclude the unassigned courses from Improvement Graph. It implies that we will look for negative cycle consisting of courses, which are already assigned to slots.
- Find subset disjoint negative cycle in above improvement graph of maximum length *k* (a parameter). We apply dynamic programming approach to find such cycle, which enumerate the paths in stages of different length, i.e., it starts with the paths of length 2 in improvement graph, once enumerated all, goes to paths of length 3 and so on. At every stage we check the cycle cost by making cycle by adding arc from last node to first node in path.
- While calculating path cost we require adjustment in it to take care of moves of nodes in the path. Suppose there is a path *i->j->k*, where courses *i* and *j* have common students. Also suppose that there is a common period in current slots of courses *j* and *k*, which is not there for the current slots of courses *i* and *j*. Therefore, students' conflict component of *cost (i, j)* must be zero as there is no common period in current slots of *i* and *j*. However, when *i* will move to the slot of *j* and course *j* will move to that of *k*, then this component will also get value, which is required to be added in path cost at the time of path enumeration. Similarly, conflict in teacher's assignment is also required to be checked.
- If found any negative cost cycle, implement the cycle and update the improvement graph with respect to change in timetable due to implementation of negative cycle. If there are unassigned courses (i.e., current solution is infeasible), check whether an unassigned course can be assigned to a permissible slot. If so, assign the course to a permissible slot resulting in least increase in objective function value. Repeat Step 3.

- Else return the current solution as local optimal solution and stop.

4.8 VLSN with Tabu Search

Our intuition says that due to hard constraints in course timetabling problem, the feasible region is highly scattered and sparse. Therefore, even with very large-scale neighborhood, the local search can converge to a solution far from global optimal solution. To overcome this we also apply *tabu search* (Gover et al. [1997]), a memory based popular meta-heuristic, which can come out a local optimal point. We have used short-term memory to avoid cycling and use long-term memory to diversify the search. Our implementation of tabu search for course timetabling problem can be described as follows.

- **Short term memory:** In tabu search to minimize the cycling of search, short-term memory is kept for the moves made in recent past. For a cycle implemented, we prohibit the algorithm to consider same move for certain number of iterations, called *tabu tenure*. For example, if course *i* is assigned to slot *s* in current iteration, then for a pre-specified number of iterations, we will not assign course *i* to any other slot. We call a move tabu if it results in assigning course *i* to a slot other than s for pre-specified number of iterations. We call a cyclic exchange tabu, if any of the moves in cycle is tabu. We assign three different tabu tenures for different tabu moves, which is based on the desirability of the move.

    1. Long tabu tenure for an assignment, which results into most preferred/ second most preferred slot for the teacher. We keep it double of short tabu tenure.
    2. Medium tabu tenure if not the above case and move do not induce new conflict in student's preference. We keep it 1.5 times the short tabu tenure.
    3. Short tabu tenure if assignment results into new conflict in student's preference. We keep it 0.1 times the number of courses.

- **Aspiration Criteria:** In tabu search, if we find a cyclic exchange which produce lucrative solution, we accept it irrespective of tabu status For course timetabling, we have kept 2 such aspiration criteria as indicated below.

    1. If objective function value after implementing the multi-exchange is better than best objective function value achieved so far, accept the multi-exchange irrespective of its tabu status.

2. If a multi-exchange results into reduced number of student's clash than that achieved so far, accept the multi-exchange irrespective of its tabu status.

- **Long Term Memory:** In tabu search, along with searching solution on a pre-specified path, we may need a diversification strategy, so that search can be done in wider solution space. To achieve, we keep long-term memory, which keeps the count of number of iterations, in which a course is assigned to a slot. This type of long-term memory is called *residence frequency*. In our search procedure, if there is no improvement in best solution found in a run for certain number of iterations, we restart the tabu search with new initial solution. In algorithm for getting initial solution, we penalize the assignments, which are having high residence frequency.

- **Stopping criteria:** We stop the tabu search after 10 * number of courses iterations.

4.9 Implementation

We have implemented the procedures for instance generation, initial solution generation and VLSN search algorithm. In VLSN, we have implemented two variants of neighborhood structure. In one, we do not make arc (i, j) if moving of course i to the slot of course j and taking out course j from its current slot creates either teacher's conflict or room unavailability. Here we assume that all other courses will remain to be assigned to their existing slot. In another approach, we do not check the teacher's conflict and room availability while creating an arc; rather, we check these constraints while enumerating cycles. This approach gives larger neighborhood than compared to first variant, as even if an arc is infeasible, the resulting cycle may be feasible, as the courses causing infeasibility might have been moved to some other slot. The neighborhood size is also influenced by the maximum cycle length and number of paths enumerated in a stage. Experimental results show that maximum cycle length beyond 5 does not benefit much to the solution quality, whereas it increases running time considerably. Similarly, a path of cost more than 100 or 10 % of current objective function value does not result into a negative cycle. Therefore, for our implementation we kept maximum cycle length limited

to 5 and enumerate paths with maximum cost min (100, 10 % of current objective function value).

Table 4.1 shows the solution obtained and time taken for implementation of integer programming formulation and VLSN algorithm. We apply 100 runs of VLSN local search on each test instance starting with initial solution obtained by greedy heuristic 2. We apply 5 runs of VLSN tabu search on each test instance starting with the best initial solution obtained in 20 runs of greedy heuristic 2. We keep stopping criteria of maximum (8 * number of courses) iterations in each run and apply diversification strategy if there is no improvement in local best in (3 * number of courses) iterations.

Table 4.1 Performance of VLSN and Tabu search

Instance Statistics			Optimal Solution		VLSN Local (100 runs)			VLSN Tabu (5 runs)		
Nperi-ods	NSlots	nCourses	Obj.	T(Sec)	Best % Dev	Avg % Dev	T(Sec)	Best % Dev	Avg % Dev	T(Sec)
20	20	10	299	0.188	0	2.34	0.375	0	0	0.094
20	30	20	447	0.406	11.63	45.41	0.625	0.45	10.45	0.453
20	30	15	298	0.109	1.34	1.34	0.438	1.34	1.34	0.281
20	20	30	2011	0.547	0	12.53	0.718	0	2.39	0.562
20	30	30	984	0.875	0	20.22	0.718	0	5.69	0.891
30	30	15	300	0.109	0	2.00	0.469	0	0.67	0.297
30	30	25	806	0.313	1.36	12.41	1.157	1.86	8.31	1.047
30	30	20	378	0.234	0	6.35	0.656	0.79	0.79	0.547
40	35	20	557	0.156	0	16.88	0.782	0	4.13	0.641
40	40	30	622	0.578	0	12.54	1.843	0	5.47	1.500
50	60	40	598	1.25	1.67	18.90	4.532	0.17	2.69	4.610
50	50	50	822	1.141	0	5.16	3.782	0.97	2.92	6.652
50	100	100			2036	2299	21.454	1991	2047	43.828
50	100	150			4229	5105	30.609	4164	4485	87.890
50	100	200			8482	10282	60.610	8349	9105	157.73

In table 4.1, the columns *nPer*, *nSlots*, and *nCour* represent number of periods, number of slots and number of courses respectively. We report the best objective function

value in column *Obj. Fn*. and average of final objective function values in each run in column *Avg. Obj*. The total times taken by different approaches are shown in column *T(Sec)*.

From table 4.1 we can deduce following:

- Implementation of integer programming formulation gives optimal solution in very short time for small size test instances. However, it fails to converge to optimal solution for larger test instances.
- VLSN local search has been able to achieve optimal solution in most of the cases; however, the average solution quality by this approach is not very robust.
- VLSN tabu search gives optimal solution in almost all the cases. The average solution quality is also much more superior to that in local search. However, this approach also takes more time as compared to local search.

4.10 Conclusion and Further Scope

In this report we have outlined the general framework of course timetabling problem and has proposed two heuristics for getting initial solution and a heuristics for local search and tabu search using VLSN. We observed that integer programming formulation can be used to obtain optimal solution for smaller test instances; however, as size of the instance increases one should use VLSN with tabu search, as it gives very robust solution in manageable time. Further work is required in setting parameters for random instance generator to make it better representative of the real life problem instances.

CHAPTER 5
BLOCK-TO-TRAIN ASSIGNMENT PROBLEM

5.1 Introduction

Transportation of goods by railroads is an integral part of the US economy. Railroads carry millions of *shipments* annually from their origins to their respective destinations. Planning and executing the transportation of these shipments involves many complex decisions at different levels of operation. In the last few years, a growing body of advances concerning several aspects in rail freight, such as train scheduling, blocking plan, block-to-train assignment, locomotive scheduling, etc., has appeared in the operations research literature (see, for example, Cordeau et al. [1998], Newman et al. [2002], and Kraft [1998]). In this paper, we study a key-planning problem, called the *block-to-train assignment* (BTA) problem, which arises after shipments are classified into blocks and a train schedule has been defined. We first give the reader a brief overview of the railroad blocking problem, the train-scheduling problem, and the block-to-train assignment problem.

The blocking problem: The first and foremost planning problem to be solved in railroad is the blocking problem, which involves the grouping of shipments into blocks. In rail transportation, a shipment may pass through many classification yards on its route from its origin to its destination. To prevent shipments from being reclassified at every yard they pass through, several shipments may be grouped together to form a *block*. After a shipment is placed in a block, it is not reclassified until it reaches the destination of that

block. The blocking problem is to identify this classification plan for all shipments in this network, called a *blocking plan*, so that the total shipment cost is minimal. The total shipment cost is the sum of two cost terms: the cost of classification of shipments and the car miles traveled by as they are carried from their origins to their destinations. Some recent references on blocking problem are: Newton et al. [1998], Barnhart et al. [1998] and Liu [2003].

The train scheduling problem: Given a blocking plan, developing a train schedule (also called train timetable) is perhaps the next most important operational planning task faced by a railroad. The train-scheduling problem is to identify train routes, their frequencies (how often to run in a week), and their timetables so as to minimize the cost of carrying cars from their origins to their destinations. This problem also includes the synchronization of real-time train movement on the links of the physical railway network. The train schedule once developed is repeated every period (mostly weekly) of a given time horizon (may be a quarter year). Some recent references on this problem are due to Farvolden and Powell [1994], Campbell [1996], Kraft [1998] and Barnnlund et al. [1998].

The block-to-train assignment problem: Once the blocking plan and the train schedule have been developed, the next step is to determine which trains should carry which blocks. This problem is known as the block-to-train assignment problem and this paper is concerned with the modeling and developing algorithms to solve this problem. The goal in this problem is to have the block transported at minimum cost and with minimum delay from due dates while meeting the operational constraints. Based on the service level of a block, railroads may impose penalty on the deviation from due date.

Ideally, in an optimal solution a block should take the shortest train route from its origin to its destination. However, there are capacity and operational constraints, which must be satisfied by a solution. These restrictions make the block-to-train problem difficult to be solved to optimally.

Most railroads solve the BTA problem manually. We are aware of only few efforts in literature to solve the BTA problem exclusively. Nozick et al. [1997] deals with the finite horizon, discrete time problem of minimizing the total variable cost of moving cars given a fixed train schedule and satisfying due dates. They have developed a procedure that involves iteratively solving a linear programming relaxation and rounding some of the resulting fractional values until a feasible integral solution is found. Another paper, Kwon et al. [1998] deals with the algorithm to improve a given blocking plan and block-to-train assignment. They formulate the problem as a linear multi-commodity flow problem and column generation technique is used as a solution approach.

There are few papers, which consider the BTA problem indirectly in their problem formulation. Early work in this area goes to Thomat [1971]. It develops a cancellation procedure that gradually replaces direct shipments by a series of intermediate train connections to minimize operation and delay costs. Crainic et al. [1984] has proposed a non-linear, mixed integer, multi-commodity flow model that deals with the interaction between blocking, make-up, and train and traffic routing decisions. Haghani [1989] proposed a formulation and heuristic approach for a combinatorial train routing and make-up and empty car distribution problems. Keaton [1989] proposed a model and a heuristic method based on Lagrangian relaxation for the combined problem of car blocking and train routing and make-up. The subsequent paper by Keaton [1992a] also

considers the constraints for blocking and maximum transient time for each origin-destination pair.

An instance of the BTA problem is defined by blocks that must be delivered; the train schedule and the physical rail system will be used to deliver the shipments. The physical rail system consists of undirected links (one or more railroad tracks) connecting stations (rail yards) and the trains carrying the shipments on this rail system. Based on the tractive power of the locomotives pulling a train and the links on which it runs, there are limitations on the carrying capacity the trains. Hence a solution of the BTA problem must honor these constraints. These constraints may be in the form of maximum number of cars in the train, maximum length of the train, and maximum weight of the train and may differ on different section of the train's route. It can be noted that these constraints are interlinked, i.e., if number of cars in a train will be high then the length of the train will also be high and so the weight of the train. To make the BTA problem somewhat simpler, railroads make some of these constraints as hard constraints while keeping other constraints as soft constraints (that is, we allow some violation but there a penalty for each violation). In our discussions in this paper, we have included one type of constraints in our formulation while assuming that others will be soft constraints and hence can be included in the objective function. Apart from these, there are many other operational constraints, which will be discussed in detail in the appropriate sections.

Two of the inputs for the BTA problem are the train schedule and the blocking plan. Generally a train schedule is for a weekly period and different trains have different running frequency in the schedule. For example, one train runs everyday of the week while another runs only on Monday, Wednesday, and Thursday. Similarly, not all the

blocks are made everyday and they may have different frequency. Therefore, ideally the block-to-train assignment problem should be formulated for a period equal to maximum of the period of train schedule and blocking plan. Obviously, size of the BTA problem depends on this period and the longer the period the larger will be the problem. To keep the size of the problem manageable, we assume that all the trains run every day and similarly all the blocks are made every day. This assumption allows us to define the BTA problem on the trains and blocks, which is repeated every day. It can be noted that this assumption does not restrict the travel time of a train and the total time taken by a block to reach its destination to one day. To solve the problem with longer periods, we can either generalize the approaches developed in this paper or can use heuristics to get the solution of the problem with longer period in the post-processing phase. In the rest of the paper, we refer the daily BTA problem simply as the BTA problem.

In this paper, we give several integer-programming formulations of the BTA problem and develop several algorithms to solve this problem. This paper makes the following contributions.

- We develop a space-time network, which allows us to formulate the BTA problem as a multi-commodity flow problem with additional side constraints.
- We develop two integer-programming formulations of the BTA problem. The first formulation formulates the BTA problem as a node-arc version of multi-commodity flow problem in the space-time network. The second formulation conceives the BTA problem as a path-flow version of the multi-commodity flow problem in the space-time network.
- The integer programming formulations can also be used to solve the BTA problem approximately. Using these formulations, we suggest a Lagrangian relaxation algorithm that can solve the BTA problem to near-optimality within a few seconds.
- We also propose a greedy construction algorithm that proceeds by assigning blocks to trains one by one until all blocks are assigning. By choosing different rules for selecting blocks and assigning them to trains, we can obtain different implementations of this generic approach.

- We perform extensive computational investigations of these algorithms. Our integer programming formulations can solve the BTA problem to optimality within a few minutes of computer time using CPLEX 8.1. The Lagrangian relaxation algorithm can solve the BTA problem to near-optimality within a few seconds. The computational performances of greedy construction algorithms are also very attractive.

To summarize, this paper reports the first serious effort to utilize the state-of-the-art mathematical programming techniques to solve the BTA problem. We demonstrate that this problem can be solved to optimality or near-optimality within reasonable computational times.

5.2 Mathematical Programming Formulations

In this section, we describe the space-time network so that the BTA problem can be formulated as a flow problem in this network. This space-time network is similar to the space-time network developed by Ahuja et al. [2002b] to model the flow of locomotives in a train network while modeling the locomotive scheduling problem.

5.2.1 The Space-Time Network

We define the BTA problem on a space-time network, in which every node is distinctly identified by place, time and train. We denote the space-time network as $G = (N, A)$, where N denotes the node set and A denotes the arc set. We construct the space-time network as follows.

A train runs from its origin to its destination at pre-specified time schedule and stops at many stations on the route to perform different activities, like, maintenance check-up, refueling, dropping or adding shipments, etc. However, a block can be attached to or detached from a train at a station, only if the station is origin of the train, or destination of the train, or a stop where shipments can be added or dropped. We call all

such stations as *valid-stops* for the train and we call the itinerary of a train between two consecutive valid-stops as a *trip*. Hence, the route of a train consists of several trips. In practice, a train route may contain as many as 20 trips. The trip l of a train is represented by a *train-arc* (l', l'') in space-time network. The tail node l' of the arc denotes the event for the departure of train at the origin of the trip and is called the *train-departure node*. The head node l'' denotes the arrival event of the train at the destination of the trip and is called the *train-arrival node*.

To connect two consecutive trips of a train, we create *train-connecting arcs* from the train-arrival node of a train at a station to the train-departure node of the same train at the same station. The time difference between these two nodes represents the stay time of the train at the station. To allow a block to be added or dropped from a train on its route, we introduce *ground nodes* and *connecting arcs*. For each train-arrival node, we create a corresponding *ground-arrival node* with same station, time and train attributes as that of train-arrival node. Similarly, for each train-departure node, we create a *ground-departure node* with the same station, time and train attributes as that of train-departure node. We connect each train-arrival node to the associated ground-arrival node by a directed arc called *arrival-connection arc*. We connect each ground-departure node to the associated train-departure node by a directed arc called *departure-connection arc*. To facilitate the switching of a block from one train to another train at a station, we sort all the ground nodes at each station in the chronological order of their time attribute, and connect each ground node to the next ground node in this order by directed arcs called *ground arcs* (we assume here without any loss of generality that ground nodes at each station have distinct

time attributes). Further, to allow a block to take a train in next period, we connect the latest ground node at a station with the earliest ground node at the same station.

To summarize, the space-time network $G = (N, A)$ has three types of nodes, namely train-arrival nodes, train-departure nodes and ground nodes (comprising of two sub-types: ground-arrival nodes and ground-departure nodes). The arc set A consists of three types of arcs, namely train arcs, connection arcs and ground arcs. The set of connection arcs further consists of three types of arcs: arrival-connection arcs, departure-connection arcs and train-connection arcs. We also associate a cost with each arc, which depends on the block flowing on the arc and is discussed in detail in Section 5.2.2. In the space-time network, there is one incoming arc and at most two outgoing arcs at each train-arrival node, at most two incoming arcs and one outgoing arc at each train-departure node, and at most two incoming arcs and two outgoing arcs at ground nodes. It implies that number of arcs in the space-time network cannot be more than double of the number of nodes, which makes the network G highly sparse. A part of the space-time network for BTA problem is shown in Figure 5.1.

With respect to the space-time network G, we define the BTA problem as follows. We say a ground-departure node as the *origin-node* of a block, if it is the earliest ground-departure node after the release time of the block at the origin station of the block. At the destination station of a block, we say the set of ground-arrival nodes as the *destination-node set* of the block. With these notations, we define the BTA problem as to find a path for each block from its origin-node to any of the node in destination-node set in space-time network G, such that the overall cost is minimum and all the operational requirements are satisfied. We claim that any solution to the above problem will also be

the solution of the BTA problem. To prove this, let p is such a path in the space-time network, which is assigned to a block in the solution. As only train-arcs are having different station attributes at its tail node and head node, the path p must consists of train-arcs along with other arcs. Further, two train-arcs in p must be connected by one or more connection arcs and/or ground arcs, which implies that a block will take a train at the origin station, will go to another station and from there it will either take the same train or will be switched to some other train, and so on, until it reaches the destination station. Therefore, path p in space-time network represents a feasible train route for the block, which proves our claim.

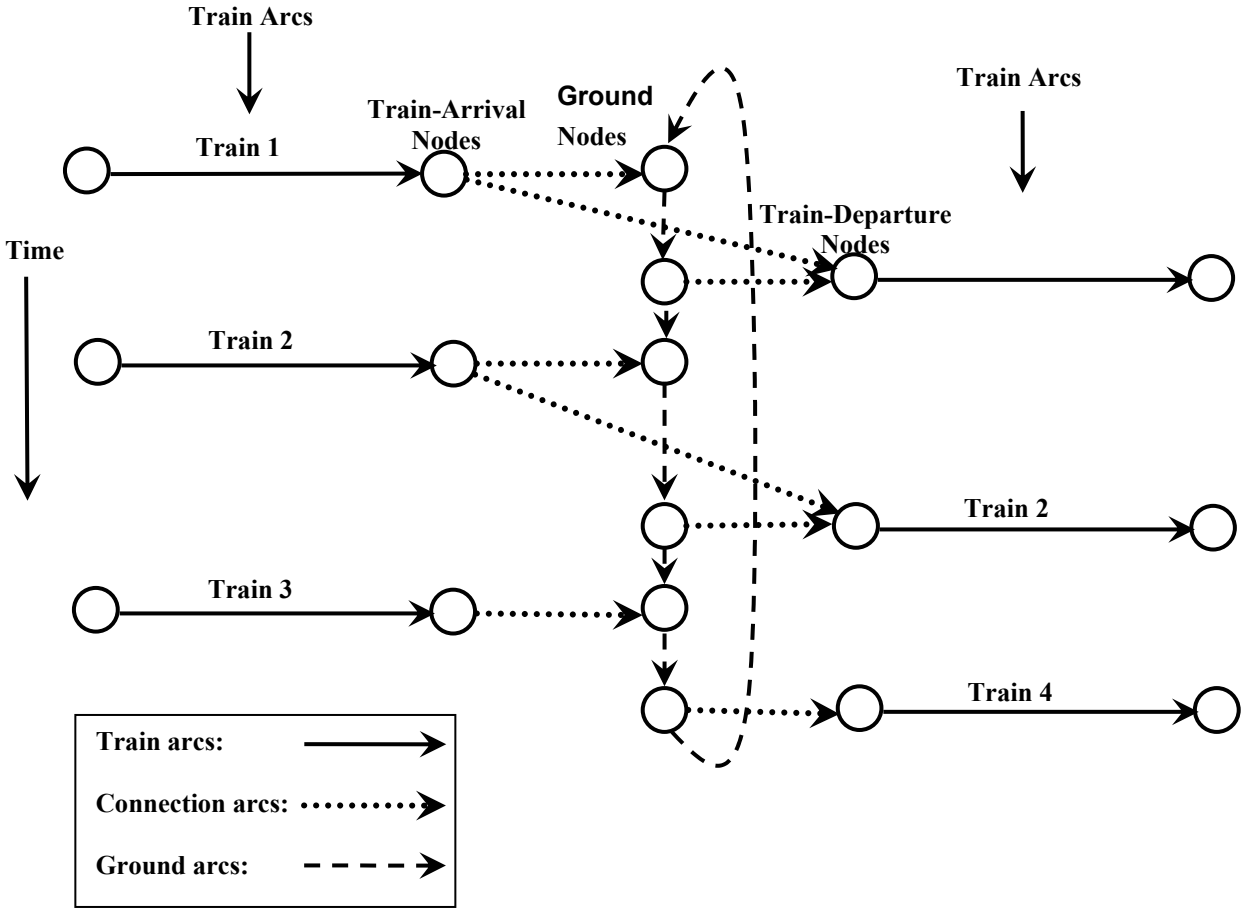


Figure 5.1 A part of the space-time network

We have developed two integer-programming (IP) formulations of the BTA problem in the space-time network G. In one formulation, decision variables are whether a block will flow on an arc or not; whereas in the other formulation the decision variables are whether a block will take a path in space-time network or not. In Sections 5.2.2 and 5.2.3, we discuss these two formulations.

5.2.2 Arc-Based IP Formulation

In Section 5.2.1, we have defined the BTA problem as to find a path in the space-time network, from origin-node of each block to one of the nodes in its destination-node set. This can be formulated as an IP in which a decision has to be made whether an arc will be in the path for a block. We use the following notation in our arc-based IP formulation.

Parameters:

G: space-time network

N: set of nodes in G and we denote a node by i

A: set of arcs in G and we denote an arc by a

B: set of blocks and we denote a block by b

v_b: number of cars in block $b \in B$

O_b: origin-node of block b in G

A_i^+: set of outgoing arcs at node $i \in N$

A_i^-: set of incoming arcs at node $i \in N$

D_b: destination-node set of block b in G

C_a: capacity of arc $a \in A$

δ_a^b: incidence taking value 1 if block $b \in B$ can flow on arc $a \in A$

c_a^b: unit cost of flowing block $b \in B$ on arc $a \in A$

Decision variables:

$$x_a^b = \begin{cases} 1 \text{ if block b} \in \text{B flows on arc an a} \in \text{A} \\ 0, \text{ otherwise} \end{cases}$$

Formulation (5.1)

$$\min \sum_{b \in B} \sum_{a \in A} v_b c_a^b x_a^b \qquad 5.1a$$

subject to

$$\sum_{a \in A_{O_b}^+} \delta_a^b x_a^b = 1 \qquad \forall b \in B \qquad 5.1b$$

$$\sum_{a \in A_i^+} \delta_a^b x_a^b - \sum_{a \in A_i^-} \delta_a^b x_a^b = 0 \qquad \forall b \in B, \forall i \in N, i \neq O_b, i \notin D_b \qquad 5.1c$$

$$\sum_{i \in D_b} \sum_{a \in A_i^-} \delta_a^b x_a^b = -1 \qquad \forall b \in B \qquad 5.1d$$

$$\sum_{b \in B} v_b x_a^b \leq C_a \qquad \forall a \in A \qquad 5.1e$$

$$x_a^b \in \{0,1\} \qquad \forall a \in A, \forall b \in B \qquad 5.1f$$

In formulation 5.1, the constraints 5.1b, 5.1c and 5.1d ensure that there exists a path for each block from its origin-node to a node in its destination-node set. Constraints 5.1e ensure that the total number of cars flowing on an arc is no more than its capacity. If an arc is a train-arc, we set its capacity equal to the capacity of associated train on the segment represented by station attribute of the tail node and the head node of the arc. For all other arcs, we set its capacity equal to infinity. It can be noted that in the absence of constraints 5.1e, the BTA problem reduces to finding a shortest path for each block.

Now, we will describe the objective function 5.1a in formulation 5.1. We have associated a cost with each arc and block combination. This cost represents the cost of flow of a car in a block on an arc. In the case of train-arcs, this cost depends on the car-miles, transit-time and service level of the block. In the case of ground arcs, the cost c_a^b depends on the waiting time (duration of arc *a*) of block *b* at the associated station and the service level of block *b*. We do not associate any cost with any of the arrival-connection arcs and train-connection arcs. However, at all the stations other than the origin station of a block, if the decision variable corresponding to the departure-connection arc takes value 1, it signifies that the block is switching the train and a cost is incurred in detaching the block from a train and in attaching it to other train. Therefore, we associate a cost equal to the train switching cost with all such departure-connection arcs.

In formulation 5.1, we have also associated an incidence scalar δ_a^b with each arc-block combination. This is required to prevent a block being assigned to an arc, which violates any of the operational requirements. Generally, to meet the government regulations or safety requirements (like, width, height or weight limitations, hazardous material limitation, etc.) a block may not be associated to a train or to any train traveling on a particular link. We set the incidence $\delta_a^b = 0$, for all such arcs and blocks combination and set $\delta_a^b = 1$ to all other combinations. Railroads may also set the incidence $\delta_a^b = 0$, if they do not want to assign a block to a trip for historical reasons or obligation to customers.

We will now briefly discuss the size of the problem 5.1 for a typical US railroad application. The number of stations in the physical network of the railroad is around

6,000 and there are around 350 trains running daily. Altogether, these trains are having around 2,000 valid stops, which results into a space-time network with around 7,000 nodes and 10,000 arcs. The railroad makes around 1,200 blocks every day. It implies that there will be around 10,000 × 1,200 = 12 million binary decision variables and around 1,200 × 7,000 + 10,000 ≈ 8.5 million constraints. It is almost impossible to solve a problem of this magnitude with commercially available optimization software.

In addition to the difficulty of this problem due to its magnitude, there are many operational requirements, which are difficult to be included in formulation 5.1. For example, a block is not allowed to travel on more than a certain number of trains from its origin to its destination. It is difficult to include this constraint in the formulation. Similarly, instead of preventing a block from being associated with a trip, railroad may want that a block should not take certain routes, or they may want that a block should take a route one amongst the limited number of routes. It is difficult to include this constraint in the formulation (1). We next describe another IP formulation, which overcomes many of these difficulties.

5.2.3 Path-based IP Formulation

This formulation requires that for each block we have already enumerated a set of feasible train paths and decision to be made is which path to use for each block. We call a path a *valid-path* for a block if it is from the origin-node of the block to a node in the destination-node set of the block in the space-time network and meets all the operational requirements, except the capacity constraint on the train. In other words, a block cannot be assigned to a valid-path only if it violates the capacity constraint on train-arcs in the space-time network. In the rest of the paper, we will interchangeably use valid-paths and

paths. In addition to the notations used before, this formulation uses some more notations. We describe these new notation next followed by the path-based formulation.

Parameters:

P_b: set of feasible paths for block $b \in B$ in G
P: set of all the paths, i.e., $P = \bigcup_{b \in B} P_b$
P'_a: {path p: $p \in P$ and arc $a \in A$ is on p}
c_p: cost of assigning path $p \in P$ to the corresponding block

Decision Variables:

$$x_p = \begin{cases} 1 \text{ if path } p \in P \text{ is assigned to the current block} \\ 0, \text{ otherwise} \end{cases}$$

Formulation (5.2)

$$\min \sum_{b \in B} \sum_{p \in P_b} c_p x_p \qquad \text{5.2a}$$

subject to,

$$\sum_{p \in P_b} x_p = 1 \qquad \forall b \in B \qquad \text{5.2b}$$

$$\sum_{\substack{p \in P'_a \\ b: p \in P_b}} v_b x_p \le C_a \qquad \forall a \in A \qquad \text{5.2c}$$

$$x_p \in \{0,1\} \qquad \forall p \in P_b, \forall b \in B \qquad \text{5.2d}$$

In the above formulation, constraints 5.2b ensure that exactly one path is assigned to each block and constraints 5.2c ensure that the train capacity is not violated. In this formulation, the number of decision variables depends on the possible number of paths for each block. In Section 5.3, we discuss the path enumeration algorithms and for the problem under consideration the number of decision variables can go up to 500,000. As we have constraints for each block and each arc in the space-time network, the number of

constraints can be at most 12,000. It can be observed that the number of constraints in the above formulation is much less than that in formulation 5.1.

An important advantage of the formulation 5.2 over formulation 5.1 is in defining the objective function. One of the important goals for the railroads is to deliver the shipments before due date to the customer. In the path-based formulation, we know that at what time a block is reaching its destination and hence we can define the path penalty proportional to the deviation from the due date. Hence, the cost associated with each path (c_p) takes into account the car-miles, deviation from due date, service level of block and switching cost of blocks. This cannot be done in the arc-based formulation of the BTA problem. Another important advantage of this formulation is that we can eliminate those paths, which do not meet operational constraints by simply not listing them in the set of valid paths. We cannot eliminate such possibilities in formulation 5.1.

However, in order for this formulation to be of reasonable size, we should not allow the set of valid paths to be too large. The total number of train paths for a block in the space-time network can be exponentially large, but we need to identify a small set of valid paths that the block is likely to use. Using some preprocessing, we can easily identify the set of valid paths for each block. This is consistent with the current practice at railroads. Railroads do not allow blocks to be sent along arbitrary paths in the train network, but there are many restrictions that these paths must follow which keeps the number of possible paths manageable. In the rest of this paper, we will use the path-based formulation of the block-to-train assignment problem.

5.3 Path Enumeration Algorithms

Our path-based formulation for the BTA problem requires that we be able to identify the set of valid paths for each block. In this section, we describe an algorithm to determine these sets of valid paths. In our algorithms, we first identify all the distinct origin-nodes (of blocks) and then identify all the feasible paths from each of the distinct origin-nodes to all other nodes in network. Therefore, instead of identifying valid paths for each block separately, we identify valid paths for all the blocks originating at a specific origin-node in one go. To restrict the number of valid paths for a block, we use the following criteria used by the railroads: (i) limit the number of arcs on the valid path to a number less than a specified parameter (say, example, 3); and (ii) limit the length of the valid path to a specific percentage over the shortest path in the train network. These restrictions are motivated by the current practices followed by US railroads.

Our algorithms for path enumeration are the variation of the algorithms to find shortest paths from a source node in a graph (see Ahuja et al. [1993] for detail). The algorithms to find shortest paths from a node, we calculate the cost in reaching other nodes from the source node through all possible paths and pick up the path for which this cost is minimum. In these algorithms, we keep a label with each node, which denotes the cost of some path from the source node to a node and algorithm updates it with lower value whenever a better path is found. In our algorithms, we keep more than one label at a node and every time we find a new path to a node from the source node, we add a new label. A label is uniquely identified by the node with which it is associated and its predecessor label. The attributes of each label and their significance are discussed later.

5.3.1 Path Enumeration Restricted by Number of Trains

Generally, railroads put an upper bound on the number of trains (say *t*) a block can take from its origin to its destination. In our algorithm, we enumerate all the paths and ignore all those paths, which take more than *t* number of trains. Our algorithm works as follows. We start from a distinct origin-node (say, the *source node*) and visit the head nodes of all the arcs emanating from this node, then visit the head nodes of all arcs emanating from the nodes just visited, and so on, until the number of trains visited is more than t. In the above procedure, every time we visit a node, we create a new label at the node; a label signifies that there exists a unique path from the source node to that node. With each label, we keep a list of attributes, such as, the node associated with this label, the distance from the source node to the node, transit time, number of trains visited, predecessor label (the label using which the current label is created), etc. Since a node can be visited multiple times, a node can have multiple labels. To initialize the algorithm, we create a label at the source node and set its predecessor to null signifying that no path exists from the source node to itself. Figure 5.2 describes this path enumeration algorithm.

Figure 5.3 gives an example of the path enumeration algorithm restricted by the number of trains. Suppose that node 1 is the origin-node and we need to enumerate all the paths from node 1. We create a label l_{11} with null predecessor at node 1 and with respect to this label, we examine arcs (1, 2) and (1, 3) and create labels l_{21} and l_{31} respectively. Next, we take label l_{21} and with respect to this label create labels l_{31} and l_{41} at nodes 3 and 4, respectively. Finally, labels l_{42} and l_{43} are created with predecessor labels at node 3. All the labels and their predecessor paths are also shown in Figure 5.3. It can be seen that by

tracing the predecessor path of each label, we get the unique path from node 1 to the node associated with label and the algorithm has been able to enumerate all the paths.

algorithm *path enumeration-I*;
begin
    for each origin-node (say *s*)
        create a label with null predecessor at *s;*
        make a set $S := \{s\}$;
        while $S \neq \Phi$ **do**
            remove a label *l* from *S;*
            for each of the outgoing arcs (say *a*) from the node associated with *l*
                if number of trains taken to reach head node of arc *a* by the path represented by *l* and *a* is not more than *t*
                then create a new label at the head node of *a and* add the new label to *S;*
end.

Figure 5.2 Path enumeration algorithm restricted by the number of trains.

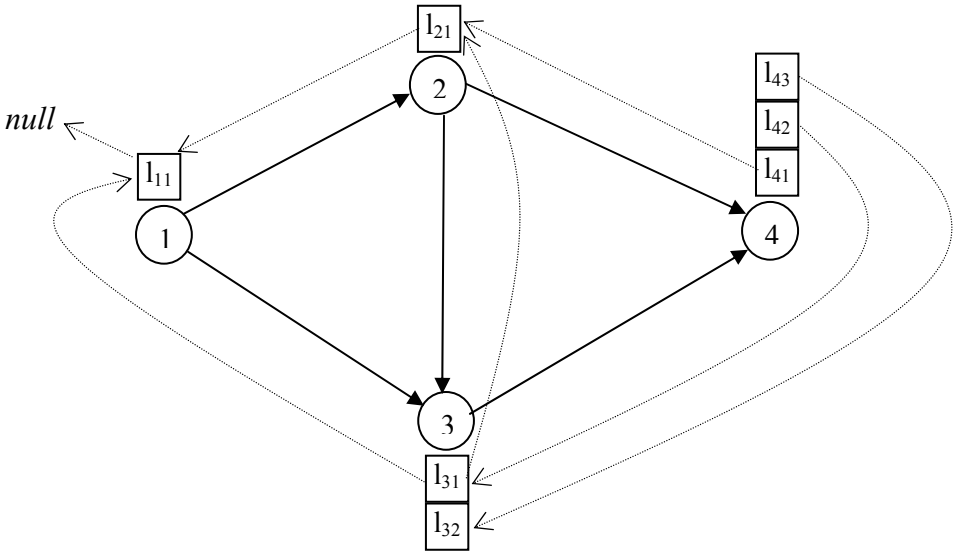


Figure 5.3 Example of path enumeration algorithm.

Theorem 1: *Algorithm in Figure 5.2 enumerates all the paths from the origin-node of a block to the nodes in the destination-node set, on which the number of trains taken is not more than t.*

Proof: We prove the above theorem by contradiction. Suppose that there exists a path p (p_1-p_2-p_3-...-p_k) from origin node (say p_1) to a node (say p_k) in the destination-node set, for which the number of trains is not more than t but our algorithm fails to find this path. As there exists an arc (p_1, p_2) and we created a label at p_1, we must also have created a label at p_2. Similarly, as we must have created a label at p_2, the algorithm examines the arc (p_2, p_3) and must have created a label at p_3, and so on, unless the number of trains taken in reaching a node (say p_i in path p) is more than t. Now, if the number of trains taken to reach p_i is more than t then we cannot reach p_k using path p and by taking trains not more than t, which contradicts our claim of existence of the path p. ♦

Our experiments on the real-life data provided by a major US railroad indicate that the above algorithm is able to enumerate all the feasible paths for all the blocks in few seconds of computational time on Pentium IV computer. However, we found that for some blocks the numbers of valid paths created by the algorithm described earlier were too large. For such blocks, we needed another criterion to reduce the number of valid paths further. We found that though several valid paths rode on no more than the specified number of trains, their paths are quite circuitous in terms of the distance of the path and/or the time taken by those paths. Railroads typically do not allow blocks to follow paths, which take more than a specific percentage over the shortest path (either distance or time). We next describe in Section 5.3.2 a method that keeps track of the distance of each valid path (or the time taken by it) and eliminates the paths, which do not meet the specified criteria.

5.3.2 Path Enumeration Algorithm Restricted by Number of Trains and Number of Labels

In the path enumeration algorithm discussed in Section 5.3.1, we make a simple modification and will keep only pre-fixed number of labels (say k) at a node. Hence, in this algorithm, whenever we create a new label at a node already having k labels, we replace one of the existing labels with new label, if it is of better quality. Now if the label to be replaced is already the predecessor of other labels, then the replacement of this label with the new label will make the entire successors of old label obsolete and hence should be removed. However, computationally it is very difficult to trace and remove all the successors of a label. Therefore, we need a mechanism in which a label is not required to be replaced, if it is the predecessor of other labels. In other words, before making a label predecessor of other labels, we ensure that the label is permanent and will not be replaced by any other label at any point of time.

In the BTA problem all the cost figures are nonnegative, which motivated us to develop an algorithm analogous to Dijkstra's algorithm. In Dijkstra's algorithm, we keep a distance label with each node, which denotes the cost of some path from source node to the node, and we iteratively make the distance label of a node permanent if this is the smallest distance label among all the distance labels, which are not permanent. In our algorithm, we also make a label permanent and enumerate paths from this label, if this is the best label among all the labels from which paths are required to be further enumerated. This path enumeration algorithm is given in Figure 5.4.

```
algorithm path enumeration-II;
begin
    for each origin-node (say s)
        create a label with null predecessor at s;
        make a set S := {s};
        while S ≠ Φ do
            take out the best label from S;
            for each of the outgoing arcs (say a) from the node associated with l
                if number of trains taken to reach head node of arc a by the path
                represented by l and a is not more than t
                then create a new label at the head node of a and add the new
                label to S;
end.
```

Figure 5.4 Path enumeration algorithm restricted by number of trains and number of labels.

Theorem 2: In the above algorithm, a permanent label will never be replaced by any new label.

Proof: We will prove this theorem by contradiction. Let us assume that we can keep at most k labels and there are k labels at a node i. Let us claim that there is k+1th label (say l_1) at node i, which is of better quality than a permanent label (say l_2) at node i. Now there exist two scenarios. In scenario 1, both the labels l_1 and l_2 have already been there at the time l_2 was made permanent and as we decided to make l_2 permanent, l_1 can not be better than l_2, which contradicts our claim. In scenario 2, label l_1 was not there when l_2 was made permanent. It implies that predecessor of l_1 must be made permanent after l_2 was made permanent and hence quality of predecessor of l_1 cannot be better than that of l_2. Also as all cost figures are non-negative, quality of l_2 cannot be better than that of any of its predecessor. It implies that l_1 cannot be of better quality than l_2, which again contradicts our claim. ♦

As our space-time network is very sparse, the size of the labels to be made permanent remains manageable if the maximum number of labels to be kept at a node is

not too high. However, this algorithm may exhibit inferior performance than that in Section 5.3.1, if the upper bound on the number of labels at a node is too high or is unrestricted. This is because of the maintenance of expensive data structure in the implementation of this algorithm.

In the algorithms discussed in Sections 5.3.1 and 5.3.2, we have been able to put some of the operational requirements while enumerating the paths. However, there are many other constraints (like height, width and weight restriction, etc.), which must be honored by a valid path. We can easily meet these requirements by simply ignoring all those paths, which violate any of these requirements. Therefore, in the path-based formulation there is no need of incidence vector and we can eliminate a large number of decision variables in the preprocessing stage. In following section, we describe few approaches to solve the path-based formulation of the BTA problem.

5.4 Solution Approaches

We have seen that the path-based formulation is more manageable than the arc-based IP formulation. Therefore, all of our approaches to solve the BTA problem are based on solving the path-based IP formulation. We have developed three approaches to solve the BTA problem: (i) solving to optimality using CPLEX, (ii) solving heuristically using Lagrangian relaxation, and (ii) solving heuristically using greedy algorithm.

To solve the BTA problem optimally, we used the commercially available CPLEX optimizer. It can be recalled that there are two types of constraints in path-based formulation: (i) each block must be assigned to a path, and (ii) a train cannot carry more cars than its capacity (called capacity constraint). Although, capacity constraints make a formulation hard, CPLEX has been able to solve a real-life BTA problem to optimality in

a very reasonable amount of time. This may be because of abundant train capacity on a large number of paths. Therefore, we cannot generalize this behavior of optimizer to all instances of the problem. Also, the branch and bound algorithm of CPLEX may not be scaleable to large problem size and may not be able to solve the problem of higher magnitude to optimality. In Section 5.4.1, we describe a heuristic to solve the BTA problem based on Lagrangian relaxation and in Section 5.4.2; we discuss the neighborhood search heuristic.

5.4.1 Lagrangian Relaxation Based Heuristic

It should be noted that in the absence of capacity constraints, the BTA problem reduces to assigning the least cost path to each block, which is trivially solvable. In other words, capacity constraints make the problem hard to solve. One common way of dealing with such constraints of an IP problem is Lagrangian relaxation (Fisher [1981]). In this approach, capacity constraints are relaxed and a penalty is assigned for each unit capacity violation on each train arc. The relaxed problem is solved iteratively with different value of penalty. If in iteration, no arc capacity is violated, the solution is optimal. On the other hand, if capacity constraints are violated on some arcs, we try to get feasible solution using heuristic approach and repeat the process with new penalty. Our heuristic to obtain feasible solution is similar to the greedy algorithm described in Section 5.4.2. In our heuristic, we first identify all the trains arcs on which capacity is violated and do not assign any path to the blocks flowing on these arcs. This step frees the capacity on few arcs and reduces the BTA problem to assigning path to few blocks. We apply greedy algorithm as in Section 5.4.2. to assign paths to these blocks. In the remainder of the section, we only describe Lagrangian relaxation approach as greedy algorithm is already discussed in Section 5.4.2.

In the Lagrangian relaxation approach, the capacity constraints are relaxed and we assign penalty $u_a \geq 0$ for each unit capacity violation on every train arc a and add the constraint to the objective function of path-based IP formulation. This may be interpreted as putting penalty u_a for each unit of violation of train capacity in a trip. The relaxed problem is given in formulation 5.3.

Formulation (5.3):

$$\min \sum_{b \in B} \sum_{p \in P_b} c_p x_p + \sum_{a \in A} u_a \Big(\sum_{\substack{p \in P'_a \\ b: p \in P_b}} v_b x_p - C_a \Big) \qquad 5.3a$$

subject to

$$\sum_{p \in P_b} x_p = 1 \qquad \forall b \in B \qquad 5.3b$$

$$x_p \in \{0,1\} \qquad \forall p \in P_b, \forall b \in B \qquad 5.3c$$

Let us call the above problem as P_u. It can be observed that for a fixed penalty vector u, problem P_u separates into independent sub-problems for each block b and is given in formulation 5.4.

Formulation (5.4):

$$\min \sum_{p \in P_b} \Big(c_p x_p + \sum_{a \in p} u_a v_b x_p \Big) \qquad 5.4a$$

subject to,

$$\sum_{p \in P_b} x_p = 1 \qquad 5.4b$$

$$x_p \in \{0,1\} \qquad \forall p \in P_b, \qquad 5.4c$$

For the above problem, we only need to assign the best path with respect to objective function (4a) to a block, which is very easy and can be solved trivially. Let φ(u) is the optimal solution to P_u for $u \geq 0$, then it is standard that $\varphi(u)$ provides a lower bound

to the BTA problem. Now the problem of finding the best possible lower bound is the so called Lagrangian dual problem and is given in formulation 5.5.

Formulation (5.5)

max φ(u) 5.5a

subject to $u \geq 0$. 5.5b

The approach to solve the problem 5.5 works as follows. We perform multiple iterations with different value of u. At iteration k, using the multipliers, u^k, we first solve the relaxed problem (P_u^k). Thereafter, we update the multipliers to get closer to the solution of 5.5 and repeat the procedure. In an iteration, we update the multipliers using the sub-gradient optimization technique. The basic idea of updating multiplier is simple. If in an iteration, capacity is violated on some arc (say a), then we set $u_a^{k+1} > u_a^k$ making the flow on this arc unattractive and else if $u_a^k > 0$ and there is no capacity violation then we set $u_a^{k+1} < u_a^k$ making the arc more attractive. We use the following formula to update multiplier u_a for an arc a in iteration k.

$$u_a^{k+1} = u_a^k + \theta_k \left(\sum_{\substack{p \in P'_a \\ b: p \in P_b}} v_b x_p - C_a \right),$$

where θ_k is the step length in iteration k and is calculated using following formula:

$$\theta_k = \frac{\lambda_k (UB - \varphi_k)}{\| \sum_{\substack{p \in P'_a \\ b: p \in P_b}} v_b x_p - C_a \|^2},$$

where λ_k is a scalar UB is an upper bound on the optimal solution of original problem.

$$\|y\| = \left(\sum y_i^2\right)^{\frac{1}{2}}.$$

Above algorithm to obtain lower bound of the BTA problem is summarized in Figure 5.5. Our computational experiment on the real-life data indicate that very good lower bound can be obtained for the BTA problem in few seconds of computational time.

algorithm *lower bounding scheme;*
begin
set k := 1 and initialize u^1 by setting u_a^1 := 0 for all $a \in A$;
set stopping criteria;
until stopping criteria is met **do**
assign the best path to a block using objective function (4a)
find the new multiplier u^{k+1};
set $k := k+1$;
end.

Figure 5.5 Lagrangian relaxation based lower bounding scheme.

5.4.2 The Greedy Heuristic Algorithms

As noted earlier, the optimal solution obtained by the integer programming software CPLEX may not be scaleable to the size of the problem or the computational time taken in solving the problem may not be acceptable to the railroads. Further, railroads are more interested in using an algorithm which is time efficient, is flexible in adjusting parameters and requirements, and produces good quality solution (not necessarily optimal solution). Lagrangian relaxation based heuristic is one approach to get such solution. In this section, we have developed another approach called greedy algorithm, which is able to produce near-optimal solution very efficiently.

Greedy heuristics are the simplest and the most popular algorithmic approach to solve combinatorial optimization problems. In this approach, values are assigned to decision variables based on certain greedy criteria. There are many variations of this algorithm in use. For BTA problem, we have developed an approach, which is able to produce reasonably good quality solution in fraction of second. In addition, our algorithm

provides flexibility to users to specify the greedy criteria at run time, a feature highly desired by the railroads.

In our greedy algorithm, we first order the blocks, based on certain priority and then assign the best feasible path to blocks in that order. At each stage of assignment of path, we ensure that train capacity is not violated on any train-arc in the space-time network. Figure 5.6 describes our greedy algorithm.

```
algorithm greedy heuristic;
begin
        order the blocks in descending order of some priority;
        for each block b in above order
                order the paths in ascending order of path cost;
                for each path p in above order
                        if none of the train capacity is violated on path p by assigning  block b
                        then assign path p to block b;
                                update the available train capacity of train arc on path p;
                                exit for;
end.
```

Figure 5.6 Greedy Heuristic for the BTA problem.

In the above algorithm, the user has the flexibility to specify the criteria on which blocks can be prioritized. Generally, there are many factors, which govern the criteria selection. Some of these criteria are listed below.

- Give high priority to the blocks having less number of possible paths, so that, for each block there will be sufficient options while selecting a path.
- Give high priority to the blocks having more number of cars, so that, best path can be assigned to them and cost can be minimized.
- Give high priority to the blocks containing shipments with high service level (service level signifies the importance of the shipment), so that, paths with least transit time can be assigned to high priority blocks.
- Give high priority to the blocks for which shortest distance from their origins to their respective destinations is maximum. This will give the possibility of assigning best paths to the blocks traveling long distance.

As discussed earlier, there are other capacity constraints on trains, which are soft constraints (such as, the maximum length of a train, the maximum weight of a train, etc.).

However, railroads may want to make these constraints hard on all the train-arcs or on some of them. The greedy algorithm gives the flexibility to users to impose these restrictions strictly. This can be simply incorporated by adding extra checks at the time we check the feasibility of a path in the greedy algorithm. In Section 5.5, we report the computational result for the case when we prioritize the blocks by the number of possible paths in the space-time network**.**

5.5 Computational Results

In this section, we present our computational results. We were supplied data for real-life problem by a major US railroad. All of our algorithms were implemented in C++ and we used ILOG CPLEX 8.1 to solve the BTA problem to optimality. The algorithms were run and tested on a Xeon PC with a speed of 2.4 GHz.

Railroad network of the company, which provided us the data, consists of over 6,000 stations and over 6,000 links connecting the stations. To transport the shipments, they run around 350 trains daily. The total number of stops in the network, where a block can be attached or detached from a train is around 2,000. On the average, the railroad makes over 1,200 blocks every day.

It can be recalled from Section 5.4 that to solve the BTA problem, we have developed three approaches: (i) solving to optimality, (ii) solving heuristically using Lagrangian relaxation, and (iii) solving heuristically using greedy method. In our first investigation, we measured the quality of solution obtained by implementing these approaches. For each of them, Table 1 gives the percentage deviation from the optimal solution and the computational time taken. This analysis corresponds to the case when we put a restriction of maximum 50 labels at each node in path enumeration algorithm. It can

be seen that the Lagrangian relaxation approach seems to be the best approach, as it gives very good quality solution in few seconds of computational time. The time efficiency of Lagrangian relaxation approach can be attributed to the fast convergence of the algorithm. For the instance under consideration, it converges to best lower bound in 143 iterations. It is worthy to note here that the lower bound of the optimal solution is only 0.03 % away from the optimal solution.

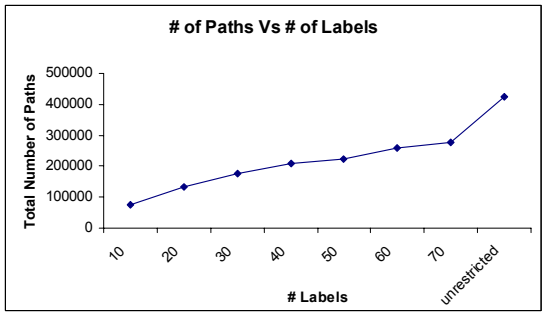
Table 5.1 Performance analysis of different solution approaches.

Approach	**% deviation from optimal sol.**	**Computational Time**
Optimal Approach	0%	97 sec
Greedy Approach	1.43 %	1 sec
Lagrangian Relaxation Approach	0.78 %	6 sec

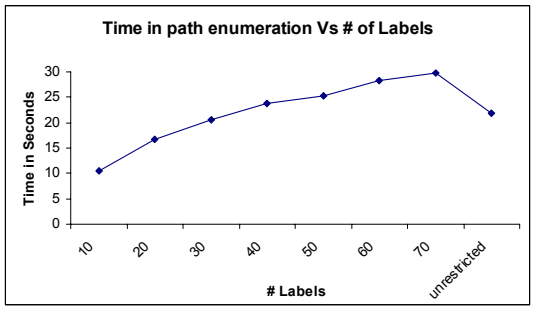
In Section 5.3, we have discussed two algorithms for path enumeration in space-time network. We also indicated that both algorithms have their own merit, depending on the restriction imposed on the number of paths to be enumerated. We analyze here the performance of these two algorithms. It can be recalled that to restrict the number of paths enumerated, we put the restriction on the maximum number of labels at a node in space-time network. Figure 5.7(a) and Figure 5.7(b) give the total number of paths enumerated and time taken in path enumeration respectively for different level of number of labels at a node. It can be seen that when we put restriction on the number of labels, the time taken in path enumeration proportionately increases with the number of paths enumerated. However, as noted earlier, if the restriction on maximum number of labels at a node is too high, it is a better option to remove this restriction and enumerate all the paths. For example, the time taken in path enumeration is around 30 seconds when we

keep at most 70 labels at each node, whereas this time is only 22 seconds in unrestricted case.

We have also analyzed the quality of optimal solution where we put the restriction on maximum number of labels at a node. Figure 5.7(c) gives the percentage deviation of the optimal solution in each case from the optimal solution of the problem, when there is no restriction on number of paths. In Figure 5.7(d), we also report the time taken by CPLEX in each case. It can be seen that although the quality of the solution is the best when we do not put any restriction on number of labels at a node; however, it also takes more computational time. In a real-life implementation, users can trade-off between the optimal objective function value and computational time. Further, even if the quality of optimal solution is slightly better in the unrestricted case, railroads may put restriction on the number of paths as they do not want to assign an inferior path in terms of transit time to any of the block. This may be required for the customer satisfaction or goodwill of the organization.

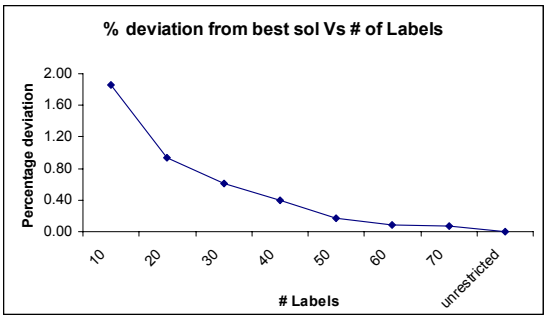


(a) Total number of paths enumerated

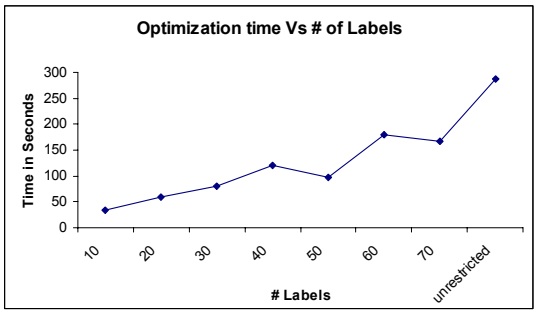


(b) Time taken in path enumeration

Figure 5.7 Analysis of performance with respect to maximum number of labels at a node.



(c) Percentage deviation from best solution



(d) time taken in solving to optimality

Figure 5.7-Continued Analysis of performance with respect to maximum number of labels at a node.

5.6 Summary and Conclusions

In this chapter, we consider the block-to-train assignment problem in rail transportation. We defined the problem on a space-time network and gave two integer programming formulations. Of these two formulations, the path-based integer programming formulation is more flexible and contains much less variables and constraints. One basic assumption in the path-based formulation is that a block can be assigned only to a predetermined set of paths; however, the user can specify the number of such paths. This formulation is manageable enough to be solved to optimality in reasonable time using a commercial IP solver. We also developed several heuristic algorithms to solve the BTA problem, including a Lagrangian relaxation based heuristic and several greedy construction heuristics. All of these heuristics proceed by enumerating feasible shortest paths and augmenting flows along those paths. We performed an extensive computational testing of this algorithm on the data provided by a major US railroad and report the results of this testing. We found that the integer programming formulation of the BTA problem can be solved to optimality within two minutes using CPLEX 8.1, the Lagrangian heuristic algorithm can obtain a solution within 1% of the

optimal solution in about six seconds, and greedy construction heuristics can obtain solutions within 2% in just one second. Hence, we find that the daily version of the BTA problem, we considered in this paper, is quite efficiently solvable. We plan to pursue research to solve the weekly BTA problem, which is substantially larger, the daily BTA problem.

CHAPTER 6
TRAIN SCHEDULE DESIGN PROBLEM

6.1 Introduction

As described in Chapter 5, railroads are full of large size and complex optimization problems. The arena of optimization problems in railroad application consists of the blocking problem, the train scheduling problem, the locomotive assignment problem, the crew scheduling problem, the block-to-train assignment problem, etc. In this chapter, we study one of the most important railroad optimization problems, called, train scheduling problem. This problem can be defined in three contexts: (i) zero based train scheduling problem in which decisions are made at the planning level about the trains to be made, their frequency, timings, routes, etc., (ii) incremental train scheduling problem, in which decisions are made about the change in existing train schedule to improve efficiency in transportation for given demand of shipments on planning level, and (iii) real time train scheduling problem, in which decisions are taken based on the positions of trains at a given time. In this chapter, we study zero based train scheduling problem and we call this problem as train schedule design problem.

In the train schedule design problem, the inputs are the blocking plan (refer chapter 5 for details) and the physical rail network. The decisions to be made are with respect to formation of train and deciding their schedule. The constraints are on the capacities of a train and number of trains that can stop at a station. Apart from these, there are many practical constraints which must be honored by a solution of the problem. Considering a

real life instance as in chapter 5, there will be millions of decision variables and constraints. Therefore, almost all the efforts in the past are made in developing heuristics to solve the train schedule design problem to near-optimality. We next describe the literatures available to solve the train schedule design problem.

The train schedule design problem consists of two parts (a) developing the train schedule, (b) routing the shipments. The problem of routing the shipments is a multi-commodity flow problem with side constraints. Generally a blocking plan is developed first and then the routings of the blocks are developed. Chapter 5 of this dissertation addresses the problem of block to train assignment and we have given the related past research to solve this problem in same chapter. In this chapter, we briefly present the literatures discussing the development of train schedule.

Thomet [1971] considers the trade-offs of replacing direct trains with a series of connections ignoring classification strategies. Morlok and Peterson [1970] develop timetables for a railroad without defining train connections and frequencies. Assad [1982] is concerned with optimal stop schedules of trains, but gives no consideration to routing of shipments. Van Dyke [1986] and Bodin et al. [1980] are concerned with the classification of shipments given a set train schedule, but do not consider the effect of classification strategy on the schedule.

Assad [1980] was the first to consider the train itinerary and makeup problems in the same study. He suggests Bender's decomposition as a possible approach to solving his model; however, he does not solve it. More recent studies have attempted to model and solve the joint train scheduling, shipment routing problem: Keaton [1989, 1991, 1992a, 1992b], Crainic et al. [1984], Crainic and Rosseau [1986], and Haghani [1989]. In

each study, the problem is broken into two components: a train-scheduling problem and a shipment routing problem. An iterative procedure is used to solve each of the two parts of the problem in succession, using the solution from the other part to guide the next iteration. Each author relies on the separability of the train scheduling problem and shipment routing problem. None of the models are meant to produce a detailed schedule for the railroad; they produce only train frequencies for a representative day's schedule which provides a rough framework for creating a schedule.

Keaton [1992a, 1992b] uses Lagrangian relaxation to simplify the problem. He incorporates the train capacity, travel time and demand flow constraints into the objective with a Lagrangian multiplier. Relaxing the constraints allows him to decompose the problem into separable train and shipment routing problems. By relaxing the train capacity constraint in his model, the shipment routing problem may be viewed as a collection of shortest path problems. Using a dual adjustment approach, the author arrives at an infeasible lower bound to the problem, then uses a simple heuristic to arrive at a feasible solution. Haghani[1989] incorporates empty equipment flows into his model, which are ignored by the other studies. He schedules them against variable daily demands but uses a fixed daily train frequency. The other two models schedule against a representative day's demands. He applies his model to a smaller network of eight nodes and ten two-way arcs. His heuristic decomposition technique depends on the special structure of his formulation and becomes untenable for larger problems. Crainic and Rosseau [1986] present a general model of multimode freight transportation, and Crainic et al. [1984] examine the specific case of freight rail. They solve the problem through decomposition and column generation techniques. A modified shortest path algorithm in

which the capacity constraints are integrated into the objective with a penalty term is used to find the best route of the shipments given a schedule. The information from the demand flow is used to generate a new train schedule. Gorman [1998] applies meta-heuristics: tabu search and genetic algorithm to solve an integrated train scheduling problem. The strength of his approach lies in considering weekly train schedule and in modeling the problem with broader scope.

Our work in this dissertation, consider the problem instance in totality, as against the past research on solving a subset of problems. We have developed a decomposition approach to solve the problem. However, our approach does not separate the train scheduling and shipment routing. We have developed heuristic algorithms to solve each of the decomposed problems. The strength of our approach is in decomposing the problem in sequential phases and developing efficient algorithms to solve the problems in each stage.

Rest of the chapter is organized as follows. In section 6.2, we describe the train schedule design problem in more details. In section 6.3, we present the decomposition strategy. Sections 6.4 and 6.5 give the solution approaches to solve the problems in Phase I and Phase II respectively. In section 6.6 we present the computational experience with a real-life instance and in section 6.7, conclusions and scope for further research are given. Also, in rest of the paper we invariably use train schedule design problem as train design problem.

6.2 Problem Description

As discussed in Section 6.1, the train design problem is closely related to other railroad scheduling problems. This problem takes input from the solutions of other

scheduling problems and produces a solution, which optimizes the objectives and honors many constraints, which are essential for the feasible execution of the schedule. The constraints set of this problem includes many historical and contractual requirements, which are very difficult to be represented mathematically. In this section we have tried to include as many constraints as possible. We describe below the inputs, constraints, objective functions and decision variables of the train design problem.

6.2.1 Input

The train design problem is to design and to schedule trains to transport given shipments using existing physical infrastructure, like, physical network, locomotives, etc. As described in section 6.1, generally blocking plans are developed first, which consolidate the shipments into blocks and the demand for the train design problem can be specified in terms of transportation of blocks. Therefore, the essential inputs for the train design problem are the existing blocking plan and the existing physical network, which includes the stations and links in the network. Apart from these, we also need input on the capacities available in network, costs of transportation and use of infrastructure, and matrices to measure the goodness of the solution. Also, very often matrices are in different units of measurement, we need some conversion factors to weigh them against each other.

6.2.2 Objective Function

In the train design problem, there are mainly two types of objectives: one which minimizes the transportation cost and transit time of shipments, and another which helps in smooth operation of the train schedule. We present below these objectives and also discuss their importance in the context of overall objective

- A big part of railroad expenditure goes in transporting shipments from their origins to their respective destinations. This cost consists of expenditure in operating the trains, maintaining the trains, managing ground operations, fuel cost and cost of capital utilized. The transportation cost of shipments greatly affects the ground level economic performance of the railroad and hence minimizing the transportation cost is the most important objective of the railroad.

- Railroads are facing stiff competition from the trucking industry and the greatest advantage of the transportation by the truck over that by rail is less transit time. From the supply chain perspective and customer satisfaction, transit time is an important factor. Apart from this, costs proportional to the transit time are also incurred in terms of the capital of the customer stuck up during transportation. Therefore, one of the important objectives of the train design problem is to minimize the transit time of shipments.

- Whenever, a train is engaged in the transportation of shipments, railroads have to incur some cost of infrastructure and crews. Therefore, to have economic advantage, railroads want to utilize minimum number of trains and this is one of the important objective of the train design problem.

- As discussed earlier, the solution of the train design problem should not warrant for additional physical infrastructure. It implies that solution should be such that it does not cause congestion and infeasibility at the stations or on the links. To achieve this, one of the objectives of the train design problem is to minimize the number of stops of a train as well as to minimize the number of trains stopping at a station.

- For the smooth operation of the trains, it is highly desired by the railroads that there should be consistency in the operation of trains. For example, if a block is made everyday of the week, then the railroad will like to assign these blocks to same train sequence everyday. Therefore, one of the objectives of the train design problem is to minimize the inconsistency in block-to-train assignment.

It can be noted that some of these objectives may be rephrased and can be considered as the constraints. However, including them in objective function makes the mathematical formulation easier to solve. It can also be noted that some of objectives are conflicting in nature. For example, to minimize the transit time we can create direct trains from the origin to the destination of each block. However, this will adversely affect the objectives of minimum number of trains and minimum transportation cost. Therefore, a trade-off is needed to balance these objectives and input is required to prioritize the objectives.

6.2.3 Decision Variables

The train design problem is a complex discrete optimization problem in which many decisions are required to be made. We present below the decision variables for this problem.

- As we are considering zero-based train design problem here, decisions are required to be made with respect to the trains to be made for the transportation of shipment. Then for each train, decisions are required to be made with respect to the origin and the destination of the train, physical route of the train, stops on the route of the train, frequency of the train and its time schedule.
- As the trains are designed to carry the shipments from their origins to their respective destinations, decisions are to be made with respect to train roué for each shipment. However, as the shipments are already consolidated into blocks, the decision variable for the train design problem is to find train route for each block. This problem is also called the block-to-train assignment problem and is studied in previous chapter, when we determine the train route for blocks using given train schedule.

6.2.4 Constraints

The constraints set for the train design problem consists of logical requirements for the feasible execution of the train schedule and practical requirements for the smooth implementation of the solution. We next describe these two types of constraints separately.

Logical requirements

- Each train must have a valid route and valid schedule, i.e., two consecutive stops of the train must be physically connected and timings at the stops must be logical and feasible.
- Each block must be assigned to a valid sequence of trains. For example, if a block changes train at a station, then this station must be valid stops for both the trains.

Practical Requirements

- The load on a train should not exceed the capacity restriction. The capacity restrictions on a train are given in terms of maximum number of cars in the train, maximum weight of the train, and maximum length of the train. Depending on the case, railroads may need all or some of these restrictions imposed while assigning

blocks to the trains. Also, the capacity of a train may vary on different parts of its route.

- If pre-specified, a block should not flow on particular links. Sometime, this restriction is imposed to prevent the flow of special kind of shipments (like hazardous material) on certain route.
- If pre-specified, a block must flow on a particular route. This restriction may be imposed to meet the specific customer requirement or to meet legal requirements.

Apart from above, there may be many additional requirements which must be honored. For example, certain trains must be made, a train must follow given timings, a block must be swapped at a particular yard, etc. In this chapter, we have tried to develop algorithms, in which these requirements can be easily incorporated.

We now discuss the size of a typical US railroad train design problem in terms of number of decision variables and number of constraints. Let us suppose that there are 6000 stations and 12000 links in the physical network and around 1300 blocks are made everyday. Now let us assume that we have a upper limit on number of trains to be formed (say 500). Then for each of these trains, there are 6000 possible origin and 6000 possible destination. Also, for each train we will have to select a proper set of links to form the route of trains. Once the train route is decided, we have 24 X 4 = 96 options for each train as starting time at origin. We assume here that once the starting time of a train is known, we can determine its timing on the route. Needless to say that altogether the number of decision variables will be in millions. It can be recalled that another important decision variable for the train design problem is to find the block-to-train assignment. It can be recalled from Chapter 5, that once the train schedule is known, number of decision variables for the block-to-train assignment problem was around 12 millions. Now when we are solving an integrated problem, these decision variables will be much more. The real-life train schedule of a railroad indicates that the number of stops on the route of a

train varies from 2 to 200. If we treat the block to train assignment problem as multi-commodity flow problem, then we need mass constraint for each of the block at each station (around 1300 X 6000). Apart from this, we also need the logical constraint for each of the train at each stop on its route. As we do not know the train route in advance, the number of flow constraints for trains will be around 500 X 6000. Besides these, we will have constraints for capacity requirements and logical timings on the route. Considering altogether, the number of constraints will be in millions. With normally used computational facility, it is extremely difficult to solve any mathematical formulation of the train design problem to optimality. Therefore, in this chapter, we are only concentrating on development of heuristics.

6.3 Problem Decomposition

As described in previous section, it is extremely difficult to solve the train design problem in totality using normally used computational resources. Therefore, it is prudent to develop some algorithms, which can produce nearly optimal solution. However, as there are millions of decision variables involved, it is again very difficult to solve the train design problem in totality to obtain nearly optimal solution. To keep the problem size manageable, we have developed an approach, which decomposes the train design problem in three stages. In our decomposition technique, the problem in Phase I is a network design problem, that in Phase II is a network flow problem and problem in Phase III is a scheduling problem. We next briefly describe the problem in each phase.

6.3.1 Phase I

It can be recalled that one of the input to the train design problem is the blocking plan and decisions are to be made with respect to assigning blocks to train sequences so

as to minimize the transportation cost. In a train sequence, a block is picked by a train at a station and then dropped at another station, where it is picked up by another train and so on until the block reaches its destination. Now, if we call the train path between two consecutive stop as the *train segment*, then in a train sequence, a block rides on one or more train segments of a train. If we treat all the train segments of same type irrespective of the trains to which they are associated then finding a train sequence of a block is equivalent to finding the sequence of train segments. In view of train segment concept, we can define the train design problem as to designing the train segments, routing the blocks on train segments and then combining the segments to create the trains. Now, in the train design problem, there is time associated with each train and hence there should be time component associated with each segment as well. However, in this formulation, as we do not know the train of which a train segment is part of, deciding the time component of a segment is impossible. To simplify this, we do not decide the time component of a train segment in this phase. Once we know of which train a segment is part of, the decisions regarding its timing are taken. Thus the problem in this phase is defined as to design the train segments and to route the blocks on a network consisting of train segments. The objectives in this phase are to minimize the transportation cost and to minimize the number of train segments terminating at a station.

6.3.2 Phase II

From Phase I, we get the train segments to be made and the route for each block. At this point we may combine train segments to create the trains. When combined, the end points of train segments denote the stops of a train. If we construct a network consisting of train segments then creating trains are equivalent of creating paths in the network such that all arcs are covered and each arc can be the member of only one path. Now, flow of

blocks can be represented as the flow on paths. Whenever, a block flows from one path to another, there is change in train and any transition in trains results into increase in transit time and transportation cost. Therefore, one of the objectives in this phase is to construct the trains in such a way that block swapping (train-transition) can be minimized. In a train we also want to keep the consistency in train load, i.e., a train should carry same load on different segments as same set of locomotives are assigned throughout its itinerary and there will be underutilization of power it pulls less load on part of its itinerary. Again as in Phase I, we do not determine the time component of trains in this phase.

6.3.3 Phase III

From Phase I and Phase II, we know the trains, their physical route and block-to-train assignment. In Phase III, we take rest of the decisions for the train design problem. This includes the time schedule of trains and their frequency. The objectives in this phase are to minimize the transit time of shipments and maximize the consistency in block-to-train assignment problem. The constraints in this phase are to honor the capacity restrictions and many other practical restrictions.

It can be noticed that by using decomposition technique, we can have better formulation of the problem in each stage and as described in remainder of the chapter, we can develop time efficient algorithm for the problems in different phases. It can be also noted that performance of an algorithm in a phase greatly affects the performance in subsequent phases. For example, a good design of train segments is very important to construct good trains and a good set of trains is very important for scheduling. To improve the performance of an algorithm in a phase, we relax some of the constraints and impose them in subsequent phases. For example, we relax the capacity constraints in

Phase I and Phase II and impose them in Phase III. Now if there are capacity violations from Phase I and Phase II, we can address them by splitting the trains or by adjusting the frequency of the train. We now briefly describe the simplifications in each phase to achieve efficiency.

It can be recalled from Section 6.1, that generally a train schedule is repeated every week on the planning horizon. It implies that the solution of the train design problem should be weekly train schedule. However, if we consider weekly time horizon in Phase I and Phase II, then it may result into different train route for a block on different days of week, which is undesirable for the railroads as they want to maintain consistency in block-to-train assignment. Therefore, we consider only the daily version of the problem in Phase I and Phase II, which produces the trains with the assumption that they will run everyday of week. It implies that before applying Phase III, we assume that frequency of each train is seven, which is adjusted by the algorithms developed in that phase. Also, in Phase I and Phase II, we do not impose the capacity restrictions on trains, as the loads on train are not the correct representation of train load on weekly time horizon. Also, we can always split up a train if there is too much load on it and conversely, combine two trains in one, if loads on trains are too low. Relaxing the capacity constraints on trains in Phase I and Phase II allowed us to develop efficient algorithm.

Now, we will briefly discuss the limitations in developing algorithms for each phase. In Phase I and II, inputs required are the blocking plan, physical network, limitations on number of trains and number of train segments at a station. These inputs are easy to obtain from the railroads and they do not have to make extra effort to generate the data. However, in Phase III, we need much more data regarding the timings of the

trains, travel time, crew policy, maintenance requirements, capacity restrictions on each train segments, etc. Needless to say, high levels of details are required for these data and deep coordination with railroads is required to develop the algorithm in Phase III. There are many more practical requirements are there, which must be incorporated in the final solution and again mutual work is required to be done with railroads to precisely and quantitatively define these additional constraints. Therefore, in this chapter, we only address the problems in Phase I and Phase II and keep the development of algorithms in Phase III as the future work. In remainder of this chapter, we will only discuss the train design problem in scope of problems defined in Phase I and II. In section 4, we describe the solution approach to solve the decomposed problems in Phase I and Phase II.

6.4 Phase I: Designing Train Segments

As described in previous section, the problem in Phase I is defined as to create the train segments and to route the blocks on the network created. The objectives in this phase are to minimize the transportation cost and to minimize the train segments emanating from a station. The constraints for this problem are the maximum number of train segments that can be created at a node and valid route for each block. We call the problem in this phase as *Train Segment Design* (TSD) Problem. The TSD problem is analogous to the blocking problem described in previous chapter. In blocking problem, the decision variables are to create block-arcs and then to route the shipment over the network consisting of stations and block-arcs so as to minimize distance traveled and number of intermediate handlings of blocks. In the blocking problem, the constraints are on the number of block-arcs that can be created at a station. Liu [2003] describes the blocking problem and solution

approaches to solve the problem. Our approach to solve Phase I problem is very similar to that of Liu [2003].

Mathematically, the TSD problem is designing a network, called a *train segment* network, and routing blocks over this network so as to optimize the total shipment cost. Figure 6.1 gives a sample train segment network, where there are three types of nodes: *origins* (where blocks originate), *yards* (where blocks are swapped), and *destinations* (where blocks terminate). We show here a simplified network as in practice yards can be origins as well as destinations, and nodes can send as well as receive shipments. Each arc in the network represents a *segment.* At the tail of the arc, the train can pick up the blocks and at the head of the arc, it can drop the blocks. This signifies that no activity by the train will be performed at the intermediate stations on a train segment.

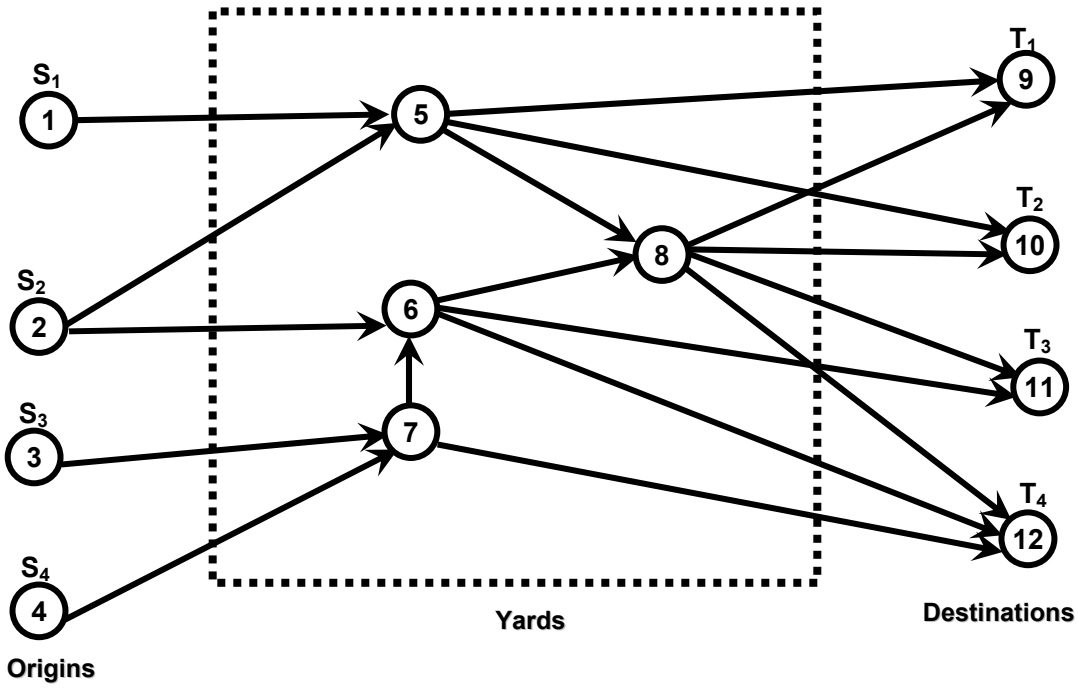


Figure 6.1 An example of a train segment network.

Blocks travel over the train segment network. We show four such blocks S_1, S_2, S_3, and S_4 which originate at the nodes 1, 2, 3, and 4, respectively, and are destined for the nodes 9, 10, 11, and 12, respectively. A block goes from its origin to some yard, then goes from one yard to another yard one or more number of times, and then goes to its destination. A block is picked up at its origin and subsequently swapped each time it changes train. For example, block S_2 can follow two paths 2-5-10 or 2-5-8-10, or 2-6-8-10 from its origin to its destination. For the path 2-5-10, it is picked up at node 2 and swapped once at node 5, if the train on segment 5-10 is different than that on 2-5. For the path 2-5-8-10, it is picked up at node 2 and swapped twice at the nodes 5 and 8, if trains on different segments are different. We refer to a path of a block over the train segment network as a *train segment path*.

The cost of routing a block is the sum of the cost of flow over the train segment network and the number of swapping it goes through. The cost of flow on a segment is linearly proportional to the length of the arc (i, j) in miles, where the length of a arc (i, j) can be considered to be the shortest route in the physical railroad network from node i to node j. If we assume that a block is swapped every time it changes train segment, each car is picked up at its origin and is swapped at each intermediate node of the train path it visits. It can be noted that number of swapping thus calculated is an upper bound on the actual number of swapping as if two segments are part of same train then there will be no swapping. Since the cost of picking up of each block at its origin is independent of the train segment path, we do not consider the picking up cost but consider only the swapping costs in our model. Each arc (i, j) in the train segment network has an associated cost c_{ij} denoting the cost of flow per car; this cost might depend upon the

distance between the nodes i to node j in the physical railroad. Further, each node i in the train segment network has an associated swapping cost d_i. In terms of these notations, the costs of the train segment paths 2-5-9, 2-5-8-9, or 2-6-8-9 are as given below.

Table 6.1 Illustrating the cost of flow in the blocking network.

Path	**Cost of path per unit flow**	**Cost of swapping per unit flow**
2-5-10	$c_{25} + c_{5,10}$	d_5
2-5-8-10	$c_{25} + c_{58} + c_{8,10}$	$d_5 + d_8$
2-6-8-10	$c_{25} + c_{68} + c_{8,10}$	$d_6 + d_8$

We determine the transportation cost of each block by adding two cost terms: (i) the cost of each segment path multiplied by the number of cars in and the average cost per mile per shipment, and (ii) the cost of swapping of a shipment multiplied by the number of cars in it and the average handling cost per swapping. The TSD problem is to design the train segment network and route all the blocks over the train segment network so that the total cost of transportation is minimum.

The TSD problem can be formulated as an Integer Programming problem, with two kinds of binary decision variables:

N: a set of nodes denoting the stations where blocks originate, terminate or are swapped. We will use the index i to denote a station in the node set.

A: a set of potential train segment arcs in $N \times N$, i.e., $(i, j) \in A$ if a segment can be built from node i to node j.

$G = (N, A)$: the blocking network.

$\delta^+(i)$: the set of arcs in A emanating from i.

$\delta^-(i)$: the set of arcs in A entering node i.

K: a set of blocks. We will use index k to denote a block.

$o(k)$: the origin of block $k \in K$.

$d(k)$: the destination of block $k \in K$.

v_k: the number of cars in block $k \in K$.

m_{ij}: the cost of flow per car on $(i, j) \in A$. In general, this cost is proportional to the length of the block.

b_i : the maximum number of train segments that can be made at node $i \in N$.

h_i: the cost of swapping a car at node $i \in N$.

c_{ij}: the cost of creating a train segment $(i, j) \in A$.

The TSD problem requires that (i) each block $k \in K$ consisting of v_k cars must be sent along a single path in the train segment network; and (ii) the number of train segment arcs emanating from node i be at most b_i. The objective function in the TSD problem is to minimize the weighted sum of flow cost and swapping cost.

The TSD problem has two sets of binary decision variables: y_{ij} and x_{ij}^k. The variable y_{ij} takes value 1 if we decide to build the arc $(i, j) \in A$, and is 0 otherwise. The decision variable x_{ij}^k is 1 if block k flows on the arc $(i, j) \in A$, and is 0 otherwise. The

TSD problem can be formulated as the following mathematical programming problem:

$$\min \sum_{k \in K} \sum_{(i,j) \in A} m_{ij} x_{ij}^k + \sum_{i \in N} \sum_{k \in K} \sum_{(i,j) \in \delta^+(i)} h_i x_{ij}^k \qquad 6.1a$$

subject to

$$\sum_{(i,j) \in \delta_i^+} x_{ij}^k \; - \; \sum_{(i,j) \in \delta_i^-} x_{ij}^k = \begin{cases} v_k & \text{if } i = o(k) \\ 0 & \text{if } i \neq o(k) \text{ or } d(k), \\ -v_k & \text{if } i = d(k) \end{cases} \quad \text{for all } k \in K, \qquad 6.1b$$

$$\sum_{k \in K} x_{ij}^k \leq u_{ij} y_{ij}, \qquad \text{for all } (i,j) \in A, \qquad 6.1c$$

$$\sum_{(i,j) \in \delta^+(i)} y_{ij} \leq b_i, \qquad \text{for all } i \in N, \qquad 6.1d$$

$y_{ij} = 0$ or 1 for each $(i,j) \in A$, and $x_{ij}^k = 0$ or v_k for all $(i,j) \in A$ and all $k \in K$. 6.1e

In the above formulation, the constraint 6.1b in conjunction with the constraint 6.1f ensures that a block flows on a single path in the train segment network, and the constraint 6.1c ensures that block can flow on an arc only if it is. The constraint 6.1d restricts the number of blocks created at a node.

Even the TSD problem is very large-scale problem and it is not possible to find an optimal or a near-optimal optimal solution for this problem using currently available commercial-level software and computing power. Hence, we focused our effort on developing heuristic algorithms for this problem that do not guarantee optimality of the solution but find a nearly optimal solution within a reasonable computational time. Neighborhood search algorithms are perhaps the most effective heuristic algorithms to solve difficult combinatorial optimization problems and, hence, we develop a neighborhood search algorithm to solve the TSD problem.

Figure 6.2 describes our VLSN search algorithm for TSD problem.

```
algorithm VLSN-TSD;
begin
    construct an initial train segment network and send all blocks along shortest paths in the
    train segment network;
    while the current solution is not locally optimal do
    for each node i ∈ N do {one pass}
        reoptimize the segments emanating from node i;
        send all blocks along shortest paths in the updated train segment network;
    end; {one pass}
end;
```

Figure 6.2 The VLSN search algorithm for the TSD problem.

This algorithm is a neighborhood search algorithm. It starts with a feasible solution of the TSD problem and iteratively improves the current solution by replacing it by its neighbor until the current cannot be improved. The neighbor of a solution is defined as all the solutions we can obtain by changing the segments at one of the nodes in the train segment network. This neighborhood search algorithm is a VLSN search algorithm because the number of neighbors is very large. First, there are n ways to select the node for which we change the segments emanating from it. Secondly, the number of ways we can select k segments out of a node with p arcs emanating from it is ${}^{p}C_{k}$ which grows exponentially with p and k.

This algorithm has two important subroutines: to construct the initial feasible solution; and to reoptimize the train segments emanating from a node. We now describe these steps in greater detail.

6.4.1 Constructing an Initial Feasible Solution

We use a fairly simple construction heuristic to create the initial solution of the train segment network. A solution of the train segment network must ensure that for each block its origin node is connected to its destination node through a train segment path. To ensure this, we first construct a directed cycle passing through all the yards exactly once

and returning to the first yard (that is, a Hamiltonian cycle). Then, we create a train segment from each origin node (that is, where some block starts) to the nearest yard. Next, we connect each destination node (that is, where some block terminates) to the nearest yard, which still has some spare arc emanating capacity. Since each origin is connected to some yard, and each destination is connected to some yard, and each yard connected to every other yard (through the Hamiltonian cycle), this construction process will ensure that each origin node is connected to each destination node. After a train segment network has been constructed, we route blocks along their shortest paths in the train segment network.

We may point that our algorithm to construct the initial solution is a fairly simple algorithm and does not do any optimization. We have found in our computational testing that the final solution obtained by our algorithm is fairly insensitive to the starting solution and improving the initial solution does not have much impact on the final solution. We therefore did not spend much effort in trying more sophisticated methods for constructing the initial solution.

6.4.2 Finding an Improved Neighbor

As described earlier, we define the neighborhood of a solution as all the solutions that can be obtained by changing the segments made at one node only and rerouting all shipments along their updated shortest paths. Our algorithm performs passes over the nodes (all origins and yards where segments are built) one by one by selecting a node (say, node i) and re-optimizes the segments made at that node assuming that the train segments at other nodes do not change. To re-optimize the segments made at node i, we have developed an algorithm, called *maximum saving algorithm*. The maximum saving algorithm performs the following steps:

- It removes the train segments already made at node *i* and reroutes the blocks passing through this node.

- It then builds the segments at node *i* one by one using the following *maximum savings rule*. For every potential train segment arc that can be built at node *i*, it computes the savings in the total cost if that train segment arc is built and all blocks are rerouted to take advantage of this new train segment arc, selects the train segment arc with the maximum savings, builds it and reroutes all the blocks for which costs would decrease by using the newly built train segment arc. We repeat this process until all the required train segments at this node are built.

We point out that it is necessary to perform several passes over the nodes since when we re-optimize the segments made at node *i*, we assume some segments at node *i*+1, *i*+2, …, *n*. But later when train segments at nodes *i*+1, *i*+2, …, *n* are re-optimized (and possibly changed), the segments made at node *i* may need to be changed in which case segments built at node *i* need to be re-optimized again. Thus, the algorithm performs passes over the nodes 1, 2, …, *n,* in order, in the train segment network and re-optimizes the train segments built at those nodes. The algorithm terminates when the solution converges, that is, in an entire pass the objective function value does not change significantly. We observed in our computational results that the solution converges within 10 passes over the nodes.

6.5 Phase II: Forming Trains

Form the phase I, we get a train segment network with the routings of blocks. In the train segment network, trains run on the segments with stops at the end points of train segment arc. For the test instance in our case, there are around 1100 segments created in phase I. As the train design problem is to design trains and their schedule, one straight forward solution for the train problem is to create train for each of the train segment. However, obviously this solution will not be viable economically and practically. This

option also means that there will be as many block swaps for a block as many train segments are there on its route. To make the solution of the train design problem more efficient, train segments are combined to create the trains. The benefits of combining trains segments are in terms of savings locomotives and crew requirements, and reduced number of block swaps. We call the problem of combining the train segments to form the trains as *Train Formation Problem* (TFP).

As discussed in previous section, we achieve an upper bound on the number of block swaps in the solution of the train design problem. In this phase our objective is to combine segments in such a way that there will be minimum number of blocks swaps. For example, in the train segment network given in Figure 6.1, let a block takes the train segment path *2-6-8-10*. Now if we create train for each segment separately, there will be two block swaps at stations *6* and *8*. Now if we combine the segments (*2*, *6*), (*6*, *8*) and (*8*, *10*) in a train, then there will be no block swap as this block will be carried by only one train from its origin to its destination. Now as a segment may carry more than one block and they may be having different train segment paths, the number of blocks depends on the way we combine the segments to form trains. For example, let there are three blocks (say k_1, k_2 and k_3), which are routed on paths *1-5-9*, *1-5-9*, and *1-5-8-10* respectively. Now if we decide to create the trains *1-5-9* (say T_1) and *5-8-10* (say T_2), then there will be one block swap, as block k_3 will be transferred from train T_1 to train T_2 at station 5. On the other hand, if we decide to create the trains *1-5-8-10* (say T_3) and *5-9* (say T_4), then there will be two block swaps as the blocks k_1 and k_2 will change train at station *5*. Therefore, the number of block swaps depends on the way train segments are

combined to form trains. One of objectives of the train formation problem is to minimize the number of block swaps.

Generally, a set of locomotives is assigned to a train, which carries it from its origin to its destination. To maximize the locomotive utilization, another important objective of the train design problem is to maintain uniformity of load in a train. For example, suppose we form a train consisting of segments *1-5* and *5-9*. Now suppose that there are 30 cars on the segment *1-5* and 60 cars on the segment *5-9* then the train must be assigned with locomotives which can pull 60 cars. However, this will result into underutilization of traction power on segment *1-5*. For the railroads, locomotives are very vital resource and proper utilization of them is very important.

The train formation problem can be defined as path covering problem in which we need to create minimum number of paths in the train segment network such that all the arcs are covered at minimum cost. The cost of creating a path consists of fixed charge for a train and inter-relation among train segments in the path. The inter-relation among train segments implies the common blocks flowing on these segments and car load on these segments. If there are more common blocks on the segments in a path, there is less number of block swaps as blocks are riding same train for a larger part of its itinerary. Similarly more uniformity of car load implies better utilization of locomotives attached to the train. In the train segment network, a valid train can also be formed by creating a directed cycle, which implies that a train starts from a station and terminates at the same stop after visiting other stations in the network. In this section we will denote such cycles as paths, which starts and terminates at same station.

We formulate the train formation problem as a minimum cost flow problem on a suitably constructed graph (say G'). In this graph, we capture the block swapping cost and penalty on the non-uniformity in load on a train as the arc cost. We create the network G' as follows (refer Figure 6.3).

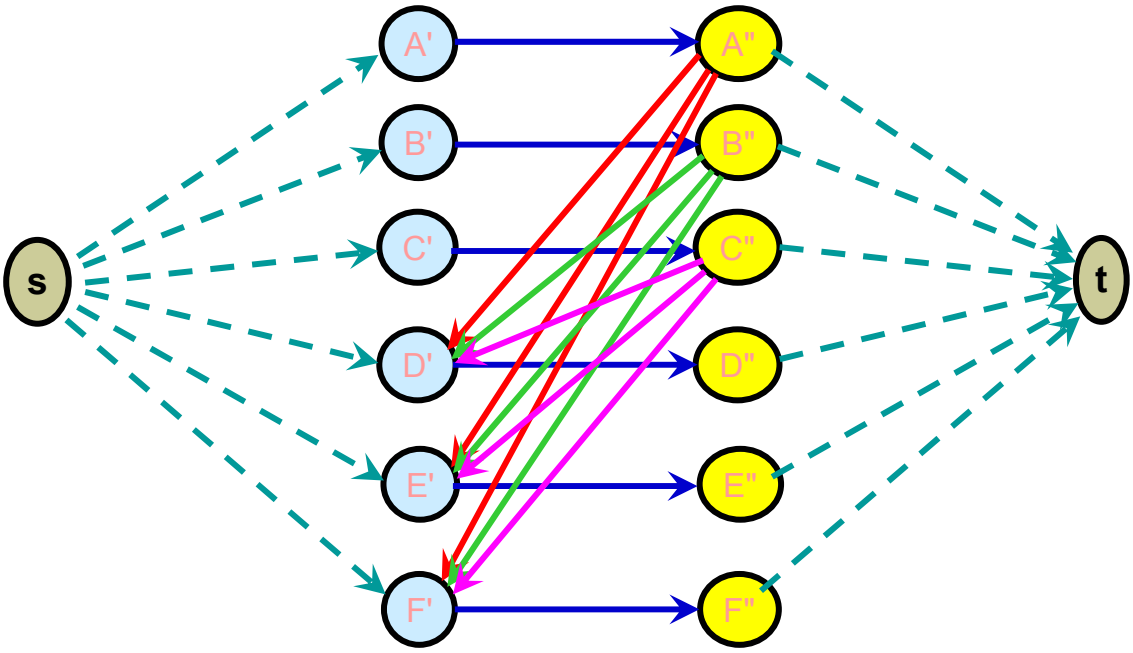


Figure 6.3 Graph for minimum cost flow formulation.

We create an arc for each of the train segment in Figure 6.1 and create nodes at the tail and head of these arcs. We call the set of nodes thus created as N' and set of arcs thus created as train arcs T'. Let there is an arc a' in G' corresponding to an arc a in train segment network. Let A_a^+ and A_a^- are the sets of arcs emanating from the head node of the arc a and entering into the tail node of arc a in the train segment network. Now, for each of the arc a' in G', we create arcs from the head of this arc to the tail node of all the arcs which represents the arcs in A_a^+ in the train segment network. We denote the set of all such arcs as connecting arcs C'. Let there are m arcs in train segment network, then as

there is a connecting arc for each pair of incoming and outgoing arcs at a node, there can be $O(m^2)$ connecting arcs in G'. Also, the number of connecting arcs increases with the density of the train segment network. However, in practice, the train segment network is very sparse and hence the number of connecting arcs is linear to the number of train arcs. Next, we create a dummy source node and a dummy sink node in the network G'. We create arc from source node to the tail of each of the train arc and similarly, create arc from the head of each of the train arc to the dummy sink node. We also create an arc from the dummy source node to the dummy sink node. Thus there will be $m + 1$ arcs from the source node and there will be $m + 1$ arcs to the sink nodes. If we add the number of different type of arcs, it will be of the $O(m^2)$. As we create two nodes in G' for every train arc, there will be $2m + 2$ nodes in the network G'. Figure 6.3 represents the graph G'.

We now set the capacity and cost of the arcs in G' and set the supply/demand at the nodes to formulate the train formation problem as minimum cost flow problem. We set the upper bound of 1 and lower bound of 0 on the arcs from source node to the tail nodes of train arcs. We also set 0 cost of flow on these arcs. The upper and lower bounds on the flow on train arcs are set to 1. This implies a compulsory unit flow on train arcs. The cost of flow on train arcs are also set to 0. We do not set any capacity limit on the flow on connecting arcs and in the following paragraph, we discuss the cost of flow on these arcs. Similar to the arcs from the source node, we set the upper bound and lower bound on flow to 1 and 0 respectively and set the cost of flow to zero. We do not put and capacity restriction on flow on arc between the source node and the sink node and set the cost of flow to 0. We put a supply of m units at the source node and put a demand of m units at the sink node.

Now, we describe the cost of flow on the connecting arcs in the network G'. Any flow on these arcs denotes that the train arcs at the tail node and at the head node are associated to the same train. Therefore, for every possibility of combining two train segments at a node, there is a connecting arc in the network G'. Let us assume that a_1' and a_2' are the respective train arcs at the tail node and head node of a connecting arc. Any block which is flowing on arc a_1 and not flowing on a_2 and not terminating at the head node of the arc a_1 in the train segment network must go for block swap as it needs to take some another train for onward itinerary. As we started with an upper bound on number of block swaps from Phase I, we calculate the savings in block swaps if segments a_1 and a_2 are combined into same train and associate with the corresponding connecting arc in G'. It can be noted that this cost is independent of the other train segments in a train as a block swap affects only at the connecting terminal of two segments. Similarly, we calculate the penalty in terms of difference in number of cars in segments a_1 and a_2 and add to the cost of flow on corresponding connecting arc. In case of practical implementation, railroad may also specify their likings for combining certain pair of segments. This may be required to improve the customer satisfaction.

Theorem: There exists a one-to-one correspondence between the solution of the train formation problem and the solution of minimum cost flow problem defined on network G'.

Proof: Any solution of the minimum cost flow problem can be represented by the flow along the paths from the source node to the sink nodes and by the flow along the cycles in the network G'. We claim that each of these paths and cycles represent a valid train and all the train segments in the train segment network are covered by a train in G'.

As there must be positive flow on each train arc, a train arc must have been covered by some path or cycle and as the flow is of unit quantity, the train arcs must be covered by single path or cycle. Now in any path or cycle in the solution, there must be connecting arc between two consecutive train arcs, which implies that there must be common station, which connects two segments. Thus every path and cycle with positive flow represents a valid train for the train formation problem. In case of paths, the origins and destinations of the trains are given by the tail node of the first train arc and head node of the last train arc respectively. In case of cyclic flow, any node on the cycle can be made as the origin and destination of the train. ♦

We have incorporated the objectives of the train formation problem into the flow cost on connecting arcs in network G', the optimal solution of the minimum cost flow formulation gives the optimal solution of the train formation problem. The minimum cost flow problem can be solved very efficiently to optimality i.e. we can get the optimal solution of the train formation very efficiently.

6.6 Computational Results

We implemented the Phase I and Phase II of our algorithm on a real life instance of a major US railroad. The railroad is engaged in carrying around 50,000 shipments per day. They build around 1,300 blocks per day to carry these shipments. The physical network of railroad consists of around 6,000 stations and 12,000 links. We implemented the algorithms on Visual Studio .Net platform using C++ programming language. The testing of the algorithms is done on a PC with 1 GB RAM and 3 GHz of processor speed.

In Phase I, we need the input on the number of train segments that can be created at a station. We get this input from the existing train schedule of the railroad. We

considered only those stops of a train where block swapping is permissible. To minimize the number of train segments, we put a penalty on the creation of an arc in the train segment network. Our algorithm produces a solution with 1,116 train segments, while honoring the constraints on the number of train segments. The computational time required to produce the solution is within 5 seconds. When compared with the existing train schedule, there is a major saving in the number of train segments, which is around 2,000.

We take the solution of Phase I as the input to Phase II. We created a graph to formulate the problem in this phase as minimum cost flow problem. The graph in this phase consists of 10,456 arcs and 2,234 nodes. It can be noted that the number of arcs is much less than the upper bound of O(m^2 = 1.2 million). We used CPLEX optimizer to solve the minimum cost flow problem. The computational time taken to obtain optimal solution is around 5 seconds and produces a solution with 325 trains. Out of these, 66 trains starts and ends at same station. In the actual train schedule, there are 339 train schedules to carry same shipments. Therefore, our solution gives a saving of 14 trains, which may result in substantial economical saving for the railroad.

6.7 Conclusion and Further Scope

In this chapter we have tried a decomposition approach to solve the train design problem. Our solution approach consists of three phases. We have developed algorithms to solve the problems in Phase I and Phase II. Further work with the coordination of a railroad is required to be done to develop algorithms for Phase III.

CHAPTER 7
FURTHER RESEARCH AND CONCLUDING REMARKS

In this dissertation, we have tried to solve few hard combinatorial optimization problems using very large-scale neighborhood (VLSN) search technique. Our effort is to use efficient network flow techniques in developing heuristics to solve these problems. In this chapter, we will first summarize the contribution and then will present the scope of future research.

7.1 Contribution Summary

Prior to our application of VLSN search heuristic to the quadratic assignment problem, this technique was only applied to partitioning problems. In chapter 2, we present the application of VLSN search technique to the quadratic assignment problem, which is a non-partitioning optimization problem. We have developed an improvement graph based approach, in which we approximate the cost of multi-exchange with the cost of a cycle in the improvement graph. We have also developed an efficient mechanism to update the improvement graph, whenever, we make some change in the current solution of the problem. We performed extensive computational experiment on the problems available in QAPLIB and our algorithm performs well all across the test instances.

In chapter 3, we studied a classical and important defense related optimization problem, called the weapon-target assignment problem. This problem is to assign given set of weapons to given targets such that loss to the enemy can be maximized. This is a

non-linear optimization problem and previous efforts are limited to solve only small size problems. In this dissertation, we suggest linear programming, integer programming, and network flow based lower bounding methods using which we obtain several branch and bound algorithms for the WTA problem. We have developed a network flow based heuristic and a VLSN search heuristic. The combination of network flow based heuristic and VLSN search produces exceptionally good quality solution in very manageable computational time. We have been bale to solve the problems of the size, which were historically intractable.

In chapter 4 of the dissertation we have developed algorithms to solve another combinatorial optimization problem called the university course-timetabling problem. In this problem we fix the meeting time of courses in a university with the objectives of maximizing the teachers' preferences and minimizing the conflict in students' interests. Most of the literatures on this problem are case specific, where algorithms are developed to solve a given test instance. We have tried to develop a generic model for the course-timetabling problem. We have developed algorithms in conjunction with tabu search, which performs very well on the randomly generated test instances.

Chapter 5 and Chapter 6 of this dissertation deal with two real-life railroad scheduling problems. In chapter 5, we have developed algorithms to solve the block-to-train assignment problem. This problem is to find train path for a block such that transportation cost and transit time are minimum, and there is no capacity violation. We formulate this problem as an integer program on a space-time network. We have developed very large-scale neighborhood search based heuristics to enumerate the best train paths for each block. We have developed numerous approaches to solve this

problem, when possible train paths are given for each block. In chapter 6, we study a broader railroad-scheduling problem called the train schedule design problem. This problem is to design trains and their schedules such that overall transportation cost can be minimized. This is very large scale and complex optimization problem. Enormous efforts have been made in the past to solve this problem. We have developed a decomposition based solution approach, in which the train schedule design problem is solved sequentially in phases. We have developed a VLSN search heuristic to solve the problem in Phase I and have formulated the problem in Phase II as minimum cost flow problem on a suitably constructed graph. Our algorithms produce very good quality solution when tested on a real-life instance.

7.2 Future Scope

Our algorithm to solve the quadratic assignment problem is an enhanced version of local search heuristic. We expect that when applied in conjunction with other meta-heuristic, the VLSN search algorithm may perform better. Further work is required to be done to develop algorithms in this regard. In chapter 3, we have considered the static version of the weapon-target assignment problem. However, in real-life application the problem is of dynamic nature. Our algorithms in chapter 3 can be easily modified to solve the dynamic version of the problem. We expect further work to modify our algorithms to accommodate dynamic nature of the problem. With respect to the course-timetabling problem, further work can be done in developing better algorithms to generate random test instances and to incorporate any additional constraint in the VLSN search algorithms. In chapter 5, we have considered daily version of the block-to-train assignment problem. In this case, we assume that a block is made everyday and a train runs each day of the

week. However, in general train schedules are on weekly basis and a train may not run everyday. Further work is required to develop proper post-processing routines to extend our approach to the seven day block-to-train assignment problem. In chapter 6, the future work is required to be done to develop the algorithm in coordination with railroads to solve the problem in Phase III. We also expect a thorough testing of algorithms developed by experimenting with the real-life data of more than one railroad.

APPENDIX: COMPUTATIONAL EXPERIMENT ON QAP INSTANCES

Table A.1 Experimental Results for Symmetric Instances

				2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
SN	**Name**	**n**	**BKS**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**
1	Chr12a	12	9552	0	45.76	9,436,220	0.89	0	31.21	2,824,741	2.34	0	28.98	3,807,890	1.45
2	Chr12b	12	9742	0	52.78	8,402,658	6.42	0	45.37	3,002,710	8.26	0	44.38	4,572,662	8.64
3	Chr12c	12	11156	0	34.92	9,823,150	0.32	0	24.95	2,957,361	0.76	0	23.82	3,907,786	0.52
4	Chr15a	15	9896	0	50.10	4,704,652	0.07	0	34.06	1,420,416	0.38	0	32.43	2,021,011	0.41
5	Chr15b	15	7990	0	60.11	4,468,591	0.17	0	47.09	1,537,817	0.25	0	46.64	2,266,858	0.37
6	Chr15c	15	9504	0	61.61	4,920,512	0.04	0	43.86	1,395,482	0.09	0	41.79	2,015,713	0.19
7	Chr18a	18	11098	0	66.83	2,730,190	0.01	0	50.01	779,653	0.17	0	49.76	1,285,044	0.06
8	Chr18b	18	1534	0	14.91	3,061,778	0.57	0	8.53	749,002	2.84	0	14.34	2,699,902	0.92
9	Chr20a	20	2192	0	47.81	2,102,113	0.00	0	33.83	531,198	0.02	0	43.11	1,537,470	0.00
10	Chr20b	20	2298	0	41.48	2,248,147	0.00	0	27.59	545,328	0.00	0	33.18	1,304,395	0.00
11	Chr20c	20	14142	0	83.48	1,638,126	0.08	0	63.75	537,304	0.20	0	68.17	1,004,525	0.23
12	Chr22a	22	6156	0	13.75	1,409,341	0.00	0	10.03	321,233	0.02	0	12.73	1,094,969	0.00
13	Chr22b	22	6194	0.581	13.98	1,524,784	0.00	0	9.55	306,967	0.00	0	11.86	935,420	0.00
14	Chr25a	25	3796	0	57.88	921,946	0.00	0	44.08	261,426	0.00	0	52.94	700,777	0.00
15	Els19	19	17212548	0	25.63	1,703,953	1.91	0	14.47	298,488	23.78	0	24.86	1,501,674	1.91
16	Esc16a	16	68	0	3.54	6,915,006	35.00	0	0.36	1,187,393	94.63	0	3.54	6,849,543	35.00
17	Esc16b	16	292	0	0.01	9,060,254	98.44	0	0.00	1,063,243	100.00	0	0.01	8,988,691	98.44
18	Esc16c	16	160	0	1.15	5,753,649	53.58	0	0.25	1,140,441	84.19	0	1.15	5,702,568	53.58
19	Esc16d	16	16	0	9.18	7,089,066	43.13	0	0.72	1,165,768	94.29	0	9.18	7,026,224	43.13
20	Esc16e	16	28	0	10.12	8,572,391	20.17	0	3.25	1,165,223	60.37	0	10.12	8,502,350	20.17
21	Esc16f	16	0	0	0.00	35,611,664	100.00	0	0.00	4,918,942	100.00	0	0.00	35,420,396	100.00
22	Esc16g	16	26	0	7.86	7,348,488	44.39	0	0.23	1,124,152	97.03	0	7.86	7,287,152	44.39
23	Esc16h	16	996	0	0.00	7,922,830	100.00	0	0.00	1,051,718	100.00	0	0.00	7,857,457	100.00
24	Esc16i	16	14	0	1.09	8,107,025	96.53	0	0.00	1,712,511	100.00	0	1.09	8,041,288	96.53
25	Esc16j	16	8	0	15.49	9,128,130	54.33	0	1.28	940,748	94.90	0	15.47	9,037,786	54.34
26	Esc32a	32	130	0	23.16	591,484	0.00	0	13.48	105,197	0.13	0	23.16	589,319	0.00
27	Esc32b	32	168	0	28.83	573,480	0.49	0	15.60	119,828	3.89	0	28.83	571,874	0.49
28	Esc32c	32	642	0	0.40	806,970	82.10	0	0.01	116,680	99.87	0	0.40	804,161	82.10
29	Esc32d	32	200	0	6.42	789,986	4.49	0	3.57	115,082	21.64	0	6.42	787,713	4.49
30	Esc32e	32	2	0	0.00	1,862,801	100.00	0	0.00	118,660	100.00	0	0.00	1,858,335	100.00
31	Esc32f	32	2	0	0.00	1,861,132	100.00	0	0.00	118,654	100.00	0	0.00	1,858,965	100.00
32	Esc32g	32	6	0	0.64	1,868,071	98.08	0	0.00	116,371	100.00	0	0.64	1,865,820	98.08

Contd ...

Table A.1 (contd.): Experimental Results for Symmetric Instances

				2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
SN	**Name**	**n**	**BKS**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**
33	Esc32h	32	438	0	3.93	732,021	1.55	0	2.00	82,942	9.08	0	3.93	728,741	1.55
34	Esc64a	64	116	0	1.48	302,102	48.88	0	0.14	21,571	95.69	0	1.48	301,986	48.88
35	Esc128	128	64	0	13.10	38,994	8.27	0	5.61	1,850	33.89	0	13.09	39,101	8.26
36	Had12	12	1652	0	1.31	8,903,617	4.97	0	0.72	1,571,863	18.01	0	1.20	6,737,857	6.69
37	Had14	14	2724	0	0.87	4,681,890	13.68	0	0.48	979,815	32.92	0	0.82	3,890,031	14.49
38	Had16	16	3720	0	0.90	3,046,483	13.75	0	0.48	674,610	26.19	0	0.87	2,658,712	13.81
39	Had18	18	5358	0	1.10	2,224,949	2.66	0	0.78	407,788	5.32	0	1.07	2,016,232	2.88
40	Had20	20	6922	0	1.19	1,562,491	2.27	0	0.84	289,189	6.76	0	1.17	1,435,751	2.29
41	Kra30a	30	88900	0	7.70	508,286	0.02	0	5.99	82,846	0.20	0	7.70	507,363	0.02
42	Kra30b	30	91420	0	5.76	505,339	0.00	0	4.13	83,349	0.04	0	5.76	504,516	0.00
43	Nug12	12	578	0	5.29	10,480,616	1.35	0	3.48	2,387,442	6.50	0	4.85	6,507,051	2.36
44	Nug14	14	1014	0	5.02	6,045,903	0.44	0	3.42	1,096,624	0.95	0	4.70	3,920,230	0.86
45	Nug15	15	1150	0	4.46	4,738,236	1.46	0	2.79	1,005,809	4.94	0	4.02	3,224,136	1.97
46	Nug16a	16	1610	0	4.86	3,835,499	0.28	0	3.43	698,454	1.61	0	4.56	2,729,922	0.53
47	Nug16b	16	1240	0	5.40	3,854,426	3.12	0	3.64	756,449	6.98	0	4.93	2,615,318	4.06
48	Nug17	17	1732	0	4.24	3,103,273	0.08	0	2.73	553,263	1.08	0	4.01	2,283,745	0.14
49	Nug18	18	1930	0	4.44	2,633,371	0.17	0	3.09	467,170	0.85	0	4.15	1,876,454	0.25
50	Nug20	20	2570	0	4.19	1,856,594	0.22	0	3.04	342,835	0.67	0	3.97	1,386,350	0.26
51	Nug21	21	2438	0	4.58	1,481,678	0.11	0	3.14	283,327	0.29	0	4.35	1,188,565	0.14
52	Nug22	22	3596	0	3.72	1,195,612	0.50	0	2.71	244,287	1.37	0	3.59	1,015,230	0.53
53	Nug24	24	3488	0	4.67	967,896	0.10	0	3.31	181,818	0.70	0	4.45	787,586	0.26
54	Nug25	25	3744	0	3.90	842,354	0.05	0	2.64	160,132	0.42	0	3.80	726,571	0.06
55	Nug27	27	5234	0	4.33	626,472	0.04	0	3.27	115,644	0.40	0	4.16	527,571	0.11
56	Nug28	28	5166	0	4.51	579,278	0.01	0	3.30	100,579	0.19	0	4.32	481,522	0.02
57	Nug30	30	6124	0	4.19	453,552	0.01	0	3.06	86,179	0.03	0	4.10	398,408	0.01
58	Rou12	12	235528	0	5.64	10,263,930	0.69	0	4.25	2,348,174	1.83	0	4.65	4,692,485	1.09
59	Rou15	15	354210	0	6.90	5,230,145	0.37	0	5.26	1,081,279	0.93	0	6.06	2,631,306	0.57
60	Rou20	20	725522	0	4.86	2,114,540	0.01	0	3.34	344,848	0.02	0	4.36	1,256,472	0.01
61	Scr12	12	31410	0	7.03	9,411,221	5.05	0	4.67	2,337,375	11.82	0	4.82	4,371,642	16.64
62	Scr15	15	51140	0	10.24	4,273,122	1.76	0	7.66	1,070,211	6.80	0	7.65	2,114,390	5.51
63	Scr20	20	110030	0	9.86	1,709,033	0.06	0	6.22	378,005	0.21	0	6.52	896,590	0.18
64	Sko42	42	15812	0.101	3.53	294,068	0.00	0	2.73	53,925	0.01	0.101	3.48	270,262	0.00
65	Sko49	49	23386	0.162	3.08	176,442	0.00	0.154	2.44	33,002	0.00	0.162	3.03	165,944	0.00
66	Sko56	56	34458	0.430	2.96	108,971	0.00	0.163	2.39	20,623	0.00	0.261	2.92	103,033	0.00
67	Sko64	64	48498	0.421	2.70	68,901	0.00	0.293	2.20	12,162	0.00	0.400	2.68	66,537	0.00
68	Sko72	72	66256	0.420	2.65	45,817	0.00	0.546	2.21	7,832	0.00	0.420	2.64	44,570	0.00
69	Sko81	81	90998	0.510	2.27	31,087	0.00	0.440	1.91	5,273	0.00	0.510	2.26	30,521	0.00

Contd ...

Table A.1 (contd.): Experimental Results for Symmetric Instances

				2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
SN	**Name**	**n**	**BKS**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**
70	Sko90	90	115534	0.535	2.23	21,473	0.00	0.460	1.89	3,425	0.00	0.535	2.22	21,126	0.00
71	Sko100a	100	152002	0.617	2.08	14,955	0.00	0.529	1.74	2,180	0.00	0.617	2.07	14,750	0.00
72	Sko100b	100	153890	0.516	2.02	15,110	0.00	0.377	1.71	2,222	0.00	0.516	2.02	14,908	0.00
73	Sko100c	100	147862	0.580	2.30	14,843	0.00	0.511	1.96	2,182	0.00	0.580	2.29	14,633	0.00
74	Sko100d	100	149576	0.646	2.07	15,170	0.00	0.513	1.74	2,179	0.00	0.646	2.07	15,038	0.00
75	Sko100e	100	149150	0.581	2.31	14,820	0.00	0.457	1.94	2,187	0.00	0.581	2.30	14,615	0.00
76	Sko100f	100	149036	0.780	2.03	15,299	0.00	0.573	1.71	2,215	0.00	0.666	2.02	15,083	0.00
77	Ste36a	36	9526	0.252	12.02	238,239	0.00	0	9.07	44,939	0.01	0.252	12.02	239,827	0.00
78	Ste36b	36	15852	0	21.46	213,196	0.01	0	15.95	47,338	0.18	0	21.46	214,163	0.01
79	Ste36c	36	8239.11	0.132	9.46	226,567	0.00	0	7.24	45,428	0.00	0.132	9.46	227,512	0.00
80	Tai12a	12	224416	0	8.93	10,073,866	2.12	0	6.88	1,715,403	4.79	0	7.86	4,651,037	3.09
81	Tai15a	15	388214	0	4.81	5,387,038	0.12	0	3.41	986,214	0.40	0	4.07	2,661,777	0.15
82	Tai17a	17	491812	0	5.69	3,620,256	0.05	0	4.22	627,478	0.21	0	4.89	1,846,451	0.17
83	Tai20a	20	703482	0	6.03	2,214,371	0.00	0	4.37	340,557	0.02	0	5.11	1,147,683	0.01
84	Tai25a	25	1167256	0	5.57	1,109,310	0.00	0	4.08	173,744	0.00	0	4.76	609,159	0.00
85	Tai30a	30	1818146	0.456	5.06	615,703	0.00	0.532	3.84	78,638	0.00	0	4.47	371,344	0.00
86	Tai35a	35	2422002	1.083	5.10	387,154	0.00	0.687	3.72	46,236	0.00	1.075	4.47	237,949	0.00
87	Tai40a	40	3139370	1.358	5.05	254,304	0.00	0.906	3.68	28,821	0.00	1.357	4.56	169,185	0.00
88	Tai50a	50	4941410	1.897	4.97	253,480	0.00	1.437	3.71	26,559	0.00	1.863	4.46	170,079	0.00
89	Tai60a	60	7208572	2.177	4.72	138,643	0.00	1.658	3.54	13,200	0.00	1.902	4.20	91,441	0.00
90	Tai64c	64	1855928	0	0.44	416,518	6.09	0	0.42	27,985	6.19	0	0.44	418,391	6.09
91	Tai80a	80	13557864	2.037	3.76	56,268	0.00	1.558	2.78	4,484	0.00	1.713	3.22	36,122	0.00
92	Tai100a	100	21125314	1.969	3.42	27,534	0.00	1.608	2.49	1,789	0.00	1.707	3.03	19,184	0.00
93	Tai256c	256	44759294	0.175	0.43	3,526	0.00	0.175	0.41	168	0.00	0.175	0.43	3,392	0.00
94	Tho30	30	149936	0	4.85	432,914	0.01	0	3.72	86,314	0.03	0	4.60	336,287	0.01
95	Tho40	40	240516	0.341	4.67	174,638	0.00	0.091	3.70	35,056	0.00	0.043	4.46	142,400	0.00
96	Tho150	150	8133484	0.877	2.42	3,578	0.00	0.814	2.12	469	0.00	0.900	2.39	2,589	0.00
97	Wil50	50	48816	0.213	1.66	18,331	0.00	0.086	1.37	30,101	0.00	0.123	1.65	149,664	0.00
98	Wil100	100	273038	0.363	1.10	15,005	0.00	0.347	0.95	2,031	0.00	0.363	1.10	14,921	0.00

Table A.2 Experimental Results for Asymmetric Instances

				2OPT				Implementation 3 (Best αn^2 paths)				Implementation 4 (Best n paths)			
SN	**Name**	**n**	**BKS**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**	**BestGap**	**AvgGap**	**nRuns**	**%Best**
1	bur26a	26	5426670	0	0.32	114,672	0.32	0	0.24	85,634	1.11	0	0.32	114,606	0.32
2	bur26b	26	3817852	0	0.41	122,308	0.29	0	0.30	87,293	0.58	0	0.41	122,247	0.29
3	bur26c	26	5426795	0	0.41	112,625	0.31	0	0.26	82,912	1.05	0	0.41	112,556	0.31
4	bur26d	26	3821225	0	0.47	119,809	0.28	0	0.29	87,760	0.54	0	0.47	119,743	0.28
5	bur26e	26	5386879	0	0.37	110,953	0.39	0	0.20	83,787	2.65	0	0.37	110,890	0.39
6	bur26f	26	3782044	0	0.46	119,957	0.66	0	0.24	87,391	3.07	0	0.46	119,891	0.66
7	bur26g	26	10117172	0	0.36	109,920	0.47	0	0.25	83,347	2.44	0	0.36	109,829	0.47
8	bur26h	26	7098658	0	0.46	116,744	0.95	0	0.35	93,540	4.50	0	0.46	116,677	0.95
9	lipa20a	20	3683	0	2.84	386,856	0.37	0	2.52	238,081	1.37	0	2.80	285,714	0.50
10	lipa20b	20	27076	0	15.22	364,584	4.27	0	12.66	277,407	12.94	0	14.80	233,970	4.96
11	lipa30a	30	13178	0	2.02	104,987	0.03	0	1.83	54,075	0.24	0	2.02	104,939	0.03
12	lipa30b	30	151426	0	16.79	102,598	1.88	0	14.96	56,856	7.34	0	15.94	61,550	4.03
13	lipa40a	40	31538	0.907	1.52	43,018	0.00	0	1.36	18,082	0.01	0.907	1.51	36,566	0.00
14	lipa40b	40	476581	0	18.64	41,186	1.26	0	16.90	19,034	5.80	0	18.23	29,478	2.03
15	lipa50a	50	62093	0.892	1.31	42,978	0.00	0.849	1.17	15,372	0.00	0.892	1.30	37,043	0.00
16	lipa50b	50	1210244	0	18.67	41,633	0.46	0	17.52	16,644	2.33	0	18.40	30,964	0.70
17	lipa60a	60	107218	0.844	1.11	24,248	0.00	0.787	0.99	7,166	0.00	0.844	1.10	21,339	0.00
18	lipa60b	60	2520135	0	20.08	24,046	0.09	0	19.22	7,938	0.42	0	19.71	16,725	0.22
19	lipa70a	70	169755	0.775	0.96	15,016	0.00	0.725	0.86	4,056	0.00	0.773	0.95	13,187	0.00
20	lipa70b	70	4603200	0	20.80	14,644	0.05	0	19.94	4,358	0.53	0	20.37	9,767	0.17
21	lipa80a	80	253195	0.678	0.85	10,026	0.00	0.628	0.75	2,335	0.00	0.678	0.83	8,510	0.00
22	lipa80b	80	7763962	0	21.69	9,701	0.03	0	20.92	2,451	0.08	0	21.23	6,214	0.06
23	lipa90a	90	360630	0.636	0.78	6,836	0.00	0.581	0.69	1,412	0.00	0.636	0.77	6,231	0.00
24	tai12b	12	39464925	0	11.25	1,619,258	3.03	0	8.94	1,682,704	12.55	0	10.35	953,511	3.56
25	tai20b	20	122455319	0	17.45	266,289	0.60	0	14.22	207,118	6.30	0	15.13	183,113	0.88
26	tai25b	25	344355646	0	16.24	121,862	0.03	0	12.10	83,619	0.59	0	15.53	89,529	0.08
27	tai30b	30	637117113	0	13.72	65,700	0.00	0	9.06	38,047	0.13	0	12.65	28,945	0.00
28	tai35b	35	283315445	0	8.72	40,095	0.00	0	6.47	23,213	0.03	0	8.12	31,932	0.01
29	tai40b	40	637250948	0	10.59	24,696	0.00	0	8.29	13,527	0.24	0	10.18	19,742	0.01
30	tai50b	50	458821517	0.073	7.24	23,734	0.00	0.058	5.73	11,315	0.00	0.073	7.10	20,253	0.00
31	tai60b	60	608215054	0.008	7.99	12,713	0.00	0.038	6.16	4,863	0.00	0.089	7.75	10,853	0.00
32	tai80b	80	818415043	1.594	6.11	4,988	0.00	0.844	5.27	1,597	0.00	1.594	6.01	4,563	0.00
33	tai100b	100	1185996137	0.875	5.31	2,298	0.00	0.653	4.43	563	0.00	0.762	5.20	2,116	0.00
34	tai150b	150	498896643	1.473	3.49	624	0.00	1.467	3.10	92	0.00	1.473	3.44	584	0.00

LIST OF REFERENCES

Aarts, E. H. L., and J. Korst. 1989. *Simulated Annealing and Boltzman Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, Chichester.

Aarts, E., and J.K. Lenstra. 1997. *Local Search in Combinatorial Optimization*. John Wiley, New York.

Ahuja, R.K., O. Ergun, J.B. Orlin, and A.P. Punnen. 2002a. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics* **123**, 75-102.

Ahuja, R.K., J. Liu, J.B. Orlin, D. Sharma, and L.A. Shughart. 2002b. Solving real-life locomotive scheduling problems. Submitted to *Transportation Science*.

Ahuja, R.K., J. Liu, J. Goodstein, A. Mukherjee, J.B. Orlin, and D. Sharma. 2003b. Solving multi-criteria combined through-fleet assignment models. In the *Operations Research in Space and Air*, Edited by T. A. Ciriani, G. Fasano, S. Gliozzi, and R. Tadei, Kluwer Academic Publishers, Boston pp. 233-256.

Ahuja, R.K., J. Liu, J. Goodstein, A. Mukherjee, and J.B. Orlin. 2003c. A neighborhood search algorithm for the combined through and fleet assignment model with time windows. Submitted to *Networks*.

Ahuja, R.K., T.L. Magnanti, J.B. Orlin. 1993. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Upper Saddle River, New Jersey.

Ahuja, R.K., J.B. Orlin, S. Pallottino, M.P. Scaparra, and M.G. Scutella. 2002c. A multi-exchange heuristic for the single source capacitated facility location. Submitted to *Management Science*.

Ahuja, R. K., J. B. Orlin, D. Sharma. 2001a. Multi-exchange neighborhood search algorithms for the capacitated minimum spanning tree problem. *Mathematical Programming* **91**, 71-97.

Ahuja, R. K., J. B. Orlin, D. Sharma. 2001b. A very large-scale neighborhood search algorithm for the combined through-fleet assignment model. Submitted to *INFORMS Journal on Computing*.

Ahuja, R.K., J.B. Orlin, and D. Sharma. 2003a. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters* **31**, 185-194.

Ahuja, R. K., J. B. Orlin, and A. Tiwari. 2000. A greedy genetic algorithm for the quadratic assignment problem. *Computers and Operations Research* **27**, 917-934.

Armour, G. C., and E. S. Buffa. 1963. Heuristic algorithm and simulation approach to relative location of facilities. *Management Science* **9**, 294-309.

Assad, A.A. 1980. Models for rail transportation. *Transportation Research* **A14**, 205-220.

Assad, A.A. 1982. A class of train-scheduling problems. *Transportation Science* **16**. 281-310.

Barnhart, C., H. Jin, and P.H. Vance. 2000. Railroad blocking: A network design application. Operations Research 48, 603-614.

Bazara, M. S., and M. D. Sherali. 1980. Benders' partitioning scheme applied to a new formulation of the quadratic assignment problem. *Naval Research Logistics Quarterly* **27**, 29-41.

Bean, J. C. 1994. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing* **6**, 154-160.

Bodin, L.D., B.L. Golden, A. Schuster, and W. Romeg. 1980. A model for the blocking of trains. *Transportation Research* **14B**, 115-120.

Braford, J.C. 1961. Determination of optimal assignment of a weapon system to several targets. AER-EITM-9, Vought Aeronautics, Dallas, Texas.

Brannlund U., P.O. Lindberg, A. Nou, and J.E. Nilson. 1998. railway timetabling using Lagrangian relaxation . *Transportation Science* **32**, 358-369.

Buffa, E. S., G. C. Armour, and T. E. Vollmann. 1964. Allocating facilities with CRAFT. *Harvard Business Review* **42**, 136-158.

Burkard, R. E. 1991. Location with spatial interactions: The quadratic assignment problem. *Discrete Location Theory,* Edited by P. B. Mirchandani and R. L. Francis, John Wiley.

Burkard, R. E., and T. Bonniger. 1983. A heuristic for quadratic boolean programs with applications to quadratic assignment problems. *European Journal of Operations Research* **13**, 374-386.

Burkard, R. E., E. Cela, P. M. Pardalos, and L. S. Pitsoulis. 1998. The quadratic assignment problem. In *Handbook of Combinatorial Optimization* **3**, Edited by D. Z. Zhu and P. M. Pardalos, Kluwer, Boston, pp. 241-337.

Burkard, R. E., S. E. Karisch, and F. Rendl. 1997. QAPLIB - A quadratic assignment program library. *Journal of Global Optimization* **10**, 391-403.

Campbell K.C. 1996. Booking and revenue management for rail intermodal services. PhD Dissertation. Department of Systems Engineering, University of Pennsylvania, Philadelphia, PA.

Carter M. W., and Laporte G. 1998. Recent Developments in Practical Course Timetabling. *PATAT'97, LNCS* **1408**, 3-19

Castanon D.A. 1987. Advanced weapon-target assignment algorithm quarterly report. TR-337, ALPHA TECH Inc., Burlington, MA.

Cela, E. 1998. *The Quadratic Assignment Problem – Theory and Algorithms*, Kluwer Academic Publishers, Boston.

Chang S.C., R.M. James, and J.J. Shaw. 1987. Assignment algorithm for kinetic energy weapons in boost defense. *Proceedings of the IEEE 26th Conference on Decision and Control*, Los Angeles, CA.

Christofides, N., and E. Benavent. 1989. An exact algorithm for the quadratic assignment problem. *Operations Research* **37**, 760-768.

Cordeau J.F., P. Toth, and D. Vigo. 1998. A survey of optimization models for train routing and scheduling. *Transportation Science* **32**, 380-404.

Crainic T.G., J.A. Ferland, and J.M. Rousseau. 1984. A tactical planning model for rail freight transportation. *Transportation Science* **18**, 165-184.

Crainic, T.G., and J.M. Rousseau. 1986. Multicommodity, miultimode freight transportation: A general modeling and algorithmic framework for the service network design problem. *Transportation Research* **208**. 225-242.

Croes, G.A. 1958. A method for solving traveling salesman problems. *Operations Research* **6**, 791-812.

Davis. L. 1991. *Handbook of Genetic Algorithms*. Van Nostrand, New York.

Day, R.H. 1966. Allocating weapons to target complexes by means of nonlinear programming. *Operations Research* **14,** 992-1013.

Deineko V. G., and G.J. Woeginger. 2000. A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem. *Mathematical Programming* **87(3),** 519-542.

DenBroader G. G., Jr., R. E. Ellison, and L. Emerling. 1958. On optimum target assignments. *Operations Research* **7,** 322-326.

Dimopoulou M., Miliotis P. 2001. Theory and methodology – implementation of a university course and examination timetabling system. *European Journal of Operational Research* **130**, 202-513.

Drezner, Z. 2003. A new genetic algorithm for the quadratic assignment problem. *INFORMS Journal on Computing* **15(3)**, 320-330.

Eckler, A.R., and S.A. Burr. 1972. Mathematical models of target coverage and missile allocation. Military Operations Research Society, Alexandria, VA.

Farvolden J.M., and W.B. Powell. 1994. Subgradient methods for the service network design problem. *Transportation Science* **28**, 256-272.

Fisher M.L. 1981. The Lagrangian relaxation method for solving integer programming problems. *Management Science* **27**, 1-18.

Fleurent, C., and J. A. Ferland. 1994. Genetic hybrids for the quadratic assignment problem. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* American Mathematical Society **16**, 173-187.

Glover, F. 1986. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research* **13**, 533-549.

Glover, F., and M. Laguna. 1997. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA.

Gorman, M.F. 1998. The freight railroad operating plan problem. *Annals of Operations Research* **78**, 51-69.

Grant, K.E. 1993. Optimal resource allocation using genetic algorithms. 1993. *Naval Review*, Naval Research Laboratory, Washington, DC, pp. 174-175.

Green D.J., J.T. Moore, and J.J. Borsi. 1997 An integer solution heuristic for the arsenal exchange model (AEM). *Military Operations Research Society,* **3(2)**, 5-15.

Haghani A.E. 1989. Formulation and solution of a combined train routing and makeup, and empty car distribution model. *Transportation Research* **23B**, 433-452.

Hertz A. 1991. Tabu Search for large scale timetabling problems. *European Journal of Operational Research* **54**, 39-47.

Holland, J.H. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI.

Ibaraki, T. 1987. *Enumerative Approaches to Combinatorial Optimization*. Baltzer, Basel.

Katter, J.D. 1986. A solution of the multi-weapon, multi-target assignment problem. Working paper 26957, MITRE.

Keaton M.H. 1989. Designing optimal railroad operating plans: Lagrangian relaxation and heuristic approaches. *Transportation Research* **23B**, 415-431.

Keaton, M.H. 1991. Service-cost tradeoffs for carload freight traffic in the U.S. rail industry. *Transportation Research* **25A**, 363-374.

Keaton, M.H. 1992a. Designing optimal railroad operating plans: A dual adjustment method for implementing Lagrangian relaxation. *Transportation Science* **26**, 262-279.

Keaton, M.H. 1992b. The impact of train timetables on average car time in rail classification yards. *Journal of the Transportation Research Forum* **32**, 345-354.

Kiaer L., and Yellen J. 1992. Weighted graphs and university course timetabling. *Computers and Operations Research* **19(1)**, 59-67.

Kirkpatrick, S. C.D. Gellatt Jr., and M.P. Vecchi. 1983. Optimization by simulated annealing. *Science* **220**, 671-680.

Kraft E.R. 1998. A reservation-based railway network operations management system. PhD dissertation. Department of Systems Engineering, University of Pennsylvania, Philadelphia, PA.

Kwon O.K., C.D. Martland, and J.M. Sussman. 1998. Routing and scheduling temporal and heterogeneous freight car traffic on rail networks. *Transportation Research* **34E**, 101-115.

Lawler, E. L. 1963. The quadratic assignment problem. *Management Science* **9**, 586-599.

Li, Y., P. M. Pardalos, and M. G. C. Resende. 1994. A greedy randomized adaptive search procedure for the quadratic assignment problem. In *Quadratic Assignment and Related Problems*, Edited by P. M. Pardalos and H. Wolkowicz, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science,* American Mathematical Society, Providence, RI, pp. 237-261.

Lin, S. 1965. Computer solutions of the traveling salesman problem. *Bell System Technical Journal* **44**, 2245-2269.

Lin, S., and B.W. Kernighan. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* **21**, 498-516.

Liu J. 2003. Solving real-life transportation scheduling problems. *PhD dissertation.* Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL.

Lloyd, S.P., and H.S. Witsenhausen. 1986. Weapons allocation is NP-complete. In *Proceedings of the 1986 Summer Conference on Simulation*, Reno, NV.

Maltin, S.M. 1970. A review of the literature on the missile-allocation problem. *Operations Research* **18**, 334-373.

Malucelli, F. 1993. Quadratic assignment problems: solution methods and applications. PhD Doctoral Dissertation, Departimento di Informatica, Universita di Pisa, Italy.

Maniezzo, V., A. Colorni, and M. Dorigo. 1994. The ant system applied to the quadratic assignment problem. Tech. Rep. IRIDIA/94-28, Université Libre de Bruxelles, Belgium.

Manne, A.S. 1958. A target-assignment problem. *Operations Research* **6**, 346-351.

Metler, W.A., and F.L. Preston. 1990. A suite of weapon assignment algorithms for a SDI mid-course battle manager. NRL Memorandum Report 671, Naval Research Laboratory, Washington, DC.

Miladenovic, N., and P. Hansen. 1997. Variable neighborhood search. *Computers and Operations Research* **34**, 1097-1100.

Morlok, E.K., and R.B. Petersen. 1970. *Railroad Freight Train Scheduling, a Mathematical Programming Formulation*. T*ransportation Center and Technological Institute, Northwestern University*, Evanston, IL.

Muller-Merbach, H. 1970. *Optimale Reihenfolgen*. Springer Verlag, Berlin, 158-171.

Murphey, R.A. 1999. Target-based weapon target assignment problems. *Nonlinear Assignment Problems: Algorithms and Applications*, Edited by P. M. Pardalos and L. S. Pitsoulis, Kluwer Academic Publishers, Boston, pp. 39-53.

Murty, K.G. 1976. *Linear and Combinatorial Optimization*. John Wiley, New York.

Newman A.M., L. Nozick, and C.A. Yano. 2002. Optimization in the rail industry. *Handbook of Applied Optimization*, Edited by P. M. Pardalos and M. Resende, Oxford University Press, New York.

Newton, H.N., C. Barnhart, and P.M. Vance. 1998. Constructing railroad blocking plans to minimize handling costs. *Transportation Science* **32**, 330-345.

Nozick L.K., and E.K. Morlok. 1997. A model for medium-term operations planning in an intermodal rail-truck service. *Transportation Research* **31A**, 91-107.

Nugent, C. E., T. E. Vollman, and J. Ruml. 1968. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research* **16**, 150-173.

Orlin, D. 1987. Optimal weapons allocation against layered defenses. *Naval Research Logistics* **34**, 605-616.

Osman, I.H., and G. Laporte. 1996. Meta-heuristics: A bibliography. *Annals of Operations Research* **63**, 513-623.

Pardalos, P. M., and J. Crouse. 1989. A parallel algorithm for the quadratic assignment problem. In *Proceedings of the Supercomputing 1989 Conference*, ACM Press, pp. 351-360.

Pardalos, P.M., F. Rendl, and H. Wolkowicz. 1994. The quadratic assignment problem. In *Quadratic Assignment and Related Problems*, Edited by P. M. Pardalos and H. Wolkowicz, DIMACS Series, American Mathematical Society, Providence, RI, pp. 1-42.

Reeves, C.R. 1993. *Modern Heuristic Techniques for Combinatorial Optimization*. Blackwell Scientific Publications, Oxford.

Resende, M. G. C., P. M. Pardalos, and Y. Li. 1994. Fortran subroutines for approximate solution of dense quadratic assignment problems using GRASP. *ACM Transactions on Mathematical Software* **22**, 104-118.

Rochat, Y., and E.D. Taillard. 1995. Probabilistic diversification and intensification in local search for vehicle routing. *Journal of Heuristics* **1**, 147-167.

Schaerf A. 1999. A survey of Automated Timetabling. *Artificial Intelligence Review* **13**, 87-127

Skorin-Kapov, J. 1990. Tabu search applied to the quadratic assignment problem. *ORSA Journal on Computing* **2**, 33-45.

Stallaert J. 1997. Automated timetabling improves course scheduling at UCLA. *Interfaces* **27**, 67-81.

Taillard, E. 1991. Robust tabu search for the quadratic assignment problem. *Parallel Computing* **17**, 443-455.

Talluri, K.T. 1996. Swapping applications in a daily fleet assignment. *Transportation Science* **31,** 237-248.

Tate, D. E. and A. E. Smith. 1985. A genetic approach to the quadratic assignment problem. *Computers and Operations Research* **22**, 73-83.

Thomet, M.A. 1971. A user oriented freight railroad operating policy. *IEEE Trans. Systems, Man Cybernet* **1**, 349-356.

Thompson, P.M., and J.B. Orlin. 1989. The theory of cyclic transfers. MIT Operations Research Center Report 200-289, Cambridge, MA.

Thompson, P. M. and H.N. Psaraftis. 1993. Cyclic transfer algorithms for multi-vehicle routing and scheduling problems. *Operations Research* **41,** 935-946.

Thompson J. M. and Dowsland K. A. 1996. Variants of simulated annealing for the examination-timetabling problem. *Annals of Operations Research* **63**, 105-128.

Valdes R.A., Crespo E., Tamarit J.M. 2002. Design and implementation of a course scheduling system using tabu search. *European Journal of Operational Research* **137**, 512-523.

Van Dyke, C.D. 1986. The automated blocking model: a practical approach to freight railroad blocking plan development. *Proceedings of the 27th Annual Meeting of the Transportation Research Forum* **27**, 116-121.

Wacholder, E. 1989. A neural network-based optimization algorithm for the static weapon-target assignment problem. *ORSA Journal on Computing* **4**, 232-246.

Werra D. D. 1997. The combinatorics of timetabling, *European Journal of Operational Research* **96**, 504-513.

West, D. H. 1983. Algorithm 608: approximate solution of the quadratic assignment problem. *ACM Transactions on Mathematical Software* **9**, 461-466.

Wilhelm, M. R., and T. L. Ward. 1987. Solving quadratic assignment problems by simulated annealing. *IEEE Transactions* **19**, 107-119.

BIOGRAPHICAL SKETCH

Krishna Jha is a native of India and got his early education at Darbhanga, India. He obtained his bachelor's degree in mechanical engineering from B. I. T. Sindri, India, and his master's degree in industrial and management engineering from I. I. T. Kanpur, India. After graduation, he worked in the largest private integrated steel plant in India, Tata Steel, for four years. Since 2000, he has been pursuing his PhD degree in the department of Industrial and Systems Engineering at University of Florida.