Cyclic Exchange and Related Neighborhood Structures for Combinatorial Optimization Problems

by

Dushyant Sharma

B. Tech., Computer Science and Engineering, Indian Institute of Technology, 1996

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

Doctor of Philosophy in Operations Research

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

© Massachusetts Institute of Technology 2002

Signature of Author

Sloan School of Management
17 May, 2002

Certified by

James B. Orlin
Edward Pennell Brooks Professor of Operations Research
Co-director, Operations Research Center
Thesis Supervisor

Accepted by

Cynthia Barnhart
Associate Professor of Civil and Environmental Engineering
Co-director, Operations Research Center
Cyclic Exchange and Related Neighborhood Structures for Combinatorial Optimization Problems

by

Dushyant Sharma

Submitted to the Sloan School of Management on 17 May 2001, in partial fulfillment of the requirement for the degree of Doctor of Philosophy in Operations Research

Abstract

In this thesis, we concentrate on neighborhood search algorithms based on very large-scale neighborhood structures. The thesis consists of three parts.

In the first part, we develop a cyclic exchange neighborhood search based approach for partitioning problems. A partitioning problem is to divide a set of $n$ elements into $K$ subsets $S_1, \ldots, S_K$ so as to minimize $f(S_1) + \ldots + f(S_K)$ for some specified function $f$. A partition $S'_1, \ldots, S'_K$ is called a cyclic exchange neighbor of the partition $S_1, \ldots, S_K$ if $|S'_i \setminus S_i| \leq 1$ and $|S_i \setminus S'_i| \leq 1$. The problem of searching the cyclic exchange neighborhood is NP-hard. We develop new exact and heuristic algorithms to search this neighborhood structure. We propose cyclic exchange based neighborhood search algorithms for specific partitioning problems. We provide computational results on these problems indicating that the cyclic exchange is very effective and can be implemented efficiently in practice.

The second part deals with the Combined Through and Fleet Assignment Model (cTFAM). This model integrates two airline planning models: (i) Fleet Assignment Model and (ii) Through Assignment Model, which are currently solved in a sequential manner because the combined problem is too large. This leads to sub-optimal solutions for the combined problem. We develop very large-scale neighborhood search algorithms for the cTFAM. We also extend our neighborhood search algorithms to solve the multi-criteria objective function version of the cTFAM. Our computational results using real-life data show that neighborhood search can be a useful supplement to the current integer-programming optimization methods in airline scheduling.

In the third part, we investigate the structure of neighborhoods in general. We call two neighborhood structures LO-equivalent if they have the same set of local optima for all instances of a combinatorial optimization problem. We define the extended neighborhood of a neighborhood structure $N$ as the largest neighborhood structure that is LO-equivalent to $N$. In this thesis, we develop some theoretical properties of the extended neighborhood and relate these properties to the performance of a neighborhood structure. In particular, we show that the well-known 2-opt neighborhood structure for the Traveling Salesman Problem has a very large extended neighborhood, providing justification for its favorable empirical performance.

Thesis Supervisor: James B. Orlin
Title: Edward Pennell Brooks Professor of Operations Research
Co-director, Operations Research Center
Contents

1 Introduction 15

1.1 Contributions 16

1.1.1 Cyclic Exchange Neighborhood Structure 16
1.1.2 Combined Through and Fleet Assignment Model 17
1.1.3 Extended Neighborhood 17

1.2 Outline of the Thesis 18

2 Cyclic Exchange Neighborhood Structure 19

2.1 Introduction 19

2.2 Cyclic Exchange Neighborhood Structure 21

2.2.1 Definition of the Neighborhood Structure 21
2.2.2 Improvement Graph 22
2.2.3 Review of Previous Work 25
2.2.4 Research Issues 26

2.3 Finding Subset Disjoint Cycles 27

2.3.1 Label-correcting heuristic for SDCP 29
2.3.2 Exact Algorithms for SDCP 32

2.4 Capacitated Minimum Spanning Tree Problem 39

2.4.1 Cyclic Exchange Neighborhood Structures for CMST 42
2.4.2 Neighborhood Search Algorithm 46
2.4.3 Generating Initial Feasible Solutions 47
2.4.4 Computational Testing 47
2.5 Additional Partitioning Problems 57
  2.5.1 Computational Results for GAP 57
2.6 Summary and Conclusions 59

3 Combined Through and Fleet Assignment Model 61
  3.1 Introduction 61
  3.2 An Integer Programming Formulation of ctFAM 66
  3.3 Swap based Neighborhood Structure for ctFAM 70
    3.3.1 A-B Improvement Graph 72
    3.3.2 Complexity of finding valid cycles 79
    3.3.3 Identifying A-B swaps 81
    3.3.4 Neighborhood Search Algorithms 82
    3.3.5 Alternate Definition of A-B Improvement Graph 82
  3.4 Implementation Details 84
  3.5 Computational Testing for FAM and ctFAM 87
  3.6 Multi-criteria Optimization for ctFAM 90
    3.6.1 Computational Testing 93
  3.7 Summary and Conclusions 94

4 Extended Neighborhood 96
  4.1 Introduction 96
  4.2 Extended Neighborhood 97
  4.3 Combinatorial Optimization Problems with linear objective 99
  4.4 Extended Neighborhood for TSP 2-opt 105
    4.4.1 Reachability in $G^{2\text{-opt}}$ and $G^{2\text{-opt}^*}$ 107
5 Conclusions and Future work
List of Figures

2-1 Illustration of 2-exchange and cyclic exchange 22
2-2 Illustration of a subset disjoint cycle and a cyclic-exchange 24
2-3 The subset-disjoint cycle heuristic based on label-correcting algorithm 31
2-4 The path enumeration algorithm for SDCP 34
2-5 A network with two nodes before and after preprocessing 36
2-6 An improvement graph with two paths 37
2-7 Forward dynamic programming recursion 38
2-8 An example of a capacitated spanning tree where each node has unit demand 40
2-9 The VLSN search algorithm for the CMST problem 46
3-1 An approach to solve ctFAM 64
3-2 Part of the connection network at a city with arrivals 1, 2, 3, and departures 4, 5, and 6 68
3-3 Illustrating an A-B swap 71
3-4 Effect of swaps on the number of planes used 72
3-5 Valid cycle and the corresponding A-B swap 77
3-6 The neighborhood search algorithm for ctFAM 82
3-7 Illustration of an approximate bank structure 83
3-8 Solutions obtained from Local1 and Local2 93
3-9 Solutions obtained from Tabu1 and Tabu2 94
3-10 Comparison of solutions from Tabu2, Local2, and Sequential 94
4-1 a tour, 2-opt moves \{1, 5\}, \{3, 8\}, 2-opt moves \{1, 5\}, \{3, 8\}, and \{6, 10\} 107
4-2  Tour obtained by move \{1, 5\}. Tour obtained by move \{3, 8\}  

108

4-3  Procedure to generate tree of some tours in 2-opt*  

110

4-4  A tour such that no node \(i\) is adjacent to \(i+1\) or \(i-1\)  

115

4-5  Algorithm to generate nested set  

116
List of Tables

2-1 Comparison of node-exchange and tree-exchange for homogeneous demand problems 51
2-2 Comparison of node-exchange and tree-exchange for heterogeneous demand problems 52
2-3 Comparison of 2-exchange with cyclic exchange neighborhood structure 53
2-4 Performance of the label-correcting heuristic 53
2-5 Computational results of composite neighborhood for homogeneous demand problems 55
2-6 Computational results of composite neighborhood for heterogeneous demand problems 56
2-7 Computational behavior of the composite neighborhood based algorithm 57
2-8 Results of cyclic exchange neighborhood search for GAP 59
3-1 Different types of arcs in the A-B improvement graph 75
3-2 Improvements obtained by the local improvement algorithm 89
3-3 Improvements obtained by the tabu search algorithm 89
3-4 Behavior of the local improvement algorithm 90
3-5 Behavior of the tabu search algorithm 90
Acknowledgements

I have received support and guidance from several individuals during the course of this thesis. I am grateful to my advisor Jim Orlin for his many useful comments and insights throughout my research, and for supporting me at MIT. I would like to thank my committee members, Cindy Barnhart and Tom Magnanti, for providing useful suggestions in writing this thesis, which have helped improve the presentation here. I am also thankful to Andreas Schulz for providing several interesting research discussions during my stay at MIT. My special gratitude goes to Ravi Ahuja for motivating me to study Operations Research during my undergraduate studies, and for continuous help in many ways during my graduate studies.

I would like to thank United Airlines for the supporting my research and providing data for testing our algorithms.

The ORC staff: Paulette, Laura, and Danielle, made the life at ORC very comfortable through their efforts. I had the pleasure of being friends with many students at ORC, whose company made the ORC experience very exciting. I specially thank Adam Jon, Kermit, Eduardo, Rudolfo, Soulaymane, Peng, Ramazan, Ozie, Sanne, Mahesh, Romy, and Anshul for sharing good and bad moments with me at various times.

My love and gratitude goes to Sally for sharing my life, and her patience and support during the last two years of my PhD. Finally, I am most grateful to my family for their continuous support of my studies. The love and appreciation of my parents have been a major driving force in this effort.
To my father and mother
Chapter 1

Introduction

A neighborhood search algorithm is an iterative procedure that typically starts with an initial solution and at each iteration searches for a better solution in the neighborhood of the current solution. If the algorithm finds a better solution, it replaces the current solution with the better solution and continues. If the neighborhood of the current solution contains no better solution, then the algorithm returns the current solution, which is called a locally optimal solution. In the past decade, the fields of operations research, mathematical programming, and computer science have all witnessed a strong interest in the development and analysis of neighborhood search algorithms and developed fairly sophisticated neighborhood search algorithms. Neighborhood search algorithms are now widely regarded as an important tool to solve difficult combinatorial optimization problems effectively (Aarts and Lenstra [1997]).

Clearly, one of the most important issues in the design of any neighborhood search approach is the definition of the neighborhood of a solution. Typically, the larger the size of the neighborhood of each solution, the better the quality of the resulting local optima. Therefore, as a rule of thumb, it is desirable to design neighborhood structures that contain a large number of neighbors for each solution. However, a larger neighborhood might take more time to search. Since solution procedures generally perform many runs of a neighborhood search algorithm with different starting solutions, the longer computation times per iteration result in fewer runs of the neighborhood search algorithm in a given amount of time. Therefore, just defining a large neighborhood does not ensure an effective neighborhood search heuristic. The neighborhood structure should also be searchable efficiently.

Researchers have proposed a number of very large-scale neighborhood (VLSN) search algorithms for combinatorial optimization problems (Ahuja et al. [2001] and Deineko and Woeginger [1998]). These algorithms are based on extremely large neighborhood structures,
typically exponential in the size of the input problem. The VLSN search algorithms rely on some form of implicit enumeration to search for a better neighbor instead of explicitly evaluating every neighbor of a solution.

In this thesis, we are interested in developing new VLSN algorithms for certain combinatorial optimization problems and also analyzing some theoretical issues related to design of neighborhood structures. We now describe the main contributions of this thesis.

1.1 Contributions

The contributions of this thesis can be divided into three parts. In the first part, we focus on the cyclic exchange neighborhood structure for a class of partitioning problems. Thompson and Orlin [1989] proposed the use of this very large-scale neighborhood structure. The problem of searching the cyclic exchange neighborhood is NP-hard. We develop new exact and heuristic algorithms to search this neighborhood structure. Using computational testing on specific partitioning problems, we show how to efficiently implement cyclic exchange in practice even though the search problem is hard. In the second part, we develop new very large-scale neighborhood search algorithms for the Combined Through and Fleet Assignment Model (cTFAM), a problem arising in the airline scheduling. Our neighborhood structure generalizes the neighborhood structure proposed by Talluri [1996] for the fleet assignment problem. We also extend our neighborhood search algorithms to solve the cTFAM when the objective function is a multi-criteria optimization problem. In the third part, we develop a new concept called the extended neighborhood of a neighborhood structure. Using this concept, we provide some theoretical insight into the behavior of certain small neighborhood structures that tend to behave quite well in practice. We now provide specific details of our contributions in each of these areas.

1.1.1 Cyclic Exchange Neighborhood Structure for Partitioning Problems

In proposing the cyclic exchange neighborhood structure for partitioning problems, Thompson and Orlin [1989] introduced a data structure called the improvement graph and showed that searching for a better neighbor in the cyclic exchange neighborhood of the solution of a partitioning problem is equivalent to finding a negative cost subset disjoint cycle in the improvement graph. They also showed that the problem of finding a subset disjoint cycle in a graph is NP-hard. In this thesis, we develop heuristic and exact algorithms to solve this problem. Our heuristic algorithms are based on modifications of the classical label-correcting algorithm to find shortest paths in a network. We propose exact algorithms that are based on integer programming, implicit enumeration, and dynamic programming. We provide methods of
efficiently implementing our DP based algorithm as well as techniques to preprocess the input to make the algorithm effective. Using our computational tests on cyclic exchange neighborhood search algorithms for a specific partitioning problem called the Capacitated Minimum Spanning Tree Problem, we show how to efficiently implement the cyclic exchange neighborhood search in practice. Finally, we provide computational results of cyclic exchange on two additional partitioning problems: Generalized Assignment Problem and Printed Circuit Board Clustering Problem, showing that the cyclic exchange is effective for a variety of partitioning problems.

1.1.2 Combined Through and Fleet Assignment Model (ctFAM)

In this part, we introduce and provide an integer programming formulation for a combined through and fleet assignment model. The ctFAM integrates two models that are currently solved sequentially: (i) fleet assignment model (FAM), and (ii) through assignment model (TAM). Analysts have adopted the sequential approach because the combined model is too large to be solved optimally using commercially available solvers. We propose two neighborhood structures for the ctFAM based on swapping of fleet types between different flight legs. These neighborhood structures are also applicable for the FAM and generalize the swap based neighborhood structure proposed by Talluri [1996]. We show that the neighborhood search problem for both of our neighborhood structures is NP-hard. We formulate this search problem as an integer programming problem. Our computational results show that CPLEX can solve this formulation very efficiently. We also discuss ways of handling additional operational constraints for the ctFAM into our neighborhood search technique. We provide computational results of our algorithms on real world instances of this problem. Finally, we consider a multiple criteria objective function version of the ctFAM. We discuss ways of applying neighborhood search to solve this multi-criteria problem.

1.1.3 Extended Neighborhood of a Neighborhood Structure

In neighborhood search some seemingly small neighborhood structures perform quite well in practice. For example, even though it is small, the 2-opt neighborhood structure for the traveling salesman problem provides very good empirical results. In this part, we introduce a new concept called the extended neighborhood for a neighborhood structure for any combinatorial optimization problem. Using this idea, permits us to interpret many small neighborhood structures as very large-scale neighborhood structures. We prove certain properties of the extended neighborhood of a neighborhood structure for combinatorial optimization problems with linear objective functions. In particular, we provide a geometric characterization of the extended neighborhood. We study properties of the extended neighborhood for the 2-opt neighborhood
structure for the TSP, showing how to view the 2-opt neighborhood structure as a very large-scale neighborhood structure in the sense of its extended neighborhood structure. This result provides some explanation for the favorable empirical performance of 2-opt in practice.

1.2 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2 we describe the cyclic exchange neighborhood structure and the research issues associated with any implementation of cyclic exchange to a partitioning problem. We describe our exact and heuristic algorithms to solve the subset disjoint cycle problem. We also provide computational results for the cyclic exchange neighborhood search algorithms for Capacitated Minimum Spanning Tree Problem and Generalized Assignment Problem. Chapter 3 covers the description of the combined through and fleet assignment problem. We describe our neighborhood search algorithms for this problem and the multi-objective version of the same problem. In Chapter 4 we introduce the concept of extended neighborhood and provide analysis for the 2-opt neighborhood for the traveling salesman problem. Finally, we provide conclusions and future research directions in Chapter 5.
Chapter 2

Cyclic Exchange Neighborhood Structure

2.1 Introduction

Partitioning problems are among the most difficult combinatorial optimization problems to solve to optimality, and they arise frequently in practice. Hence there is a need for good heuristics that can find nearly optimal solutions efficiently. Thompson and Orlin [1989] proposed the cyclic exchange neighborhood structure for partitioning problems. This neighborhood structure contains an extremely large number of neighbors for each solution and can be considered as a very large-scale neighborhood structure. Thompson and Orlin [1989] also defined a data structure called the improvement graph such that the problem of finding a better cyclic exchange neighbor for a solution can be transformed into the problem of finding a negative cost (constrained) cycle in the improvement graph corresponding to the solution. One of the difficulties with applying this neighborhood structure to any partitioning problem is that the problem of searching for a better cyclic exchange neighbor of a solution itself is NP-hard. Many of the neighborhood search algorithms for partitioning problems search a subset of the cyclic exchange neighbors of a solution. In this chapter, we present new heuristic and exact algorithms for searching the cyclic exchange neighborhood of a solution. We are also concerned about the efficiency of our algorithms in practice and describe methods of efficiently implementing them. We develop new cyclic exchange neighborhood structure based algorithms for a particular partitioning problem: Capacitated Minimum Spanning Tree Problem (Amberg et al. [1996]). Our algorithms achieved or improved the best-known results for commonly used benchmark instances for this problem. Our computational results for these algorithms show that the cyclic exchange neighborhood structure can be implemented efficiently in practice and is quite powerful.

The rest of the chapter is organized as follows. We provide a description of the cyclic exchange neighborhood structure and the concept of the improvement graph in Section 2.2. In
this section, we also outline the primary research issues that need to be examined while applying cyclic exchange based algorithms and review the previous work on cyclic exchange neighborhood structures. In Section 3, we develop new algorithms for searching the cyclic exchange neighborhood structure and discuss their implementation issues. We propose new cyclic exchange based neighborhood search algorithms for the capacitated minimum spanning tree in Section 4. We also provide computational results for these algorithms on a commonly used set of benchmark problems. In Section 5, we provide brief description of a cyclic exchange neighborhood search algorithm and its computational results on generalized assignment problem, a well-known partitioning problem. Finally, in Section 6 we provide a summary and conclusions for the results in this chapter.

We first define the class of partitioning problem for which we are interested in studying the cyclic exchange neighborhood structure. Let \( A = \{a_1, a_2, a_3, \ldots, a_n\} \) be a set of \( n \) elements. The subsets \( S_1, S_2, S_3, \ldots, S_K \) for some integer \( K > 1 \) define a partition of \( A \) if they are pairwise disjoint and their union is \( A \). We represent this partition as \( S = \{S_1, S_2, S_3, \ldots, S_K\} \). We let \( c_k(S_k) \) denote the optimal cost of the part \( S_k \), \( k = 1, \ldots, K \). The cost function \( c_k \) may be complex and time consuming to evaluate and may involve the optimal cost of arranging elements in the subset \( S_k \). For example, if \( S_k \) denotes a set of cargoes assigned to vehicle \( k \), then \( c_k(S_k) \) may denote the minimum cost of a tour for vehicle \( k \) that delivers all of the cargoes in \( S_k \). The partitioning problem is to

\[
\text{Minimize } c(S) = \sum_{k=1}^{K} c_k(S_k), \text{ where } S = \{S_1, S_2, S_3, \ldots, S_K\} \text{ is a partition of } A. \tag{2.1}
\]

An important property of this partitioning problem is that the objective function is a sum of the optimization functions \( c_k \) that only depend on the set \( S_k \). There are several interesting problems that have the form of the partitioning problem in (2.1). For example, in the Vehicle Routing Problem (Laporte [1997]), a set of customers needs to be assigned to service vehicles and the optimal routes of these service vehicles must be determined. In this case, the assignment of customers to the vehicles is a partition and the cost of the set of customers assigned to a vehicle is equal to the minimum length route going through that set of customers, starting and ending at a depot. Other examples of partitioning problems include Generalized Assignment Problem (Savelsbergh [1997]), Capacitated Minimum Spanning Tree Problem (Amberg et al. [1996]), Facility Location Problems (Labbe and Louveaux [1997]), Clustering Problems (Mirkin and Muchnik [1998]), and several Parallel Machine Scheduling problems (Pinedo [1995]). This class
of partitioning problems contains several important application problems that are hard to solve and there is a need to develop effective heuristics to solve these problems. We now describe the cyclic exchange neighborhood structure that can be applied to any problem in this class of partitioning problems. It is a very large-scale neighborhood structure and the problem of searching for a better solution in the cyclic exchange is NP-hard.

2.2 Cyclic Exchange Neighborhood Structure

A common neighborhood structure used for the partitioning problems is the 2-exchange or swap neighborhood. Given a partitioning solution \( S = \{S_1, S_2, S_3, \ldots, S_K\} \) and \( a_i \in A \), we use \( S[a_i] \) to represent the partition containing \( a_i \), i.e., \( S[a_i] = k \) if \( a_i \in S_k \). A swap neighbor of \( S \) is obtained by exchanging the assignments of two elements \( a_i, a_j \) such that \( S[a_i] \neq S[a_j] \). Clearly, the number of 2-exchange neighbors of the solution \( S \) is \( O(n^2) \). The cyclic exchange neighborhood is a generalization of this neighborhood structure. We now describe it.

2.2.1 Definition of the Neighborhood Structure

A cyclic exchange is represented as \( a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_r \rightarrow a_1 \), where the elements \( a_1, a_2, a_3, \ldots, a_r \) belong to different subsets. It represents the following changes: element \( a_1 \) moves from the subset \( S[a_1] \) to the subset \( S[a_2] \), element \( a_2 \) moves from the subset \( S[a_2] \) to the subset \( S[a_3] \), and so on, and, finally, the element \( a_r \) moves from the subset \( S[a_r] \) to the subset \( S[a_1] \), thus completing a cycle of changes. We represent a path exchange as \( a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_r \rightarrow a_1 \), where the elements \( a_1, a_2, \ldots, a_r \) belong to different subsets. This path exchange represents the same changes as represented by the cyclic exchange \( a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_r \rightarrow a_1 \) except that the element \( S_i \) does not leave the subset \( S[a_i] \) to join \( S[a_1] \). Notice that a path exchange decreases the cardinality of the subset \( S[a_1] \), increases the cardinality of the subset \( S[a_r] \), and the cardinalities of all other subsets remain unchanged. On the other hand, a cyclic exchange does not change the cardinality of any subset. We illustrate the cyclic exchange and 2-exchange using an example in Figure 2-1. One can transform any partitioning problem so that a path exchange for the original problem is transformed into a cyclic exchange for the modified problem. (It suffices to introduce a dummy subset as well as dummy nodes in each part.) For this reason, in subsequent discussion, we will only consider cyclic exchanges.

Let \( S' = \{S'_1, S'_2, \ldots, S'_K\} \) be the partitioning solution obtained by performing a cyclic exchange \( a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_r \rightarrow a_1 \) on \( S \). We denote the cost of this exchange as \( c(S') \). The
cost of the cyclic exchange is \( c(S') - c(S) = \sum_{k=1}^{r} (c_k(S'[a_k]) - c_k(S[a_k])) \). (In many applications, the cost function is the same for all subsets (for example, in the parallel machine scheduling problem when all machines are identical). In other words, to determine the cost of the cyclic exchange, we compute \( c_k(S'[a_k]) \) for each subset \( S'[a_k] \) for \( 1 \leq k \leq r \), and compute the sum of the changes in the costs of these subsets. Computing \( c_k(S'[a_k]) \) typically involves solving an optimization problem for the subset \( S'[a_k] \). The cyclic exchange is called cost-decreasing if \( c(S') - c(S) < 0 \).

![Diagram](image)

**Figure 2-1. Illustration of 2-exchange and cyclic exchange.**

For a given feasible partition \( S \), \( S' \) is said to be a neighbor of \( S \) if \( S' \) can be obtained from \( S \) by performing a cyclic exchange. We define the cyclic exchange neighborhood of \( S \) as the collection of all the partitions that are neighbors of \( S \). The neighborhood based on the cyclic exchanges is generally very large and grows exponentially with \( n \) and \( K \). Hence, explicitly examining the entire neighborhood to identify a cost-decreasing cyclic exchange (or, the optimum cost-increasing cyclic exchange) is often computationally infeasible. However, one can use the concept of improvement graph, to implicitly search this neighborhood. We now describe this data structure.

### 2.2.2 Improvement Graph

The improvement graph for the cyclic exchange neighborhood is defined with respect to a feasible partition \( S \) and is represented by \( G(S) = (N, A) \). The graph \( G(S) \) is a directed graph with
$n$ nodes, where each node $i$ corresponding to the element $a_i \in A$. A directed arc $(i, j)$ in $G(S)$ signifies that element $a_i$ leaves its current subset and moves to the subset containing element $a_j$, that is, the subset $S[a_j]$, and simultaneously the element $a_j$ leaves $S[a_i]$. To construct $G(S)$, we consider every pair $a_i$ and $a_j$ of elements in $S$, and add arc $(i, j)$ to $G(S)$ if and only if the elements $a_i$ and $a_j$ belong to different subsets in $S$. Let $m$ denote the number of arcs in the improvement graph $G(S)$. We define the cost $c_{ij}$ of arc $(i, j)$ as $c((a_i) \cup S[a_j](a_j)) - c(S[a_j])$. A directed cycle $W = i_1 - i_2 - \ldots - i_r - i_1$ in $G$ is called subset disjoint if $S[i_k] \neq S[i_{k'}]$ for $k \neq k'$, that is, if the elements in $S$ corresponding to the nodes in $W$ belong to different subsets. We use $N(W) = \{i_1, i_2, \ldots, i_r\}$ and $A(W) = \{(i_k, i_{k+1}) : k = 1, \ldots, r-1\} \cup \{(i_r, i_1)\}$ to represent the set of nodes and the set of arcs in the cycle $W$, respectively. The cost of the cycle $W$ is given by $C(W) = \sum_{(i,j) \in A(W)} c_{ij}$.

The following theorem states the correspondence between cyclic exchanges and the improvement graph.

**Theorem 2.1** (Thompson and Orlin [1989]). There is a one-to-one correspondence between cyclic exchanges with respect to $S$ and subset-disjoint directed cycles in the improvement graph $G(S)$, and both have the same cost.

Theorem 2.1 implies that a cost-decreasing cyclic-exchange with respect to $S$ corresponds to a negative-cost subset-disjoint (directed) cycle in $G(S)$. In Figure 2-2, we show an example of a cyclic exchange illustrating that the cost of the exchange is equal to the cost of the corresponding subset disjoint cycle in the improvement.

Although there exist polynomial-time algorithms to identify (unconstrained) negative cycles (Ahuja, Magnanti and Orlin [1993]), the following hardness result for the finding a subset disjoint in a graph was shown by Thompson and Orlin [1989].

**Theorem 2.2** (Thompson and Orlin [1989]). The problem of finding a subset disjoint cycle in a graph is NP-hard.

The NP-hardness of identifying subset disjoint cycles suggests that theoretically efficient algorithms are unlikely for this problem. So we limit attention to heuristics as well as to exact search approaches that are not guaranteed to run in polynomial time.
cost of a subset disjoint cycle = cost of the cyclic-exchange

Figure 2-2. Illustration of a subset disjoint cycle and a cyclic-exchange.

Finally, we note here that even though we described a cyclic exchange as a transformation where at most a single element is transferred into a subset or out of a subset, one can certainly extend the definition of cyclic exchange to permit movement of more than one element out of a subset or into a subset. For example, in the vehicle routing problem, we can define improvement graphs where each node $i$ represents a subset of the customers that are assigned to a vehicle. We denote the subset corresponding to node $i$ as $D(i)$, which would be a proper subset of $S[a_j]$. In this case, arc $(i, j)$ in the improvement graph corresponds to adding $D(i)$ to $S[a_j]$ and then eliminating $D(j)$ from the subset $S[a_j]$. Since there are an exponential number of possible subsets, it would not be practical to create a node in the improvement graph for each subset of elements. However, by carefully selecting the set of subsets that should be considered for exchanges, the cyclic exchange can be made very effective. For example, Thompson and Psaraftis [1993] introduced a node in the improvement graph for each pair of customers currently being served consecutively by the same vehicle. It is easy to see that the number of such nodes is
less than the number of customers in this case, resulting in a small increase in the number of nodes in the improvement graph.

2.2.3 Review of Previous Work

Some prior work has applied cyclic exchange based neighborhood search algorithms for partitioning problems. Thompson and Psaraftis [1993] studied several cyclic exchange neighborhood structures for variants of the vehicle routing problem (with and without time-windows). They considered cyclic exchanges permitting one or two customers to be transferred between different vehicles. They developed a variable depth search heuristic for the negative cost subset disjoint cycle problem, which they used along with 2-exchange and 3-exchange moves to search a subset of the cyclic exchange neighbors of a solution. They were able to obtain results comparable and in some cases better than the previous algorithms at the time. Gendreau et al. [1998] developed cyclic exchange based algorithms for the dynamic vehicle routing problem, where new customers are added to the planned schedule using a cyclic exchange. They used a heuristic based on modification of the Floyd-Warshall algorithm for finding shortest paths in a graph (Ahuja, Magnanti, Orlin [1993]) to find a negative cost subset disjoint cycle.

We note that the cyclic exchanges can be considered as a special case of the concept of ejections chains proposed by Glover [1992]. Many of the ejection chain based algorithms for partitioning problems search a subset of the solutions in the cyclic exchange neighborhood of a solution. These algorithms treat a cyclic exchange as a sequence of shifts of single elements between subsets. Typically, they start the search by shifting one element from its current subset to another, and then try to identify another element to be moved into the current subset of the last shifted element using some greedy criterion. Using this process, they generate a sequence of shift moves that either corresponds to a cyclic exchange or a path exchange. The process can be viewed as a limited enumeration of the cyclic and path exchanges. Usually, these methods do not compute the improvement graph as they need to evaluate only a very small number of the edges in the improvement graph. Ejection chain based methods have been successfully applied to some partitioning problems. The algorithms of Kernighan and Lin [1970] and Fiduccia and Mattheyes [1982] for graph partitioning are ejection chain algorithms. Rego and Roucairol [1996] and Rego [1998] applied ejection chain algorithms to vehicle routing problem. An ejection chain based tabu search method has been proposed by Yagiura et al. [1999] for the generalized assignment problem. Ejection chain algorithms were applied to multi-level generalized assignment problem by Laguna et al. [1995]. Clustering algorithms using ejection chains were given by Dondorf and
Pesch [1994]. Other applications of ejection chains include algorithms for categorized assignment problem by Aggarwal et al. [1986], and nurse scheduling by Dowsland [1998].

2.2.4 Research Issues

We have described the cyclic exchange neighborhood structure for a generic partitioning problem. In Section 2.2.2, we noted that the cyclic exchange can be defined in several ways for a problem by considering the transfer of single or multiple elements from a subset to another subset. Clearly, one of the research issues in applying cyclic exchange to any problem is the definition of the cyclic exchange itself. This choice may be determined experimentally or through prior knowledge of the structure of the partitioning problem being considered. Different definitions of the cyclic exchange neighborhood can be obtained by considering different choices of the elements that can be moved as a group.

Once the definition of the cyclic exchange neighborhood structure to be used in a neighborhood search algorithm is determined, there are two primary computational issues that need to be addressed: (i) constructing and updating improvement graphs; and (ii) identifying subset disjoint cycles in improvement graphs. The first computational issue is inherently dependent on the specific partitioning problem at hand because the construction of the improvement graph involves computing the arc costs, which are dependent on the objective function of the problem. On the other hand, the problem of identifying negative cost subset disjoint cycles is an independent research problem and any algorithmic success in solving it better could be beneficial to any cyclic exchange neighborhood search algorithm. We study this problem in greater detail in section 3 of this chapter. In the rest of this section, we outline some generic research questions related to the issue of constructing and updating the improvement graphs. We shall answer some of these questions for our cyclic exchange neighborhood search algorithms for the capacitated minimum spanning tree problem and the generalized assignment problem, which are described later in this chapter.

Constructing and Updating Improvement Graphs

In our description of the improvement graph, we consider every pair of elements $i$ and $j$ such that $a_i$ and $a_j$ do not belong to the same subset. However, adding all the arcs to the improvement graph may not be useful for the following reasons. We formulate the partitioning problem as an unconstrained problem and assumed that any side constraints could be captured using large penalties in the objective function for violation of these constraints. Therefore, any arc $(i, j)$ that results in making the subset containing $j$ infeasible would have a huge positive cost.
Such an arc is not likely to be part of any negative cost cycle and can be removed. Even when the move corresponding to an arc \((i, j)\) keeps the subset containing \(a_j\) feasible, it may not be so profitable and unlikely to be in a cost decreasing cycle and could be removed. Therefore, we need to address the following research issues: (i) should an arc \((i, j)\) be added to the improvement graph? and if yes (ii) how to compute the cost of the arc \((i, j)\) efficiently? Algorithms for computing the cost of an arc in the improvement graph are inherently problem-specific. In some cases, one may compute the cost of \((i, j)\) efficiently, whereas in other cases, computing the cost of arc \((i, j)\) may itself be an NP-hard problem. For example, for the capacitated minimum spanning tree problem determining whether arc \((i, j)\) belongs to the improvement graph (that is, the cyclic exchange might lead to a feasible configuration) takes \(O(1)\) time and its cost can be determined in \(O(Q \log Q)\) time where \(Q\) denotes the number of nodes in a subtree connected to the root node. For the generalized assignment problem, both the feasibility and cost of an arc can be determined in \(O(1)\) time. On the other hand, for the vehicle routing problem, determining the cost of arc \((i, j)\) is an NP-hard problem since it requires solving a traveling salesman problem over the modified subset. Hence one important issue in this research is to establish the complexity of creating the improvement graph for a variety of subclasses of partitioning problems. In cases where creating the improvement graph is NP-hard, one can focus on developing effective approximation heuristics to create the graph. That is, one can compute the costs of the improvement graph approximately when the costs cannot be computed exactly. Thompson and Psaraftis [1993] considered the cyclic exchange neighborhood for the vehicle routing problem and proposed several ways of computing the arc costs approximately in the improvement graph.

We can also investigate other methods for speeding up the construction of the improvement graph. For example, in many cases the cost of the arc \((i, j)\) does not change subsequent to a cyclic exchange, in which case we would not need to compute it again. Also, in computing the cost of arc \((i, j)\), we may obtain valuable information for computing the cost of another arc \((i, k)\). Hence an important research issue is whether one can compute the cost for all arcs of the improvement graph in far less total time than it takes to compute each of the arc costs one at a time. We show such a result later on for a cyclic exchange neighborhood structure for the capacitated minimum spanning tree problem.

2.3 Finding Subset Disjoint Cycles

From Theorem 2.1, we know that searching for a better neighbor in the cyclic exchange neighborhood of a partitioning solution is equivalent to finding a negative cost subset disjoint cycle in the improvement graph corresponding to the solution. Therefore, we need to solve the
problem of finding negative subset disjoint cycles in a graph. In some search algorithms such as
tabu search, we need to find a neighbor with minimum objective value if there are no neighbors
better than the current solution. This corresponds to finding the least cost non-empty subset
disjoint cycle in the improvement graph. Therefore, for the tabu search algorithms we are
interested in finding a minimum cost non-empty subset disjoint cycle in the improvement graph.
We can formulate these requirements as an optimization problem as follows. Given a graph \( G = (N, A) \), arc costs \( c_{ij} \) associated with each arc \( (i, j) \in A \), and a partition \( \{S_1, S_2, \ldots, S_K\} \) of the node
set \( N \), the \textit{subset disjoint cycle problem (SDCP)} is to find the minimum cost non-empty subset
disjoint cycle in \( G \).

Thompson and Orlin [1989] showed that finding a subset disjoint cycle in a graph is \textit{NP-}
complete, which suggests that any theoretically efficient algorithms for SDCP are unlikely.
Therefore, we focus on heuristics, and exact algorithms that do not have guarantees to run in
polynomial time. We note that for the purpose of neighborhood search, we do not have to find the
minimum cost negative cost subset disjoint cycle in the improvement graph. It is sufficient to find
any negative cost subset disjoint cycle to get a better neighbor. Therefore, a heuristic algorithm
for the SDCP that consistently finds a negative cost subset disjoint cycle (if the improvement
graph contains one) is good enough for a neighborhood search algorithm. In case of tabu search,
if the improvement graph does not contain any negative cost subset disjoint cycle, it may be
sufficient to compute the best non-empty cycle obtained by the heuristic instead of using the
minimum cost non-empty subset disjoint cycle. In this chapter, we present some new heuristic
and exact algorithms for SDCP.

There has been limited effort in developing algorithms for SDCP in the past. Thompson
and Psaraftis [1993] used a variable depth search based method to heuristically search the cyclic
exchange neighborhood. In addition to their variable depth search method, they also searched the
2-exchange and 3-exchange neighbors of the current solution. Gendreau et al. [1998] used a
heuristic obtained by modifying the Floyd-Warshall algorithm for finding shortest paths in a
graph (Ahuja, Magnanti, Orlin [1993]). Many of the ejection chain search methods for
partitioning problems can be viewed as algorithms for finding negative cost subset disjoint cycles
in the improvement graph, but these methods do not use the improvement graph.

We tested the modification of the Floyd-Warshall algorithm used by Gendreau et al.
[1998] and found that it failed to find a negative cost subset disjoint cycle in many cases when the
improvement graph contained such a cycle. In Section 2.3.1, we describe a new heuristic
algorithm for finding negative cost subset disjoint cycles. Our computational results in Section
2.4.4 indicate that the method is very consistent in finding negative cost subset disjoint cycles in
the improvement graph. Among the exact algorithms, we propose a dynamic programming based
method that takes $O(nmK^2)$. This is the best-known running time of any exact algorithm for the
SDCP. We note that the running time of this algorithm is exponential only in the number of
partitions in the solution. Therefore, this method can certainly be very efficient for problems
where the number of partitions is small. Our computational results indicate that the algorithm is
quite robust even for partitioning problems with large number of partitions. We describe some
preprocessing techniques to improve the empirical performance of our algorithm.

We note that concurrent to the work in this thesis, Ahuja et al. [2001b] developed a class
of label setting algorithms that have the same running time as our algorithm. Also, Ergun [2001]
developed an algorithm for searching the cyclic exchange neighborhood using an approximate
dynamic programming formulation for the general partitioning problem.

The rest of this section is organized as follows. In Section 2.3.1, we provide a class of
heuristic methods based on modification of the classical label-correcting algorithm (Ahuja et al.
[1993]) for the SDCP. Our exact algorithms for the SDCP are covered in Section 3.2. These
algorithms include an integer programming formulation for SDCP, an enumeration algorithm
with search pruning criteria, and a dynamic programming algorithm for SDCP.

2.3.1 Label-correcting heuristic for SDCP

Our heuristic algorithm is a simple modification of the well-known label-correcting
algorithms for the shortest path problem (Ahuja, Magnanti, and Orlin [1993]). A label-correcting
algorithm determines a shortest path from a specified node $s$ to every other node in the network.
A label-correcting algorithm maintains two indices with each node $j$: $d(j)$, the distance label of
the node $j$, and $\text{pred}(j)$, the predecessor index of node $j$. The distance label $d(j)$ is either $\infty$,
indicating that the algorithm has yet to discover a directed path from node $s$ to node $j$, or it is the
length of some directed path from the node $s$ to node $j$. The predecessor index, $\text{pred}(j)$, records the
node prior to node $j$ in the current directed path of length $d(j)$. The predecessor indices allow us to
trace the current shortest path from node $j$ back to node $s$. Let $P[j]$ denote the current directed path
from node $s$ to node $j$ as given by the indices $\text{pred}(\cdot)$. The optimality conditions for the shortest
path problem require that $d(j) \leq d(i) + c_{ij}$ for each arc $(i, j)$ in the network.

A label correcting algorithm starts with some node $s$ as the starting node, sets $d(s) = 0$
and $d(j) = \infty$ for all $i \in N \setminus \{s\}$. The basic step in a label-correcting algorithm is to identify an arc
$(i, j)$ violating its optimality condition, that is, $d(j) > d(i) + c_{ij}$, and decrease the distance label $d(j)$
to $d(i) + c_{ij}$; this step is called the *distance update* step. The algorithm repeatedly performs distance update steps and terminates when all the arcs satisfy their optimality conditions. To efficiently identify an arc $(i, j)$ violating its optimality condition, the algorithm maintains a list, LIST, of nodes with the property that if an arc $(i, j)$ violates its optimality conditions then LIST must contain node $i$ (Possibly LIST contains other nodes as well). In each iteration, the algorithm selects a node $i$ from LIST, removes it from LIST, and examines it by performing a distance update step for all arcs emanating from node $i$.

To identify negative cost subset disjoint cycles in the improvement graph, we try to enforce the property that for each node $i$ such that $d(i)$ is finite, the path $P[i]$ is subset disjoint. The subset disjoint property is defined for path in a manner similar to that for cycles, that is, a path $P = i_1 - i_2 \ldots i_\ell$ is called subset disjoint if the elements corresponding to the nodes $\{i_1, i_2, \ldots, i_\ell\}$ belong to different subsets in $S$. This property is difficult to enforce efficiently for all nodes and at all steps of the algorithm; however, we try to enforce it heuristically as much as we can. To accomplish this, we take the following additional steps. While executing the label-correcting algorithm, whenever we remove a node $i$ from LIST, we check whether its current path $P[i]$ is a subset-disjoint path. If not, then we do not examine node $i$ and set $d(i)$ to $\infty$; otherwise we set $d(i)$ equal to the length of the path $P[i]$ and examine arcs emanating from it one by one. While examining the arc $(i, j)$, we check whether $d(j) \leq d(i) + c_{ij}$. If this condition is satisfied, we skip the arc $(i, j)$ and examine another arc emanating from node $i$. If $d(j) > d(i) + c_{ij}$, then there are three cases to consider:

1. $j \in P[i]$. In this case, we have discovered a subset-disjoint cycle: the subpath from $j$ to $i$ in $P[i]$ combined with the edge $(i, j)$. Since $d(j) > d(i) + c_{ij}$, it is easy to see that this cycle has a negative cost. In this case, we can return the cycle as a negative cost subset disjoint cycle.

2. $j \not\in P[i]$ and there is no node corresponding to elements in $S[a_j]$ is in $P[i]$. In this case, adding arc $(i, j)$ to $P[i]$ creates a subset-disjoint path $P[i] \cup \{(i, j)\}$. We update $d(j) = d(i) + c_{ij}$, add node $j$ to LIST, and continue the label-correcting algorithm.

3. $j \not\in P[i]$ and there is some node corresponding to an element from $S[a_j]$ in $P[i]$. In this case, adding arc $(i, j)$ to $P[i]$ does not create a subset-disjoint path. We do not update $d(j)$ and continue the label-correcting algorithm.

We present a formal statement of our adaptation of the label-correcting algorithm in Figure 2-3.
algorithm SDCP-heuristic;
begin
1. set d(s) ← 0 and pred(s) ← s;
2. set d(j) ← ∞ for each node j ∈ N\{s};
3. LIST ← {s};
4. while LIST ≠ s do
5. begin
6. remove an element i from LIST;
7. if P[i] is not subset disjoint then
8. go back to the beginning of the while loop at line 4
9. for each arc (i, j) ∈ A do
10. if d(j) > d(i) + c_{ij} then
11. begin
12. if j is in P[i] then
13. return the negative cost subset disjoint cycle consisting of the subpath from j to i in P[i] and the arc (i, j)
14. else
15. begin
16. if no node corresponding to elements in S[a_j] is in P[i] then
17. set d(j) ← d(i) + c_{ij}, pred(j) ← i, and add node j to LIST;
18. end;
19. end;
20. end;
end.

Figure 2-3. The subset disjoint cycle heuristic based on label-correcting algorithm.

We note that the algorithm is not completely specified as the manner in which the nodes are added or deleted from LIST can drastically affect the running time. Two of the popular implementations of the LIST are the first-in-first-out (FIFO) and the deque implementation of Pape [1980]. In the case of the FIFO label-correcting algorithm the algorithm can be implemented to run in time O(K(Kn+m)). The primary observation to obtain this running time is that each node is examined at most K times during the algorithm. During each examination, we can spend O(K) time for preprocessing before the examination of any arc (i, j) so that the step of checking whether a node j is in P[i] can be implemented in O(1). Since there are n nodes and each one can be examined at most K times, O(K^2n) time is spent in this preprocessing. The other bottleneck step is the testing of optimality of arc (i, j) when a node i is examined. The total effort spent on this operation is O(mK) as each arc is tested at most K times.

In the case of deque implementation, the algorithm above is not guaranteed to run in polynomial time, but may perform very well in practice. We tested this implementation and found that it runs very quickly.

We apply the modified label-correcting algorithm, as described above, with some regular node as node s. The modified label-correcting algorithm during its execution can discover several
profitable cyclic or path exchanges or none of them. The algorithm above selects the first negative cost subset disjoint cycle. We can easily modify it to run to completion and let it select the minimum (possibly negative) cost subset disjoint cycle in the improvement graph.

Our empirical investigations revealed that the success of the modified label-correcting algorithm in finding profitable exchanges depends upon which regular node is used as node $s$. If we apply it just once with some regular node as node $s$, we miss many profitable exchanges. We thus applied the modified label-correcting algorithm once for each node as the starting node. The running time of our method for finding profitable exchanges is $n$ times the time taken by a single execution of the modified label-correcting algorithm. Since we may apply the label-correcting algorithm as many as $n$ times (one per node), the total time taken by the algorithm is $O(n^3K^2+nmK)$ in the case of FIFO implementation of LIST.

2.3.2 Exact Algorithms for SDCP

In this section, we present some exact algorithms for the SDCP. We consider three of the commonly used tools in math programming: (i) integer programming, (ii) enumeration, and (iii) dynamic programming, to develop these exact algorithms. We first formulate the SDCP as a 0/1 min cost flow problem with side constraints. After that, we consider an explicit enumeration technique that enumerates subset disjoint paths and converts them into subset disjoint cycles. We describe a pruning criterion, using which the number of paths generated by the algorithm is usually very small in practice. We also provide a preprocessing technique for SDCP, which also reduces the number of paths in some special cases. Lastly, we solve the SDCP using a dynamic programming formulation which has a worst-case running time of $O(nmK2^K)$. A nice feature of this approach is that the running time is only exponential in the number of partitions in the current solution. Therefore, the approach is certainly useful for problems with small number of partitions. We propose methods for implementing it efficiently in practice.

**Integer Programming Formulation**

In this section, we present an integer programming formulation that can be used to find negative cost subset disjoint cycles in the improvement graph. Our integer program is actually a 0/1 min cost flow problem with side constraints. We introduce a binary flow variable $x_{ij}$ for each arc $(i,j) \in A$. The variable $x_{ij}$ takes a value of 1 if the arc $(i,j)$ is included in a subset disjoint cycle and 0 otherwise. Using these variables, we present the formulation below.

$$\text{Minimize} \quad \sum_{(i,j) \in A} c_{ij}x_{ij}$$ 

(2.2a)
Subject to

$$\sum_{(i,j) \in A} x_{ij} - \sum_{(j,i) \in A} x_{ji} = 0 \quad \text{for each } i \in N$$ (2.2b)

$$\sum_{i \in S_k} \sum_{(i,j) \in A} x_{ij} \leq 1 \quad \text{for } k = 1, \ldots, K$$ (2.2c)

$$x_{ij} \in \{0, 1\} \text{ for } (i, j) \in A$$ (2.2d)

We note that the constraints (2.2b) in the formulation 2.2 represent flow balance constraints for each node. Therefore, any feasible solution to this formulation will be a 0-1 circulation in the network G. The constraints (2.2c) ensure that the circulation contains at most one arc going out of any subset $S_k$. It is easy to see that any feasible circulation for the formulation is a set of subset disjoint cycles such that no two cycles in the set are incident on the same subset. If the improvement graph G contains a negative cost subset disjoint cycle then an optimal circulation for the formulation would contain a negative cost subset disjoint cycle. Using flow decomposition (Ahuja, Magnanti, Orlin [1993]), the circulation can be easily separated into cycles to identify a negative cost subset disjoint cycle. In case the improvement graph does not contain a negative cost subset disjoint cycle, the optimal solution will have objective value 0 because a flow of zero units in each arc satisfies all the constraints. In this case, we can add an additional constraint to force the flow to be non-zero:

$$\sum_{(i,j) \in A} x_{ij} \geq 1$$ (2.2e)

By adding the constraint (2.2e) into the formulation 2.2, we ensure that if there are no negative cost cycles in the improvement graph, the optimum solution to the formulation is a minimum cost non-empty subset disjoint cycle.

We found that the CPLEX6.5 solves the problem very effectively for a small number of nodes and partitions. However, most of our improvement graphs are quite dense which makes the size of formulation 2.1 large very quickly with number of nodes. The solution grow quickly with the increase in number of nodes. Another disadvantage of this approach is that we do not have any control over the way in which the neighborhood is searched. We have observed empirically that in many improving iterations of the neighborhood search algorithm using cyclic exchange there is a 2-exchange that can improve the current solution as well. One approach to search the cyclic exchange is to explore all the 2-exchanges first, followed by 3-exchanges, 4-exchanges, and so on. We propose an enumeration algorithm motivated by this observation in the next section.
Enumeration Approach

A simple way to search the cyclic exchange neighborhood is to explicitly enumerate each subset disjoint cycle in the improvement graph and compute its cost. Note that a subset disjoint path cycle \( W = i_1 - i_2 - \ldots - i_r - i_1 \) can be obtained by adding the arc \((i_r, i_1)\) to the subset disjoint path \( P = i_1 - i_2 - \ldots - i_r \). We use \( \text{Cycle}(P) \) to denote the cycle corresponding to path \( P \). We can enumerate subset disjoint cycles by enumerating all the subset disjoint paths. We now describe an algorithm to generate all the subset disjoint cycles by enumerating all the subset disjoint paths.

**Algorithm SDCP-Enumeration**;

begin
1. Let \( P^1 = \{(i, j) \in A : S[a_i] \neq S[a_j]\} \) denote the set of all subset disjoint paths of length 1;
2. \( k \leftarrow 1; \)
3. while \( P^k \neq \emptyset \) do
4. begin
5. \( P^{k+1} \leftarrow \emptyset; \)
6. for \( P \in P^k \) do
7. begin
8. if \( \text{Cycle}(P) \) is negative cost then return \( \text{Cycle}(P) \);
9. let \( P = i_1 - i_2 - \ldots - i_k \), set \( P^{k+1} \leftarrow P^{k+1} \cup \{(i_1 - i_2 - \ldots - i_k - j) : (i_l, j) \in A \text{ and } S[a_j] \neq S[a_{i_l}] \} \) for \( l = 1, \ldots, k \).
10. end;
11. end;
end.

Figure 2-4. The path enumeration algorithm for SDCP.

We start with \( P^1 \), the set of all subset disjoint paths of length one, that is, \( P^1 = \{(i, j) \in A : S[a_i] \neq S[a_j]\} \). We examine the subset disjoint cycles corresponding to each of these paths. If a negative cost subset disjoint cycle is found, we return it. Otherwise, for each path \( P = i_1 - i_2 \) we generate all the paths \( i_1 - i_2 - j \) such that \((i_2, j) \in A \text{ and } S[a_j] \neq S[a_{i_2}] \) for \( l = 1, 2 \). This creates all the \( P^2 \), the set of all subset disjoint paths of length 2. We now examine the cycles corresponding to each path in \( P^2 \). If a negative cost cycle is found, we return it, otherwise we generate \( P^3 \) from \( P^2 \) in the same way as we generated \( P^2 \) from \( P^1 \). The algorithm terminates if a negative cost cycle is found at any stage or if \( P^k \) is empty, which happens for \( k = K+1 \). The formal description of the algorithm is given in Figure 2-3.

Pruning Technique

Since we enumerate all the subset disjoint paths, the worst-case complexity of this algorithm is exponential and further it will certainly perform very badly in practice as well. However, we exploited the following result shown by Lin and Kernighan [1973] to develop a
path pruning mechanism for the previous algorithm so that it performs quite well in practice. We provide a proof here for completeness.

**Theorem 2.3.** (Lin and Kernighan [1973]) Given any negative cost directed cycle \( W = i_1 - i_2 - \ldots - i_k - i_1 \), there exists \( l \in \{1, \ldots, k\} \) such that \( C(P') < 0 \) for \( r = l, \ldots, k \), where \( P' = i_1 - i_{l+1} - \ldots - i_r \) is the subpath of \( W \) from \( i_l \) to \( i_r \).

**Proof:** Suppose that the node \( i_1 \) does not satisfy this property. Consider the index \( r^* \) such that the cost of the subpath \( i_1 - i_2 - \ldots - i_{r^*} \) is maximum among all subpaths \( i_1 - i_2 - \ldots - i_r \) for \( r = 2, \ldots, k \), and \( r^* \) is the largest index with this property. Note that the cost of cycle \( W \), \( C(W) \), is equal to the cost of subpath \( i_1 - i_2 - \ldots - i_{r^*} \) plus the cost of subpath \( i_{r^*} - i_{r^*+1} - \ldots - i_1 \). That is, \( C(i_1 - i_2 - \ldots - i_{r^*}) + C(i_{r^*} - i_{r^*+1} - \ldots - i_1) = C(W) < 0 \). Note that \( C(i_{r^*} - i_{r^*+1} - \ldots - i_j) \) must be strictly less than zero for all \( j = r^* + 1, \ldots, k \) because \( r^* \) was chosen to be the largest index such that the subpath \( i_1 - i_2 - \ldots - i_{r^*} \) has the maximum cost among all paths \( i_1 - i_2 - \ldots - i_r \). Further, since \( C(i_1 - i_2 - \ldots - i_r) \leq C(i_1 - i_2 - \ldots - i_{r^*}) \) for all \( r = 2, \ldots, r^*-1 \), \( C(i_1 - i_2 - \ldots - i_r) + \text{Cost}(i_{r^*} - i_{r^*+1} - \ldots - i_1) < 0 \). Therefore, all sub-paths \( i_{r^*} - i_{r^*+1} - \ldots - i_r \) have negative cost and \( l = r^* \) is the desired index.

The Theorem 2.3 can be interpreted as follows. For any negative cost cycle in a graph, there exists a node \( i \) in the cycle such that every subpath of the cycle starting at node \( i \) has negative cost. Note that if the cycle is a subset disjoint then the subpaths of the cycle are also subset disjoint. Since our enumeration algorithm enumerates subset disjoint paths, the Theorem 2.1 implies that we only need to consider negative cost subset disjoint paths \( P = i_1 - i_2 - \ldots - i_r \) such that each of its subpaths \( i_1 - i_2 - \ldots - i_{r'} \) for \( r' \leq r \) are also negative cost. Therefore, we can start with \( P^1 \) as the set of negative cost paths of length 1, that is, \( P^1 = \{(i, j) \in A: S[a_i] \neq S[a_j] \text{ and } c_{ij} < 0\} \). In general, if \( P = i_1 - i_2 - \ldots - i_k \) is a negative cost subset disjoint path in \( P^k \) then we only add a subset disjoint paths \( i_1 - i_2 - \ldots - i_k - j \) to \( P^{k+1} \) only if it has a negative cost, all the other paths can be discarded. This observation substantially reduces the number of paths generated by the algorithm in practice. However, the worst-case complexity of the algorithm is not improved by this modification.

**Preprocessing the Improvement Graph**

One of the problems that we encountered implementing this enumeration method in practice was that in some cases the improvement graph had a few arcs with very large negative costs but no negative cost subset disjoint cycles through them. Any subset disjoint path starting at
one of these large negative cost edges is likely to be negative as well. Therefore, our enumeration algorithm was generating a large number of negative cost subset disjoint paths without finding a negative cost subset disjoint cycle. In order to tackle this problem, we used a preprocessing routine to modify the arc costs in the improvement graph so that all the costs are small in magnitude and the set of negative cost cycles in the improvement graph is not modified.

It is known (Ahuja, Magnanti, Orlin [1993]) that if we are given a set of node potentials $\pi_i$ for $i \in N$, then modifying the cost of each arc $(i, j)$ from $c_{ij}$ to $c_{ij} - \pi_i + \pi_j$ does not change the cost of any cycle in the graph. We use this property to preprocess the arc costs in the improvement graph by finding a set of node potentials such that the magnitude of the most negative arc in the improvement graph is minimum. We setup a linear optimization problem: 
Maximize $\{\lambda : \lambda \leq c_{ij} - \pi_i + \pi_j \text{ for } (i, j) \in A\}$ to determine the value of the node potentials to achieve our goal. It is easy to see that this optimization problem is the well-known minimum mean cycle problem and there are many efficient algorithms to solve it (Ahuja, Magnanti, Orlin [1993]).

\begin{figure}[h]
\centering
\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node (1) at (0,0) {$1$};
  \node (2) at (1,0) {$2$};
  \draw[->, line width=2pt] (1) to [out=180, in=270] node [midway, below] {0} (2);
  \draw[->, line width=2pt] (2) to [out=90, in=0] node [midway, above] {-2} (1);
\end{tikzpicture}
\caption{(a)}
\end{subfigure}
\begin{subfigure}{0.4\textwidth}
\centering
\begin{tikzpicture}
  \node (1) at (0,0) {$1$};
  \node (2) at (1,0) {$2$};
  \draw[->, line width=2pt] (1) to [out=180, in=270] node [midway, below] {-1} (2);
  \draw[->, line width=2pt] (2) to [out=90, in=0] node [midway, above] {-1} (1);
\end{tikzpicture}
\caption{(b)}
\end{subfigure}
\caption{A network with two nodes before and after preprocessing.}
\end{figure}

We used the preprocessing mentioned in the previous paragraph to reduce the number of paths enumerated by the algorithm. However, this processing technique is not always guaranteed to reduce the number of negative paths in a graph. We illustrate this through the example in Figure 2-5. Note that using the initial costs in Figure 2-5(a), there is only one negative cost path in the graph, namely 1 → 2. However, if we use the optimal node potentials $\pi_1 = 0$ and $\pi_2 = 1$ the arc costs in the network are changed to those in Figure 2-5(b). Clearly, there are two paths of negative cost in Figure 2-5(b).

Although the enumeration approach works efficiently for small problems, the worst-case complexity of this algorithm is quite bad. Its performance deteriorates quickly as the number of nodes or the number of subsets in the problem increases. We now describe a dynamic programming based approach that has some nice theoretical as well as practical properties.
Dynamic Programming Approach

Our dynamic programming formulation is based on the following observation. Consider the two subset disjoint paths: 1 - 2 - 3 - 4 and 1 - 5 - 6 - 4 in the improvement graph given in Figure 2-6. Both these paths contain a node from set $S_2$ and $S_3$. Since the cost of the path 1 - 5 - 6 - 4 is less than the cost of the path 1 - 2 - 3 - 4, for every subset disjoint cycle containing the second path, we can replace the second path by the first path and get a subset disjoint cycle of lower cost. Therefore, in our enumeration algorithm in the previous section, we do not need to enumerate the path 1 - 2 - 3 - 4. In fact any subset disjoint path from node 1 to node 4 containing a node from $S_2$ and $S_3$ with cost greater than or equal to -5 does not need to be evaluated. Therefore, if we know the value of the minimum cost subset disjoint path from node 1 to node 4 going through subsets $S_2$ and $S_3$, we can eliminate all the paths with costs greater than or equal to that value. Using this observation, we create a dynamic programming formulation to find the minimum cost subset disjoint cycle in the improvement graph as follows.

![Graph](image)

**Figure 2-6. An improvement graph with two paths**

Our DP formulation has $K$ stages. We use $X^k$ to denote the set of states at the $k$th stage in our formulation. For $k = 0, \ldots, K-1$, we define $X^k$ as:

$$X^k = \{(i, j, L) : i, j \in \mathbb{N}, S[a_i] \neq S[a_j], L \subseteq S \setminus \{S[a_i], S[a_j]\} \text{ where } |L| = k\}$$  \hspace{1cm} (2.3)

A state $(i, j, L)$ represents a minimum cost subset disjoint path from node $i$ to node $j$ in $G(S)$ such that the path contains exactly one node from each of the subsets in set $L$. We associate with a value $f(i, j, L)$ with each state $(i, j, L)$. This value is the cost of the minimum cost subset disjoint path represented by the state. We start with the states in $(i, j, \emptyset) \in X^0$, which represent arcs in the
improvement graph as $L = \emptyset$. We set the value of a state $(i, j, \emptyset)$ to $c_{ij}$ if $(i, j) \in A$, and $\infty$ otherwise. Given that we have already calculated the value for all the states in $X^k$, we calculate the value function for a state $(i, j, L) \in X^{k+1}$ using the following procedure:

procedure forward-recursion;
begin
1. for $(i, j, L) \in X^{k+1}$ do
2. $f(i, j, L) \leftarrow \infty$;
3. for $(i, j, L) \in X^k$ do
   begin
5. for $(j', j) \in A$ do
6. if $S[a_{j'}] \notin L$ and $f(i, j', L \cup \{S[a_{j'}]\}) > f(i, j, L) + c_{jj'}$ then
7. $f(i, j', L \cup \{S[a_{j'}]\}) \leftarrow f(i, j, L) + c_{jj'}$;
   end;
end.

Figure 2-7. Forward dynamic programming recursion.

It is easy to see that the forward recursion computes the value of each state $(i, j, L)$ is calculated correctly. We note that as we calculate the value of each state $(i, j, L)$, we can also check whether $f(i, j, L) + c_{jj} < 0$ to check whether the minimum cost path represented by the state $(i, j, L)$ leads to a negative cost subset disjoint cycle. Therefore, after calculating $X^0$, we can determine if there is a negative cost subset disjoint cycle of length 2, and in general, after computing $X^k$, we can determine if the improvement graph has a negative cost subset disjoint cycle of length $k+2$. We now perform the worst-case run time analysis of this dynamic programming recursion.

Theorem 2.4. The number of states in the DP formulation is $O(n^22^K)$ and the total time required to compute the value of all the states using forward-recursion is $O(nmK2^K)$.

Proof. The total number of states $(i, j, L)$ in our formulation is bounded by $O(n^22^K)$ as $i$ and $j$ can each take $n$ different values and the number of possible subsets $L$ is bounded by $2^K$. Note that for a given arc $(j, j')$, the statements in lines 6-7 are performed exactly once for each state of the form $(i, j, L)$. Since there are at most $n2^K$ states containing node $j$, the statements 6 and 7 are performed for arc $(j, j')$ at most $n2^K$ times. Each execution of these statements takes $O(K)$ time as the size of the state $(i, j, L)$ is $O(K)$. Hence the running time of the DP recursion is $O(nmK2^K)$.

Each state in our DP formulation implicitly represents a path. In particular, the states in $X^k$ correspond to certain paths in $P^k$ from section 3.2.2. Recall that we used Theorem 2.3 in Section 2.3.2 to show that we only need to extend those paths in $P^k$ that have negative cost to obtain negative cost subset disjoint cycles. Correspondingly, in our DP forward recursion, while
performing the forward recursion from $X^k$ for $k = 0, \ldots, K-2$, we only consider states that have a negative value, the rest of the states can be discarded.

Using this state pruning technique, the number of states whose value is actually computed may be much smaller compared to the total number of states. However, the procedure forward-recursion has to initialize the value of each state to $\infty$, hence it must spend at least time $O(n^2 2^K)$. One way to avoid this overhead is to use hashing tables (Aho et al. [1983]) to store the values for only those states that have value less than zero. This can be done as follows. Instead of initializing the value of each state in the beginning of procedure forward-recursion, we create an empty hash table. Each time we need to check the value of a state in $X^{k+1}$, we look for it in the hash table. If the value is present in the hash table, we use it. Otherwise, we know that the current value of the state must be greater than zero. In this case, if the new value of the state is less than zero, we add it to the hash table. Since each value in the hash table can be accessed in $O(K)$ average time, the average running time of the forward-recursion using hash table is the same as its worst-case running time without using the hash table. In practice, the use of hash table substantially improves the running time of this approach.

We now consider application of the cyclic exchange to a specific partitioning problem called the capacitated minimum spanning tree problem. We propose new neighborhood search algorithms for this problem based on cyclic exchange neighborhood structure. Our computational results indicate that the search algorithms based on cyclic exchange are superior compared to the algorithms based 2-exchange and searching the cyclic exchange neighborhood takes about the same time as that of searching a 2-exchange.

### 2.4 Capacitated Minimum Spanning Tree Problem

The capacitated minimum spanning tree (CMST) problem is a fundamental problem in the telecommunication network design. In this problem, we are given a central processor, represented by node 0, a set of $n$ terminals $V = \{1, \ldots, n\}$ with $d_i$ representing the demand of node $i \in V$, a set of undirected edges $E$ representing the possible connections between nodes, and a weight $w_{ij}$ for each edge $(i, j) \in E$ representing the cost of construction of the link. The capacitated minimum spanning tree problem is to construct a spanning tree connecting the terminal with the central processor so that the traffic on any arc of the network is at most $Q$ and the total construction cost is minimum. Observe that the flow on any such arc $(i, 0)$ will be equal to the total demand of the nodes in the tree rooted at node $i$, since this arc is used to satisfy the demands of all nodes in that tree. In view of this observation, any feasible solution of the
capacitated minimum spanning tree problem will be a partition of the node set \( N \) into \( K \) subsets \( S_1, S_2, \ldots, S_K \) satisfying \( \sum_{i \in S_k} d_i \leq Q \) for each \( k, 1 \leq k \leq K \). Let \( T_k \) denote a minimum (cost) spanning tree over the node subset \( S_k \cup \{0\} \) for each \( k, 1 \leq k \leq K \), and \( T \) denote the union of these trees. The capacitated minimum spanning tree problem is to identify the node partitions \( S_1, S_2, \ldots, S_K \) so that the resulting tree \( T \) has the minimum possible cost. There are two versions of the problems: when \( d_i = d_j \) for all \( i \) and \( j \) (the homogenous demand case), and when \( d_i \neq d_j \) for some \( i \) and \( j \) (the heterogeneous demand case).

We refer to the special node 0 as the source node. We assume that arcs in the tree \( T \) denote the parent-child relationship, the node closer to the source node being the parent of the node farther from the source node. For each node \( i \) in \( T \), we denote by \( T_i \) the subtree of \( T \) rooted at node \( i \). We denote by \( D_i \) the set of descendants of node \( i \), that is, children of node \( i \), children of their children, and so on. We refer to the children of the source nodes as root nodes and the trees rooted at root nodes as rooted subtrees. We will often refer to the rooted subtrees simply as subtrees, when the rooted subtree will be obvious from the context. For any node \( i \), we denote by \( T[i] \) the rooted subtree containing node \( i \). We denote by \( S[i] \) the set of nodes contained in the subtree \( T[i] \). For a subset \( S \) of nodes, we let \( d(S) = \sum_{i \in S} d_i \). We say that the subset \( S \) is feasible if and only if \( d(S) \leq Q \). If \( S \) is feasible, we denote by \( c(S) \) the cost of a minimum cost tree spanning the node set \( S \cup \{0\} \).

![Figure 2-8](image.png)

Figure 2-8. An example of a capacitated spanning tree where each node has unit demand.
We illustrate these definitions using Figure 2-8, where we partition V into three subsets
S_1 = \{1, 4, 5, 10, 11, 15\}, S_2 = \{2, 6, 7, 12, 16\}, and S_3 = \{3, 8, 9, 13, 14\}. We assume that each
node has unit demand and Q = 6. The minimum spanning trees over these node subsets are as
shown. The cost of the tree T is 104. In Figure 2-8, T_5 denotes the subtree rooted at node 5 in
T[1] and D_5 = \{5, 11, 15\}. For each node i = 4, 5, 10, 11, and 15, T[i] = T_1 and S[i] = \{1, 4, 5, 10,
11, 15\}.

There is a substantial research literature devoted to the capacitated minimum spanning
tree problem. The survey paper by Gavish [1991] gives a detailed understanding of
telecommunication design problems where the capacitated minimum spanning tree problem
arises. The paper by Amberg et al. [1996] presents an excellent survey of exact and heuristic
algorithms for the CMST problem. Typically, exact algorithms can solve problems of size at most
50 nodes. A cutting plane algorithm by Hall [1996], and Lagrangian relaxation based approaches
by Gouveia [1998], and Gouveia and Martins [1999, 2000] obtain good solutions for the
capacitated minimum spanning tree problem and give linear programming based upper bounds on
the percentage error. Amberg et al. [1996] and Sharaiha et al. [1997] have developed tabu search
algorithms for solving the capacitated minimum spanning tree problem.

Among the best available neighborhood search algorithms for CMST are those of
Amberg et al. [1996] and Sharaiha et al. [1997]. Amberg et al. use a neighborhood structure that
is based on exchanging single nodes between two subtrees. The nodes can be anywhere in the
subtrees and may not always be the leaf nodes of the subtrees. For example, in Figure 2-8,
exchanging nodes 11 and 12 between T_1 and T_2 gives a neighbor of the solution shown. The
neighborhood structure due to Sharaiha et al. moves a part of a subtree, from one subtree to
another subtree or to the root node. For example, in Figure 2-8, we can obtain a neighbor by
deleting the node subset \{5, 11, 15\} from T_1 and attaching it to the root node. The number of
neighbors in both of these neighborhood structures is no more that n^2. Both papers report
computational results of tabu search methods based on their neighborhood structures. Amberg et
al. report results on the unit-demand instances and Sharaiha et al. report results on both unit-
demand and heterogeneous demand benchmark instances available at the OR-library website
(http://www.ms.ic.ac.uk/info.html). Amberg et al. obtained the best available solutions for all the
benchmark instances for the unit demand case and Sharaiha et al. obtained the best available
solutions for the heterogeneous demand case. We will subsequently refer to their neighborhood
structures as two-exchange neighborhood structures since a neighboring solution is obtained by
changing at most two subtrees.
In this section, we are interested in applying cyclic-exchange based neighborhood search algorithms to the capacitated minimum spanning tree problem. We introduce two cyclic-exchange neighborhood structures that can be regarded as generalizations of the above two-exchange neighborhood structures. We also introduce a composite cyclic-exchange neighborhood structure that is a generalization of our own neighborhood structures. We provide details on the computational issues in the implementation of these neighborhood structures. We performed computational results with these neighborhood structures on the set of benchmark instances at the OR-Library website. Our algorithms achieved, and in many cases, improved the previously best-known solutions for these instances. Our results also show that cyclic exchange based approaches perform much better than the 2-exchange based search approaches and the additional time to search the cyclic exchange neighborhood is not significant. We used our adaptation of the label-correcting algorithm and the dynamic programming based algorithm to search our proposed cyclic exchange neighborhood structures. We provide computational results showing that the adaptation of a label-correcting algorithm is very consistent in finding negative cost subset disjoint cycles and the dynamic programming based approach is quite robust even for problems with a large number of partitions.

2.4.1 Cyclic Exchange Neighborhood Structures for CMST

We consider three different cyclic exchange neighborhood structures for the CMST. These neighborhood structures are based on exchanging single nodes, and subtrees rooted at nodes between rooted trees. We define appropriate improvement graphs for each of the proposed neighborhood structures later in the section. We now give a description of each of the neighborhood structures.

**Node-Based Cyclic-Exchange Neighborhood**: This neighborhood structure is obtained by performing simple cyclic and path exchanges where we consider adding and/or removing a single node from each rooted subtree. A cyclic exchange \(i_1 - i_2 - \ldots - i_r - i_1\) represents the following changes: node \(i_1\) moves from the subtree \(T[i_1]\) to the subtree \(T[i_2]\), node \(i_2\) moves from the subtree \(T[i_2]\) to the subtree \(T[i_3]\), and so on, and finally node \(i_r\) moves from the subtree \(T[i_r]\) to the subtree \(T[i_1]\). A path exchange \(i_1 - i_2 - \ldots - i_r\) is similar to the cyclic exchange except that the last node \(i_r\) does not move from the subtree \(T[i_r]\) to the subtree \(T[i_1]\).

**Tree-Based Cyclic-Exchange Neighborhood**: This neighborhood structure is similar to node-based cyclic-exchange but we move subtrees instead of nodes. A cyclic exchange represented by the sequence of nodes \(i_1 - i_2 - \ldots - i_r - i_1\) with respect to the solution \(T\), represents the following
changes - nodes in the subtree $T_{i_1}$ move from the subtree $T[i_1]$ to the rooted $T[i_2]$, nodes in the subtree $T_{i_2}$ move from the rooted tree $T[i_2]$ to the rooted tree $T[i_3]$, and so on, and finally nodes in the subtree $T_{i_r}$ move from the rooted tree $T[i_r]$ to the rooted tree $T[i_1]$. Path exchanges are defined in a similar manner as for the node-based neighborhood structure.

**Composite Cyclic-Exchange Neighborhood:** A *composite cyclic exchange* is defined by a sequence of nodes $i_1 - i_2 - \ldots - i_r - i_1$, where the nodes $i_1, i_2, \ldots, i_r$ belong to different rooted trees, that is, $T[i_l] \neq T[i_{l'}]$ for $l \neq l'$. This exchange represents the following changes: either node $i_1$ or the subtree $T_{i_1}$ moves from the rooted tree $T[i_1]$ to the rooted tree $T[i_2]$, either node $i_2$ or the subtree $T_{i_2}$ moves from the rooted tree $T[i_2]$ to the rooted tree $T[i_3]$, and so on, and finally either node $i_r$ or the subtree $T_{i_r}$ moves from the rooted tree $T[i_r]$ to the rooted tree $T[i_1]$. Thus, for each node $i_l$ in the composite cyclic exchange, we allow either node $i_l$ to move or the subtree $T_{i_l}$ to move. A *composite path exchange* $i_1 - i_2 - \ldots - i_r$ can be defined in a similar fashion to the composite cyclic exchange with the difference that last node $i_r$ or the subtree $T_{i_r}$ does not move from the subtree $T[i_r]$ to the subtree $T[i_1]$. If we always choose node $i_l$ to move for each $l$, $1 \leq l \leq r$, then the resulting composite cyclic exchange reduces to a node-based cyclic exchange. If we always choose $T_{i_l}$ to move for each $l$, $1 \leq l \leq r$, then the resulting composite cyclic exchange reduces to a tree-based cyclic exchange. Thus, the composite cyclic exchange subsumes both the node-based and tree-based cyclic exchanges. In addition, it allows other possibilities that were not allowed in either of the previous neighborhoods. The previous neighborhoods did not allow node $i_l$ to move into the rooted tree $T[i_{l+1}]$ and replace the subtree $T_{i_{l+1}}$; it also did not allow the subtree $T_{i_l}$ to move into the rooted tree $T[i_{l+1}]$ and replace the node $i_{l+1}$. Both these possibilities are permitted in the composite cyclic exchange. In fact, the number of composite exchange neighbors of solution may be as much as $2^k$ times the number of node and tree exchange neighbors. However, we note that the improvement graph resulting for the composite exchange is also much bigger.

We use the improvement graph data structure to search each of our neighborhood structure. Since the construction of the improvement graph depends on the definition of the neighborhood structure, we next describe the construction of each of our neighborhood structures. We observed in Section 2.2.1 that the path exchanges can be easily transformed into cyclic exchanges by adding an extra node to each subset in the current solution. In the case of CMST,
we can add a node \( t_k \) to each subtree \( T_k \) such that \( d_{t_k} = 0 \) and this node only has edges with weight 0 to each of the root nodes. Note that this ensures that the node \( t_k \) can be added to any rooted subtree without affecting its cost and feasibility. In order to simplify the explanation, we assume that this transformation has already been made to the solution and only show the construction of improvement graphs to identify cyclic exchanges.

**Node-Based Improvement Graph:** The improvement graph \( G^1(\mathbf{T}) = (N^1, A^1) \) for the node-based cyclic exchange neighborhood is constructed as follows. The set of nodes in the graph \( G^1 \) is the same as the set of terminals, that is \( N^1 = V \). To construct the improvement graph, we consider every pair \( i \) and \( j \) of nodes in \( N^1 \), and add arc \((i, j)\) if and only if (i) \( T[i] \neq T[j] \), and (ii) \( \{i\} \cup S[j] \setminus \{j\} \) is a feasible subset of nodes (that is, \( d(\{i\} \cup S[j] \setminus \{j\}) \leq Q \)). We define the cost \( c_{ij}^1 \) of arc \((i, j)\) as \( c_{ij}^1 = c(\{i\} \cup S[j] \setminus \{j\}) - c(S[j]) \).

**Tree-Based Improvement Graph:** The improvement graph \( G^2(\mathbf{T}) = (N^2, A^2) \) for the tree-based multi-exchange neighborhood is constructed as follows: We create a node in \( G^2(\mathbf{T}) \) for each node \( i \in V \). An arc \((i, j)\) in the improvement graph signifies that the subtree \( T_i \) leaves the rooted tree \( T[i] \) and joins the rooted tree \( T[j] \) and simultaneously the subtree \( T_j \) leaves the rooted tree \( T[j] \). To construct the improvement graph, we consider every pair \( i \) and \( j \) of nodes in \( N^2 \), and add arc \((i, j)\) if and only if (i) \( T[i] \neq T[j] \), and (ii) \( \{D_i\} \cup S[j] \setminus \{D_j\} \) is a feasible subset of nodes (that is, \( d(\{D_i\} \cup S[j] \setminus \{D_j\}) \leq Q \)). We define the cost \( c_{ij}^2 \) of arc \((i, j)\) as \( c_{ij}^2 = c(\{D_i\} \cup S[j] \setminus \{D_j\}) - c(S[j]) \).

**Composite Improvement Graph:** The composite improvement graph is an extension of the node-based and tree-based improvement graphs. It allows for the possibilities of exchanges given by node-based improvement graph \( G^1(\mathbf{T}) \) and the tree-based improvement graph \( G^2(\mathbf{T}) \). In addition, it allows possibilities where a node replaces a subtree and a subtree replacing a node. We capture all these four possibilities by defining the composite improvement \( G^3(\mathbf{T}) \) in the following manner. For each node \( i \) in \( V \), we create two nodes in \( G^3(\mathbf{T}) \), \( i' \) and \( i'' \). The node \( i' \) represents the singleton node \( i \) and the node \( i'' \) represents the subtree \( T_i \) in \( \mathbf{T} \). For each pair of nodes \( i \) and \( j \) belonging to different rooted trees in \( \mathbf{T} \), we introduce as many as four arcs in the composite improvement graph as described below.

**Case 1:** \((i', j')\): This arc represents the situation when node \( i \) replaces the node \( j \) in the rooted tree \( T[j] \). We introduce this arc if \( d(\{i\} \cup S[j] \setminus \{j\}) \leq Q \). We define the cost of this arc as the change in the cost of the rooted tree represented by the arc \((i, j)\), that is, \( c_{ij}^3 = c(\{i\} \cup S[j] \setminus \{j\}) - c(S[j]) \).
**Case 2:** $(i'', j'')$: This arc represents the situation when the subtree $T_i$ replaces the subtree $T_j$ in the rooted tree $T[j]$. We introduce this arc if $d(D_i \cup S[j] \setminus D_j) \leq Q$. We define the cost of this arc as the change in the cost of the rooted tree represented by the arc $(i, j)$, that is, $c_{ij}^3 = c(D_i \cup S[j] \setminus D_j) - c(S[j])$.

**Case 3:** $(i', j')$: This arc represents the situation when node $i$ replaces the subtree $T_j$ in the rooted tree $T[j]$. We introduce this arc if $d(i \cup S[j] \setminus D_j) \leq Q$. We define the cost of this arc as $c_{ij}^3 = c(i \cup S[j] \setminus D_j) - c(S[j])$.

**Case 4:** $(i'', j')$: This arc represents the situation when the subtree $T_i$ replaces the node $j$ in the rooted tree $T[j]$. We introduce this arc when $d(D_i \cup S[j] \setminus \{j\}) \leq Q$. We define the cost of this arc as $c_{ij}^3 = c(D_i \cup S[j] \setminus \{j\}) - c(S[j])$.

The improvement graphs $G^1(T)$ and $G^2(T)$ each contain $n$ nodes and may contain as many as $n^2$ arcs. The improvement graph $G^3(T)$ has $2n$ nodes and may contain almost as many as $4n^2$ arcs. However, due to the capacity constraints, the number of arcs in practice may be substantially less than $4n^2$. For each possible arc in all three improvement graphs, its feasibility can be tested in $O(1)$ time using appropriate data structures, but computing its cost requires solving a minimum spanning tree problem. Let $U$ denote the maximum cardinality of a set $S_k$ for $k = 1, \ldots, K$. Then, computing the cost of any arc takes time $O(U^2)$. Solving these minimum spanning tree problems is the most time-consuming operation in the construction of all the improvement graphs. We note that our neighborhood search algorithm does not construct the entire improvement graph in each iteration. In fact, we only update the improvement graph in each iteration, and this typically takes far less time than constructing it from scratch. We next explain how to determine the costs of arcs in the improvement graph $G^1$ more efficiently.

**Computing arc costs in $G^1$ efficiently**

Consider an arc $(i, j)$ in $G^1$ and let $L = |S[j]|$. Note that the cost of the arc is given by $c_{ij}^1 = c((i) \cup S[j] \setminus \{j\}) - c(S[j])$. We note that the cost $c(S[j])$ is already known. Hence to determine $c_{ij}^1$, we need to compute $c((i) \cup S[j] \setminus \{j\})$. This involves deleting a node $j$ from $T[j]$ and adding node $i$ to it, and determining the minimum cost spanning tree on the set $(i) \cup \{0\} \cup S[j] \setminus \{j\}$. Note that if try to compute the minimum spanning tree from scratch, we need to consider $O(L^2)$ edges between the nodes in the set $(i) \cup \{0\} \cup S[j] \setminus \{j\}$. Therefore, computing the cost of arc $(i, j)$ from scratch takes $O(L^2)$ time using Prim's algorithm (Ahuja, Magnanti, Orlin [1993]). Using this
method, we can compute the cost of all arcs coming into node \( j \) in time \( O(nL^2) \). An alternative approach is to first compute the minimum spanning tree over the set \( \{ 0 \} \cup S[j] \setminus \{ j \} \). To solve a minimum spanning tree over the node set \( \{ i \} \cup \{ 0 \} \cup S[j] \setminus \{ j \} \), we need to consider the arc set that is the union of the current spanning tree over \( \{ 0 \} \cup S[j] \setminus \{ j \} \) and the arcs \( (i, h) \) for every \( h \in \{ 0 \} \cup S[j] \setminus \{ j \} \). This requires solving a spanning tree problem on a subnetwork with \( L+1 \) nodes and \( O(L) \) arcs and can be done in \( O(L \log L) \) time. Solving a minimum spanning tree over \( \{ 0 \} \cup S[j] \setminus \{ j \} \) takes \( O(L^2) \) time. Once this is done, we solve a minimum spanning tree over the node set \( \{ i \} \cup \{ 0 \} \cup S[j] \setminus \{ j \} \) for each node \( i \in V \setminus S[j] \) and determine the cost of the corresponding arc in the improvement graph. Hence we can determine the costs of all arcs entering a specific node \( j \) in \( O(L^2) + O((n-L)L \log L) = O(nL \log L) \) time, and the costs of all arcs in \( O(n^2U \log U) \) time. If we do some additional preprocessing, where we sort the set of edges incident on a node \( i \) in each set \( S_k \) in separate buckets, then we can compute the cost of the minimum spanning tree over the set \( \{ i \} \cup \{ 0 \} \cup S[j] \setminus \{ j \} \) in time \( O(L) \), and compute the cost of all the arcs in \( O(n^2L) \) time. The preprocessing itself takes time \( O(n^2 \log U) \). Hence, the total time to compute the arc costs in the improvement graph is \( O(n^2U) \).

We note that this is an example of a cyclic exchange neighborhood structure where computing the cost of each arc from scratch is less efficient compared to computing the cost of all the arcs coming into a node.

### 2.4.2 Neighborhood Search Algorithm

We describe in Figure 2-9 the VLSN search algorithm for the CMST problem. This is a generic statement of the neighborhood search algorithm using any cyclic exchange neighborhood structure for the CMST. The algorithm starts with a feasible solution of the CMST problem and successively improves it using a collection of cyclic exchanges until the solution is locally optimal.

```
algorithm Local Improvement;
begin
1. obtain a feasible solution \( T \);
2. construct the improvement graph;
3. while improvement graph contains a negative cost subset disjoint cycle do
4. begin
5. obtain a negative cost subset disjoint cycle \( W \) in the improvement graph;
6. use the cyclic-exchange corresponding to \( W \) to update the solution \( T \);
7. update the improvement graph;
8. end;
end.
```

Figure 2-9. The VLSN search algorithm for the CMST problem.
We point out that in each while loop of the above neighborhood search algorithm, we update the improvement graph instead of computing it afresh. If the cyclic exchange performed during an iteration changes a subset \( S_k \), then we need to update the arcs in the improvement graph entering and leaving all nodes corresponding to elements in \( S_k \). Depending upon the number of subsets involved in the cyclic exchange, updating is generally much faster than computing the improvement graph afresh.

### 2.4.3 Generating an Initial Feasible Solution

In our neighborhood search algorithms for the CMST, we used a randomized version of the Esau-William's [1966] algorithm to generate the initial feasible solution. The algorithm by Esau-William is one of the most popular construction heuristics for CMST. The Esau-William algorithm starts with each subtree containing a singleton node. In each iteration, the algorithm joins two subtrees into a single subtree so that the new subtree satisfies the capacity constraints and the savings achieved by the join operation are maximum. In this randomized version of Esau-William algorithm, at each iteration we determine the \( p \) most profitable join operations for some small value \( p \). We then generate an integer random number \( k \) uniformly distributed between 1 and \( p \) and perform the \( k^{th} \) most profitable join operation. This method in general provides a new feasible tree each time it is applied. Since at each step it performs one of the \( p \) most profitable join operations, the feasible tree obtained is generally a good tree. In our investigations, we used \( p = 3 \). We note that in the solution produced by the Esau-William's algorithm, a rooted subtree \( T_k \) may not be the minimum spanning tree over the set \( S_k \). Therefore, we compute the minimum spanning tree over \( S_k \cup \{0\} \) for each \( k = 1,\ldots,K \) after obtaining the solution from Esau-William's algorithm.

### 2.4.4 Computational Testing

In this section, we present the results of our computational investigation of the three proposed neighborhood structures. The testing is broken into two parts. In the first part, we investigate our neighborhood search algorithms based on node-exchange and the tree-exchange neighborhoods to identify the properties of the two kinds of exchanges. In the second part, we perform computational testing of the algorithm based on the composite neighborhood. We used our adaptation of the label-correcting algorithm to find negative cost subset disjoint cycles in the improvement graph during our first investigation. We use the dynamic programming based algorithm in our second investigation. We performed some tests to check the efficacy of these algorithms, which are also reported in this section.

47
We implemented a greedy randomized adaptive search procedure (GRASP). A GRASP is a search algorithm that applies a local improvement algorithm many times starting at different feasible solutions where starting feasible solutions are generated using some greedy randomized scheme. In our computational testing, we used the randomized version of Easu-William's algorithm as the greedy randomized scheme to generate initial solutions. We refer the reader to the papers by Feo and Resende [1995] and Festa and Resende [2000] for further details on GRASP. We refer to the GRASP procedures corresponding to the node-exchange, tree-exchange, and composite neighborhood structures as GRASP1, GRASP2, and GRASP3 respectively.

We tested our algorithms on three classes of benchmark problems given below, these problems are available at the website with URL: http://www.ms.ic.ac.uk/info.html.

**tc problem class:** These are randomly generated problems where the nodes are uniformly generated in a square grid. Each node has a unit demand. The weight of the edge between two nodes is linearly related to the Euclidean distance between the nodes. The problems in this set have two sizes: 40 and 80 nodes in the set V (excluding node 0). The edge capacities for these instances are Q = 5, 10, and 20. Lastly, the source node is located in the center of the grid. There are 35 problems in this class.

**te problem class:** These problems are generated in a similar manner as the tc problem class except that the source node is in the corner of the grid instead of the center. This problem class is considered to be somewhat harder than the tc problem class. There are 25 problems in this class.

**cm problem class:** These are also randomly generated problems. In this case, the nodes have heterogeneous demands and these are generated uniformly at random from 0 to 100. The weights of the edges between nodes are generated randomly between 0 to 100. The problems in this set have 50, 100, and 200 nodes. The edge capacities for these instances are Q = 200, 400, and 800. There are 45 problems in this class.

**Performance of Node-Exchange and Tree-Exchange based algorithms**

We applied our two algorithms, GRASP1 and GRASP2, corresponding to the node-exchange and tree-exchange based neighborhood structures to all the above benchmark problems that contain at least 50 nodes. For all the problems with 40 nodes, our algorithms obtained the optimal solutions. We applied our algorithms for 600 seconds for problems with fewer than 100 nodes and for 1200 seconds for problems with at least 100 nodes. We conducted the tests on a Pentium 1.4 GHz machine with 512 MB RAM and Linux operating system. We compared the best objective function value obtained by these algorithms with the best previously available
objective function values for the tested benchmark instances. We give in Tables 2-1 and 2-2, the percent deviations from the best available solutions for GRASP1 and GRASP2. The starred values in Tables 2-1 and 2-2 are known to be optimal solutions of the corresponding benchmark solutions. Our principle findings are the following:

1. For the tc and te problem classes, GRASP1, the local improvement algorithm using the node-based neighborhood structure, obtains better solutions on average compared to GRASP2.

2. For the cm problem class, GRASP2, the local improvement algorithm using the tree-based neighborhood search algorithm improves almost all the previous best available solutions. The average improvement is around 3% and the maximum improvement is around 18%.

3. We find that the two neighborhood structures have different strengths. The node-based neighborhood structure is quite effective for solving unit demand problems, and the tree-based neighborhood structure is very effective in solving heterogeneous demand problems.

4. For those problems, whose optimal solutions are known, we obtain their optimal solutions.

We have the following observations that may partially explain the relative performance of the two search algorithms. For the unit-demand case, exchanges of two nodes $i$ and $j$ belonging to two rooted subtrees are always possible but exchanges of two subtrees $T_i$ and $T_j$ may not be possible because the subtrees may contain a different number of nodes. Consequently, the neighborhood for GRASP1 is generally a larger neighborhood than that of GRASP2. Perhaps this explains in part why GRASP1 performs better than GRASP2. For the heterogeneous demand case, exchanges of two nodes are not always possible because nodes have different demands and subtrees may not have enough spare capacity to absorb additional demands. In order to add one node to a subtree to $T_i$ (say, of large demand), we may have to remove several nodes of smaller capacity from $T_i$. The neighborhood of GRASP2 admits these possibilities. Perhaps this explains in part why GRASP2 performs better than GRASP1 for the heterogeneous case.

We performed additional tests to compare the cyclic exchange neighborhood structure over the two-exchange neighborhood structure. We considered variations of our GRASP algorithms where we performed only two-exchanges and do not allow changes involving more than two subtrees. We give in Table 2-3, the results of these tests on 18 benchmark instances where we compare the quality of the local optimal solutions obtained by GRASP using two-exchange neighborhood and cyclic exchange neighborhoods. We applied GRASP1 to tc and te problems, and GRASP2 to CM problems. Recall that we perform multiple runs of GRASP with
different starting solutions and terminate each run when the algorithm finds a local optimal solution. We compare the average solution value, standard deviation of the local optimal solutions obtained by the procedures. We also note the average number of iterations performed to reach a local optimum, and average time per iteration for both the two-exchange and cyclic exchange neighborhood search algorithms. In the table, the columns corresponding to the heading of k-Ex give results of multi-exchange neighborhood search algorithms. It follows from the results in the table that for the multi-exchange neighborhood search algorithms, the average solution value is lower, the standard deviation is lower, the average number of iterations are higher, and the average time per iteration is comparable to the two-exchange neighborhood search algorithms. We thus observe that the multi-exchange neighborhoods do not significantly increase the running times of search algorithms per iteration, but they do improve the quality of the local optimal solution found by the algorithms. They take somewhat more time than two-exchange neighborhoods, but the increased running time is primarily due to the greater number of iterations rather than the greater time per iteration.

**Performance of the label-correcting heuristic**

We also performed some tests to determine the effectiveness of our heuristic based on label-correcting algorithm (from Section 2.3.1) to determine negative cost subset disjoint cycles. This method is not always guaranteed to find negative cost subset disjoint cycles when they exist in the improvement graphs. We used our heuristic method to find a cycle and whenever the method failed to find one, we applied the exact enumeration method from Section 3.2 to check whether there existed any negative cost subset disjoint cycle in the improvement graph. We performed this test on six benchmark instances and Table 2-4 below gives the results of this test. For each problem, the table shows the total number of iterations performed by the GRASP1 routine and the number of iterations in which there was an improving negative cost subset disjoint cycle that the heuristic failed to identify. The results show that the number of cycles missed is almost insignificant compared to the number of cycles found by our heuristic method.
<table>
<thead>
<tr>
<th>Problem-ID</th>
<th>Q</th>
<th>Previous best available solutions</th>
<th>GRASP1 % age deviation</th>
<th>GRASP2 % age deviation</th>
<th>New best available solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Te-1</td>
<td>5</td>
<td>1099*</td>
<td>0.00</td>
<td>0.82</td>
<td>1099</td>
</tr>
<tr>
<td>Te-2</td>
<td>5</td>
<td>1100*</td>
<td>0.18</td>
<td>1.00</td>
<td>1100</td>
</tr>
<tr>
<td>Te-3</td>
<td>5</td>
<td>1073*</td>
<td>0.00</td>
<td>1.30</td>
<td>1073</td>
</tr>
<tr>
<td>Te-4</td>
<td>5</td>
<td>1080*</td>
<td>0.00</td>
<td>1.02</td>
<td>1080</td>
</tr>
<tr>
<td>Te-5</td>
<td>5</td>
<td>1287*</td>
<td>0.08</td>
<td>0.62</td>
<td>1287</td>
</tr>
<tr>
<td>Te-1</td>
<td>10</td>
<td>888</td>
<td>0.00</td>
<td>0.00</td>
<td>888</td>
</tr>
<tr>
<td>Te-2</td>
<td>10</td>
<td>877</td>
<td>0.00</td>
<td>0.00</td>
<td>877</td>
</tr>
<tr>
<td>Te-3</td>
<td>10</td>
<td>880</td>
<td>-0.23</td>
<td>-0.23</td>
<td>878</td>
</tr>
<tr>
<td>Te-4</td>
<td>10</td>
<td>868</td>
<td>0.00</td>
<td>0.00</td>
<td>868</td>
</tr>
<tr>
<td>Te-5</td>
<td>10</td>
<td>1002*</td>
<td>0.00</td>
<td>0.00</td>
<td>1002</td>
</tr>
<tr>
<td>Te-1</td>
<td>20</td>
<td>834*</td>
<td>0.00</td>
<td>0.00</td>
<td>834</td>
</tr>
<tr>
<td>Te-2</td>
<td>20</td>
<td>820*</td>
<td>0.00</td>
<td>0.00</td>
<td>820</td>
</tr>
<tr>
<td>Te-3</td>
<td>20</td>
<td>828*</td>
<td>0.00</td>
<td>0.00</td>
<td>828</td>
</tr>
<tr>
<td>Te-4</td>
<td>20</td>
<td>820*</td>
<td>0.00</td>
<td>0.00</td>
<td>820</td>
</tr>
<tr>
<td>Te-5</td>
<td>20</td>
<td>916*</td>
<td>1.53</td>
<td>0.00</td>
<td>916</td>
</tr>
<tr>
<td>Te-1</td>
<td>5</td>
<td>2544*</td>
<td>0.00</td>
<td>0.35</td>
<td>2544</td>
</tr>
<tr>
<td>Te-2</td>
<td>5</td>
<td>2551</td>
<td>0.04</td>
<td>0.43</td>
<td>2551</td>
</tr>
<tr>
<td>Te-3</td>
<td>5</td>
<td>2612</td>
<td>0.00</td>
<td>1.00</td>
<td>2612</td>
</tr>
<tr>
<td>Te-4</td>
<td>5</td>
<td>2558</td>
<td>0.00</td>
<td>0.78</td>
<td>2558</td>
</tr>
<tr>
<td>Te-5</td>
<td>5</td>
<td>2469*</td>
<td>0.00</td>
<td>0.28</td>
<td>2469</td>
</tr>
<tr>
<td>Te-1</td>
<td>10</td>
<td>1657</td>
<td>0.00</td>
<td>0.48</td>
<td>1657</td>
</tr>
<tr>
<td>Te-2</td>
<td>10</td>
<td>1643</td>
<td>-0.24</td>
<td>0.73</td>
<td>1639</td>
</tr>
<tr>
<td>Te-3</td>
<td>10</td>
<td>1688</td>
<td>0.00</td>
<td>0.83</td>
<td>1688</td>
</tr>
<tr>
<td>Te-4</td>
<td>10</td>
<td>1629*</td>
<td>0.37</td>
<td>1.41</td>
<td>1629</td>
</tr>
<tr>
<td>Te-5</td>
<td>10</td>
<td>1603</td>
<td>0.56</td>
<td>0.81</td>
<td>1603</td>
</tr>
<tr>
<td>Te-1</td>
<td>20</td>
<td>1275</td>
<td>0.00</td>
<td>0.00</td>
<td>1275</td>
</tr>
<tr>
<td>Te-2</td>
<td>20</td>
<td>1225</td>
<td>0.08</td>
<td>0.08</td>
<td>1225</td>
</tr>
<tr>
<td>Te-3</td>
<td>20</td>
<td>1267</td>
<td>0.24</td>
<td>0.24</td>
<td>1267</td>
</tr>
<tr>
<td>Te-4</td>
<td>20</td>
<td>1265</td>
<td>0.00</td>
<td>0.00</td>
<td>1265</td>
</tr>
<tr>
<td>Te-5</td>
<td>20</td>
<td>1240</td>
<td>0.00</td>
<td>0.08</td>
<td>1240</td>
</tr>
<tr>
<td>Average deviation</td>
<td>0.09</td>
<td>0.40</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2-1. Comparison of node-exchange and tree-exchange for homogeneous demand problems.
<table>
<thead>
<tr>
<th>Problem-ID</th>
<th>Previous best available solutions</th>
<th>GRASP1 %age deviation</th>
<th>Number of runs</th>
<th>GRASP2 %age deviation</th>
<th>Number of runs</th>
<th>New best available solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM50R1Q200</td>
<td>1135</td>
<td>-3.26</td>
<td>-3.26</td>
<td>-3.00</td>
<td>55395</td>
<td>1098</td>
</tr>
<tr>
<td>CM50R2Q200</td>
<td>1023</td>
<td>-4.79</td>
<td>-4.79</td>
<td>-3.81</td>
<td>54729</td>
<td>1074</td>
</tr>
<tr>
<td>CM50R3Q200</td>
<td>1229</td>
<td>-2.93</td>
<td>-3.50</td>
<td>-1.71</td>
<td>55688</td>
<td>1186</td>
</tr>
<tr>
<td>CM50R4Q200</td>
<td>811</td>
<td>-1.36</td>
<td>-1.36</td>
<td>-1.36</td>
<td>61941</td>
<td>800</td>
</tr>
<tr>
<td>CM50R5Q200</td>
<td>970</td>
<td>-4.02</td>
<td>-4.33</td>
<td>-4.12</td>
<td>66656</td>
<td>928</td>
</tr>
<tr>
<td>CM50R1Q400</td>
<td>726</td>
<td>-6.20</td>
<td>-6.20</td>
<td>-4.82</td>
<td>23680</td>
<td>681</td>
</tr>
<tr>
<td>CM50R2Q400</td>
<td>642</td>
<td>-1.71</td>
<td>-1.09</td>
<td>-1.56</td>
<td>25553</td>
<td>631</td>
</tr>
<tr>
<td>CM50R3Q400</td>
<td>741</td>
<td>-0.81</td>
<td>-0.81</td>
<td>-0.81</td>
<td>23231</td>
<td>735</td>
</tr>
<tr>
<td>CM50R4Q400</td>
<td>583</td>
<td>-2.57</td>
<td>-2.74</td>
<td>-2.74</td>
<td>23136</td>
<td>567</td>
</tr>
<tr>
<td>CM50R5Q400</td>
<td>628</td>
<td>-2.71</td>
<td>-2.55</td>
<td>-2.55</td>
<td>25786</td>
<td>612</td>
</tr>
<tr>
<td>CM50R1Q800</td>
<td>544</td>
<td>-8.64</td>
<td>-9.01</td>
<td>-9.01</td>
<td>7414</td>
<td>495</td>
</tr>
<tr>
<td>CM50R2Q800</td>
<td>531</td>
<td>-2.83</td>
<td>-3.01</td>
<td>-3.01</td>
<td>9962</td>
<td>515</td>
</tr>
<tr>
<td>CM50R3Q800</td>
<td>554</td>
<td>-2.53</td>
<td>-3.97</td>
<td>-3.25</td>
<td>15977</td>
<td>532</td>
</tr>
<tr>
<td>CM50R4Q800</td>
<td>472</td>
<td>0.64</td>
<td>0.64</td>
<td>0.64</td>
<td>7468</td>
<td>471</td>
</tr>
<tr>
<td>CM50R5Q800</td>
<td>501</td>
<td>-1.40</td>
<td>-1.40</td>
<td>-1.80</td>
<td>10329</td>
<td>492</td>
</tr>
<tr>
<td>CM100R1Q200</td>
<td>551</td>
<td>-2.90</td>
<td>-4.17</td>
<td>-4.54</td>
<td>7239</td>
<td>520</td>
</tr>
<tr>
<td>CM100R2Q200</td>
<td>616</td>
<td>-1.46</td>
<td>-2.11</td>
<td>-1.79</td>
<td>8488</td>
<td>602</td>
</tr>
<tr>
<td>CM100R3Q200</td>
<td>608</td>
<td>-5.43</td>
<td>-6.58</td>
<td>-9.21</td>
<td>6916</td>
<td>549</td>
</tr>
<tr>
<td>CM100R4Q200</td>
<td>445</td>
<td>3.37</td>
<td>3.15</td>
<td>-0.23</td>
<td>9196</td>
<td>444</td>
</tr>
<tr>
<td>CM100R5Q200</td>
<td>442</td>
<td>2.94</td>
<td>1.13</td>
<td>-0.23</td>
<td>8527</td>
<td>427</td>
</tr>
<tr>
<td>CM100R1Q400</td>
<td>259</td>
<td>4.63</td>
<td>3.86</td>
<td>-1.54</td>
<td>3281</td>
<td>253</td>
</tr>
<tr>
<td>CM100R2Q400</td>
<td>278</td>
<td>2.88</td>
<td>3.60</td>
<td>0.00</td>
<td>3794</td>
<td>278</td>
</tr>
<tr>
<td>CM100R3Q400</td>
<td>238</td>
<td>4.62</td>
<td>2.10</td>
<td>-0.84</td>
<td>3561</td>
<td>236</td>
</tr>
<tr>
<td>CM100R4Q400</td>
<td>223</td>
<td>4.04</td>
<td>4.93</td>
<td>-1.79</td>
<td>3307</td>
<td>219</td>
</tr>
<tr>
<td>CM100R5Q400</td>
<td>227</td>
<td>4.41</td>
<td>2.64</td>
<td>-1.32</td>
<td>3409</td>
<td>224</td>
</tr>
<tr>
<td>CM100R1Q800</td>
<td>182</td>
<td>2.20</td>
<td>4.40</td>
<td>0.00</td>
<td>1483</td>
<td>182</td>
</tr>
<tr>
<td>CM100R2Q800</td>
<td>179</td>
<td>1.12</td>
<td>2.24</td>
<td>0.00</td>
<td>1595</td>
<td>179</td>
</tr>
<tr>
<td>CM100R3Q800</td>
<td>175</td>
<td>3.43</td>
<td>4.00</td>
<td>0.00</td>
<td>1777</td>
<td>175</td>
</tr>
<tr>
<td>CM100R4Q800</td>
<td>183</td>
<td>2.73</td>
<td>3.83</td>
<td>0.00</td>
<td>1778</td>
<td>183</td>
</tr>
<tr>
<td>CM100R5Q800</td>
<td>187</td>
<td>2.67</td>
<td>2.14</td>
<td>-0.54</td>
<td>1801</td>
<td>186</td>
</tr>
<tr>
<td>CM200R1Q200</td>
<td>1147</td>
<td>-8.20</td>
<td>-8.46</td>
<td>-8.89</td>
<td>1381</td>
<td>1037</td>
</tr>
<tr>
<td>CM200R2Q200</td>
<td>1505</td>
<td>-15.75</td>
<td>-15.81</td>
<td>-18.27</td>
<td>1186</td>
<td>1230</td>
</tr>
<tr>
<td>CM200R3Q200</td>
<td>1464</td>
<td>-3.62</td>
<td>-2.53</td>
<td>-5.94</td>
<td>1346</td>
<td>1367</td>
</tr>
<tr>
<td>CM200R4Q200</td>
<td>1017</td>
<td>-2.95</td>
<td>-2.85</td>
<td>-5.90</td>
<td>1120</td>
<td>942</td>
</tr>
<tr>
<td>CM200R5Q200</td>
<td>1145</td>
<td>-13.19</td>
<td>-10.22</td>
<td>-13.71</td>
<td>1285</td>
<td>981</td>
</tr>
<tr>
<td>CM200R1Q400</td>
<td>421</td>
<td>3.33</td>
<td>1.66</td>
<td>-4.28</td>
<td>655</td>
<td>399</td>
</tr>
<tr>
<td>CM200R2Q400</td>
<td>498</td>
<td>4.22</td>
<td>8.84</td>
<td>-1.81</td>
<td>629</td>
<td>486</td>
</tr>
<tr>
<td>CM200R3Q400</td>
<td>587</td>
<td>1.87</td>
<td>2.04</td>
<td>-3.24</td>
<td>658</td>
<td>566</td>
</tr>
<tr>
<td>CM200R4Q400</td>
<td>404</td>
<td>5.20</td>
<td>4.95</td>
<td>-1.73</td>
<td>685</td>
<td>397</td>
</tr>
<tr>
<td>CM200R5Q400</td>
<td>442</td>
<td>1.81</td>
<td>0.45</td>
<td>-3.39</td>
<td>752</td>
<td>425</td>
</tr>
<tr>
<td>CM200R1Q800</td>
<td>256</td>
<td>4.69</td>
<td>6.25</td>
<td>0.39</td>
<td>281</td>
<td>256</td>
</tr>
<tr>
<td>CM200R2Q800</td>
<td>296</td>
<td>4.73</td>
<td>3.72</td>
<td>0.34</td>
<td>328</td>
<td>294</td>
</tr>
<tr>
<td>CM200R3Q800</td>
<td>362</td>
<td>3.32</td>
<td>3.32</td>
<td>0.00</td>
<td>337</td>
<td>362</td>
</tr>
<tr>
<td>CM200R4Q800</td>
<td>276</td>
<td>5.80</td>
<td>7.97</td>
<td>0.00</td>
<td>366</td>
<td>276</td>
</tr>
<tr>
<td>CM200R5Q800</td>
<td>295</td>
<td>4.75</td>
<td>6.10</td>
<td>-0.34</td>
<td>325</td>
<td>293</td>
</tr>
</tbody>
</table>

Table 2-2. Comparison of node-exchange and tree-exchange for heterogeneous demand problems.
Table 2-3. Comparison of 2-exchange with cyclic exchange neighborhood structure.

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>Average Solution Value</th>
<th>Standard Deviation</th>
<th>Average number of iterations</th>
<th>Average time per iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc80-1Q=5</td>
<td>1142.2 1108.0</td>
<td>80.7 81.2</td>
<td>8.7 23.0</td>
<td>0.02 0.02</td>
</tr>
<tr>
<td>tc80-3Q=5</td>
<td>1101.7 1082.3</td>
<td>111.2 43.5</td>
<td>9.5 18.9</td>
<td>0.02 0.02</td>
</tr>
<tr>
<td>tc80-5Q=5</td>
<td>1318.9 1301.1</td>
<td>89.2 43.3</td>
<td>9.4 20.2</td>
<td>0.02 0.02</td>
</tr>
<tr>
<td>tc80-1Q=10</td>
<td>913.4 905.6</td>
<td>126.2 85.8</td>
<td>6.0 12.0</td>
<td>0.05 0.06</td>
</tr>
<tr>
<td>tc80-3Q=10</td>
<td>900.3 890.0</td>
<td>68.7 50.8</td>
<td>4.0 7.0</td>
<td>0.06 0.07</td>
</tr>
<tr>
<td>tc80-5Q=10</td>
<td>1032.3 1023.9</td>
<td>108.5 89.0</td>
<td>2.5 6.3</td>
<td>0.08 0.08</td>
</tr>
<tr>
<td>te80-1Q=5</td>
<td>2572.8 2555.9</td>
<td>207.7 79.7</td>
<td>5.2 11.6</td>
<td>0.02 0.03</td>
</tr>
<tr>
<td>te80-3Q=5</td>
<td>2654.5 2624.8</td>
<td>163.9 88.3</td>
<td>9.3 19.7</td>
<td>0.02 0.03</td>
</tr>
<tr>
<td>te80-5Q=5</td>
<td>2508.6 2486.5</td>
<td>103.8 76.2</td>
<td>7.1 18.0</td>
<td>0.02 0.02</td>
</tr>
<tr>
<td>te80-1Q=10</td>
<td>1728.6 1701.8</td>
<td>260.8 264.6</td>
<td>5.2 14.7</td>
<td>0.06 0.06</td>
</tr>
<tr>
<td>te80-3Q=10</td>
<td>1742.6 1719.0</td>
<td>198.2 132.3</td>
<td>5.2 13.9</td>
<td>0.06 0.06</td>
</tr>
<tr>
<td>te80-5Q=10</td>
<td>1673.8 1651.0</td>
<td>206.1 170.6</td>
<td>5.4 13.4</td>
<td>0.06 0.06</td>
</tr>
<tr>
<td>CM100R1Q=400</td>
<td>285.8 264.7</td>
<td>94.5 60.8</td>
<td>11.0 25.2</td>
<td>0.03 0.04</td>
</tr>
<tr>
<td>CM100R3Q=400</td>
<td>263.3 245.8</td>
<td>90.6 52.5</td>
<td>9.6 21.3</td>
<td>0.04 0.04</td>
</tr>
<tr>
<td>CM100R5Q=400</td>
<td>247.0 233.0</td>
<td>81.5 36.0</td>
<td>10.1 23.6</td>
<td>0.03 0.04</td>
</tr>
<tr>
<td>CM200R1Q=200</td>
<td>1242.5 1059.0</td>
<td>288.3 129.6</td>
<td>16.7 86.5</td>
<td>0.02 0.03</td>
</tr>
<tr>
<td>CM200R3Q=200</td>
<td>1607.9 1411.8</td>
<td>321.0 199.4</td>
<td>19.4 90.3</td>
<td>0.02 0.03</td>
</tr>
<tr>
<td>CM200R5Q=200</td>
<td>1179.9 1015.6</td>
<td>297.6 108.5</td>
<td>22.4 99.1</td>
<td>0.02 0.02</td>
</tr>
</tbody>
</table>

Table 2-4. Performance of the label-correcting heuristic.

<table>
<thead>
<tr>
<th>Problem ID</th>
<th>Total Cycles</th>
<th>Missed Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc80-1K=5</td>
<td>2,261</td>
<td>1</td>
</tr>
<tr>
<td>tc80-2K=5</td>
<td>2,058</td>
<td>0</td>
</tr>
<tr>
<td>te80-1K=5</td>
<td>1,253</td>
<td>0</td>
</tr>
<tr>
<td>te80-2K=5</td>
<td>15,211</td>
<td>0</td>
</tr>
<tr>
<td>CM100-1K=200</td>
<td>4,136</td>
<td>2</td>
</tr>
<tr>
<td>CM100-2K=200</td>
<td>3,331</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>14,560</td>
<td>4</td>
</tr>
</tbody>
</table>

Performance of Composite Neighborhood Structure based Algorithms

We tested our algorithm GRASP3 on a Pentium 4 processor with 512MB RAM and 256KB L2 cache. We ran our algorithm for 1,000 seconds for 50 node problems, 1,800 seconds for problems with 80 and 100 nodes, and 3,600 seconds for problems with 200 nodes. We used
the dynamic programming based method to identify negative cost subset disjoint cycles in the improvement graph. The results from our computational tests are presented in Tables 2-5 and 2-6.

We observe from the results presented in Table 2-5 and 2-6 that the solution values by the composite neighborhood structure are better than the previous algorithms. We obtained the best known solutions for all the standard benchmark instances and improved about 36% of them. We also performed some additional tests to understand the behaviour of our algorithm. Table 2-7 gives the results of these tests on 8 instances of different sizes. We find that the improvement graph is fairly dense; it contains 2n nodes and about 2n^2 arcs. The number of negative cost arcs is a fairly small percent of the total number of arcs and typically varies from 1% to 7%. We find that the algorithm takes longer to construct and update improvement graphs than to identify valid cycles. The time needed to identify valid cycles critically depends upon the number of rooted trees in the optimal solution. As Q decreases, the number of rooted trees increase and it takes longer to identify valid cycles. We also find that as k increases, the number of states generated by the DP decreases rapidly.
<table>
<thead>
<tr>
<th>Problem type</th>
<th>Number of nodes</th>
<th>Number of arcs</th>
<th>Capacity</th>
<th>Best available solution</th>
<th>Number of runs</th>
<th>Composite neighborhood solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Te80-1</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>1099</td>
<td>6901</td>
<td>1099</td>
</tr>
<tr>
<td>Te80-2</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>1100</td>
<td>8045</td>
<td>1100</td>
</tr>
<tr>
<td>Te80-3</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>1073</td>
<td>9175</td>
<td>1073</td>
</tr>
<tr>
<td>Te80-4</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>1080</td>
<td>7552</td>
<td>1080</td>
</tr>
<tr>
<td>Te80-5</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>1287</td>
<td>8842</td>
<td>1287</td>
</tr>
<tr>
<td>Te80-1</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>888</td>
<td>4649</td>
<td>888</td>
</tr>
<tr>
<td>Te80-2</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>877</td>
<td>4863</td>
<td>877</td>
</tr>
<tr>
<td>Te80-3</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>880</td>
<td>5164</td>
<td>878</td>
</tr>
<tr>
<td>Te80-4</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>868</td>
<td>3751</td>
<td>868</td>
</tr>
<tr>
<td>Te80-5</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>1002</td>
<td>6115</td>
<td>1002</td>
</tr>
<tr>
<td>Te80-1</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>834</td>
<td>2985</td>
<td>834</td>
</tr>
<tr>
<td>Te80-2</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>820</td>
<td>2390</td>
<td>820</td>
</tr>
<tr>
<td>Te80-3</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>828</td>
<td>4120</td>
<td>828</td>
</tr>
<tr>
<td>Te80-4</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>820</td>
<td>2667</td>
<td>820</td>
</tr>
<tr>
<td>Te80-5</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>916</td>
<td>2124</td>
<td>916</td>
</tr>
<tr>
<td>Te80-1</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>2544</td>
<td>6649</td>
<td>2544</td>
</tr>
<tr>
<td>Te80-2</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>2551</td>
<td>4935</td>
<td>2551</td>
</tr>
<tr>
<td>Te80-3</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>2612</td>
<td>3879</td>
<td>2612</td>
</tr>
<tr>
<td>Te80-4</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>2558</td>
<td>4967</td>
<td>2558</td>
</tr>
<tr>
<td>Te80-5</td>
<td>81</td>
<td>3240</td>
<td>5</td>
<td>2469</td>
<td>5668</td>
<td>2469</td>
</tr>
<tr>
<td>Te80-1</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>1657</td>
<td>3413</td>
<td>1657</td>
</tr>
<tr>
<td>Te80-2</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>1639</td>
<td>5153</td>
<td>1639</td>
</tr>
<tr>
<td>Te80-3</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>1687</td>
<td>3451</td>
<td>1687</td>
</tr>
<tr>
<td>Te80-4</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>1629</td>
<td>2606</td>
<td>1629</td>
</tr>
<tr>
<td>Te80-5</td>
<td>81</td>
<td>3240</td>
<td>10</td>
<td>1603</td>
<td>3504</td>
<td>1603</td>
</tr>
<tr>
<td>Te80-1</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>1275</td>
<td>1512</td>
<td>1275</td>
</tr>
<tr>
<td>Te80-2</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>1224</td>
<td>1348</td>
<td>1224</td>
</tr>
<tr>
<td>Te80-3</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>1267</td>
<td>2182</td>
<td>1267</td>
</tr>
<tr>
<td>Te80-4</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>1265</td>
<td>1460</td>
<td>1265</td>
</tr>
<tr>
<td>Te80-5</td>
<td>81</td>
<td>3240</td>
<td>20</td>
<td>1240</td>
<td>2167</td>
<td>1240</td>
</tr>
</tbody>
</table>

Table 2.5. Computational results of Composite neighborhood for homogeneous demand problems.
<table>
<thead>
<tr>
<th>Problem type</th>
<th>Number of nodes</th>
<th>Number of arcs</th>
<th>Capacity</th>
<th>Best available</th>
<th>Number of runs</th>
<th>Composite neighborhood</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM50-1</td>
<td>50</td>
<td>1225</td>
<td>200</td>
<td>1098</td>
<td>16131</td>
<td>1098</td>
</tr>
<tr>
<td>CM50-2</td>
<td>50</td>
<td>1225</td>
<td>200</td>
<td>974</td>
<td>19574</td>
<td>974</td>
</tr>
<tr>
<td>CM50-3</td>
<td>50</td>
<td>1225</td>
<td>200</td>
<td>1186</td>
<td>15785</td>
<td>1186</td>
</tr>
<tr>
<td>CM50-4</td>
<td>50</td>
<td>1225</td>
<td>200</td>
<td>800</td>
<td>25149</td>
<td>800</td>
</tr>
<tr>
<td>CM50-5</td>
<td>50</td>
<td>1225</td>
<td>200</td>
<td>928</td>
<td>22114</td>
<td>928</td>
</tr>
<tr>
<td>CM50-1</td>
<td>50</td>
<td>1225</td>
<td>400</td>
<td>681</td>
<td>9914</td>
<td>681</td>
</tr>
<tr>
<td>CM50-2</td>
<td>50</td>
<td>1225</td>
<td>400</td>
<td>631</td>
<td>11198</td>
<td>631</td>
</tr>
<tr>
<td>CM50-3</td>
<td>50</td>
<td>1225</td>
<td>400</td>
<td>735</td>
<td>12782</td>
<td>732</td>
</tr>
<tr>
<td>CM50-4</td>
<td>50</td>
<td>1225</td>
<td>400</td>
<td>567</td>
<td>15516</td>
<td>564</td>
</tr>
<tr>
<td>CM50-5</td>
<td>50</td>
<td>1225</td>
<td>400</td>
<td>612</td>
<td>10306</td>
<td>611</td>
</tr>
<tr>
<td>CM50-1</td>
<td>50</td>
<td>1225</td>
<td>800</td>
<td>495</td>
<td>4963</td>
<td>495</td>
</tr>
<tr>
<td>CM50-2</td>
<td>50</td>
<td>1225</td>
<td>800</td>
<td>515</td>
<td>5498</td>
<td>513</td>
</tr>
<tr>
<td>CM50-3</td>
<td>50</td>
<td>1225</td>
<td>800</td>
<td>532</td>
<td>9347</td>
<td>532</td>
</tr>
<tr>
<td>CM50-4</td>
<td>50</td>
<td>1225</td>
<td>800</td>
<td>471</td>
<td>5839</td>
<td>471</td>
</tr>
<tr>
<td>CM50-5</td>
<td>50</td>
<td>1225</td>
<td>800</td>
<td>492</td>
<td>4779</td>
<td>492</td>
</tr>
<tr>
<td>CM100-1</td>
<td>100</td>
<td>4950</td>
<td>200</td>
<td>520</td>
<td>2874</td>
<td>516</td>
</tr>
<tr>
<td>CM100-2</td>
<td>100</td>
<td>4950</td>
<td>200</td>
<td>602</td>
<td>2818</td>
<td>596</td>
</tr>
<tr>
<td>CM100-3</td>
<td>100</td>
<td>4950</td>
<td>200</td>
<td>549</td>
<td>2480</td>
<td>541</td>
</tr>
<tr>
<td>CM100-4</td>
<td>100</td>
<td>4950</td>
<td>200</td>
<td>444</td>
<td>3166</td>
<td>437</td>
</tr>
<tr>
<td>CM100-5</td>
<td>100</td>
<td>4950</td>
<td>200</td>
<td>427</td>
<td>2766</td>
<td>425</td>
</tr>
<tr>
<td>CM100-1</td>
<td>100</td>
<td>4950</td>
<td>400</td>
<td>253</td>
<td>1742</td>
<td>252</td>
</tr>
<tr>
<td>CM100-2</td>
<td>100</td>
<td>4950</td>
<td>400</td>
<td>278</td>
<td>1974</td>
<td>278</td>
</tr>
<tr>
<td>CM100-3</td>
<td>100</td>
<td>4950</td>
<td>400</td>
<td>236</td>
<td>1970</td>
<td>236</td>
</tr>
<tr>
<td>CM100-4</td>
<td>100</td>
<td>4950</td>
<td>400</td>
<td>219</td>
<td>1730</td>
<td>219</td>
</tr>
<tr>
<td>CM100-5</td>
<td>100</td>
<td>4950</td>
<td>400</td>
<td>224</td>
<td>1901</td>
<td>223</td>
</tr>
<tr>
<td>CM100-1</td>
<td>100</td>
<td>4950</td>
<td>800</td>
<td>182</td>
<td>826</td>
<td>182</td>
</tr>
<tr>
<td>CM100-2</td>
<td>100</td>
<td>4950</td>
<td>800</td>
<td>179</td>
<td>1028</td>
<td>179</td>
</tr>
<tr>
<td>CM100-3</td>
<td>100</td>
<td>4950</td>
<td>800</td>
<td>175</td>
<td>961</td>
<td>175</td>
</tr>
<tr>
<td>CM100-4</td>
<td>100</td>
<td>4950</td>
<td>800</td>
<td>183</td>
<td>936</td>
<td>183</td>
</tr>
<tr>
<td>CM100-5</td>
<td>100</td>
<td>4950</td>
<td>800</td>
<td>186</td>
<td>1074</td>
<td>186</td>
</tr>
<tr>
<td>CM200-1</td>
<td>200</td>
<td>19900</td>
<td>200</td>
<td>1037</td>
<td>409</td>
<td>1017</td>
</tr>
<tr>
<td>CM200-2</td>
<td>200</td>
<td>19900</td>
<td>200</td>
<td>1230</td>
<td>360</td>
<td>1221</td>
</tr>
<tr>
<td>CM200-3</td>
<td>200</td>
<td>19900</td>
<td>200</td>
<td>1367</td>
<td>408</td>
<td>1365</td>
</tr>
<tr>
<td>CM200-4</td>
<td>200</td>
<td>19900</td>
<td>200</td>
<td>942</td>
<td>334</td>
<td>927</td>
</tr>
<tr>
<td>CM200-5</td>
<td>200</td>
<td>19900</td>
<td>200</td>
<td>981</td>
<td>404</td>
<td>965</td>
</tr>
<tr>
<td>CM200-1</td>
<td>200</td>
<td>19900</td>
<td>400</td>
<td>399</td>
<td>587</td>
<td>397</td>
</tr>
<tr>
<td>CM200-2</td>
<td>200</td>
<td>19900</td>
<td>400</td>
<td>486</td>
<td>589</td>
<td>478</td>
</tr>
<tr>
<td>CM200-3</td>
<td>200</td>
<td>19900</td>
<td>400</td>
<td>566</td>
<td>525</td>
<td>560</td>
</tr>
<tr>
<td>CM200-4</td>
<td>200</td>
<td>19900</td>
<td>400</td>
<td>397</td>
<td>670</td>
<td>392</td>
</tr>
<tr>
<td>CM200-5</td>
<td>200</td>
<td>19900</td>
<td>400</td>
<td>425</td>
<td>689</td>
<td>420</td>
</tr>
<tr>
<td>CM200-1</td>
<td>200</td>
<td>19900</td>
<td>800</td>
<td>256</td>
<td>340</td>
<td>254</td>
</tr>
<tr>
<td>CM200-2</td>
<td>200</td>
<td>19900</td>
<td>800</td>
<td>294</td>
<td>382</td>
<td>294</td>
</tr>
<tr>
<td>CM200-3</td>
<td>200</td>
<td>19900</td>
<td>800</td>
<td>362</td>
<td>380</td>
<td>361</td>
</tr>
<tr>
<td>CM200-4</td>
<td>200</td>
<td>19900</td>
<td>800</td>
<td>276</td>
<td>432</td>
<td>275</td>
</tr>
<tr>
<td>CM200-5</td>
<td>200</td>
<td>19900</td>
<td>800</td>
<td>293</td>
<td>403</td>
<td>292</td>
</tr>
</tbody>
</table>

Table 2-6. Computational results of the Composite neighborhood for heterogeneous demand problems.
| Problem type | Q | Average # of arcs in G³ | Average # of negative cost arcs in G³ | Average time to construct G³ (millisecond) | Average time to find a cycle (millisecond) | \( |X_k| \) |
|--------------|---|-------------------------|--------------------------------------|---------------------------------------------|--------------------------------------------|-----|
| Tc80-1       | 5 | 15277                   | 298                                  | 9                                           | 2                                          | 900 | 26 | 16 |
| Tc80-1       | 20| 15461                   | 114                                  | 117                                         | 2                                          | 613 | 175| 4  |
| Te80-1       | 5 | 14797                   | 1117                                 | 9                                           | 13                                         | 913 | 20 | 32 |
| Te80-1       | 20| 13162                   | 283                                  | 129                                         | 2                                          | 920 | 44 | 0  |
| CM50-1       | 200| 4235                   | 255                                  | 3                                           | 2                                          | 844 | 5  | 1  |
| CM50-1       | 800| 4815                   | 109                                  | 41                                          | 1                                          | 861 | 5  | 1  |
| CM200-1      | 200| 73718                   | 2870                                 | 25                                          | 74                                         | 352 | 157| 56 |
| CM200-1      | 800| 97247                   | 1277                                 | 375                                         | 11                                         | 672 | 67 | 15 |

Table 2-7. Computational behavior of the composite neighborhood based algorithm.

2.5 Additional Partitioning Problems

In addition to studying cyclic exchanges for the capacitated minimum spanning tree problem, we also applied cyclic exchange based neighborhood search algorithms to two other interesting partitioning problems. These problems are generalized assignment problem (GAP) and the integrated clustering and machine setup (ICMS) model in printed circuit board manufacturing (Magazine et al. [2001]). In the case of ICMS, the cyclic exchange based neighborhood search algorithm obtained the optimal solution for all the problem instances for which the optimal solution was known. We do not describe this problem in detail here, and refer the reader to the paper of Magazine et al. [2001] for the description of the model and the results of cyclic exchange neighborhood search. In this section, we provide computational results on the GAP.

2.5.1 Computational Results for GAP

The generalized assignment problem (GAP) is a well-known NP-hard problem. In this problem, we are given \( n \) jobs \( J = \{1, \ldots, n\} \) and \( K \) agents \( I = \{1, \ldots, K\} \). The assignment of job \( j \) to agent \( i \) costs \( c_{ij} \) and takes an amount \( a_{ij} \) of resource on agent \( i \). The total resource available at agent \( i \) is given by \( b_i \). The data \( c_{ij}, a_{ij}, \) and \( b_i \) are all non-negative. The generalized assignment problem is to find a minimum cost assignment of jobs to agents such that the resource consumption by the jobs assigned to any agent does not exceed the resource available at the agent. Any feasible solution \( \sigma \) to the GAP is a partition of the set of jobs into \( K \) subsets \( S_1, S_2, \ldots \).
$S_k$ such that $\sum_{j \in S_i} a_{ij} \leq b_i$ for $i = 1, \ldots, K$. The cost associated with the jobs assigned to agent $i$ in the partition is given by $c(S_i) = \sum_{j \in S_i} c_{ij}$. We use $S[j]$ to denote the subset of jobs containing job $j$ in the solution $\sigma$.

There is substantial literature devoted to the development of exact and heuristic algorithms for GAP given its practical interest. We refer the reader to the papers of Catrysse and Wassenhove [1992] and Yagiura et al. [1999a], and their references for algorithms developed for GAP. Currently, the best available heuristic algorithm for solving this problem is by Yagiura et al. [1999b], which used an ejection chain based tabu search algorithm.

It is NP-hard to even find a feasible solution to the generalized assignment problem. Therefore, the neighborhood search algorithm has to work with an infeasible solution. We perform this by removing the constraints on each subset $S_i$ and penalize their violation in the objective function. Given a penalty parameter $\alpha$, we redefine the objective function associated with each set $Si$ as: $c'(S_i) = c(S_i) + \alpha \max \{0, \sum_{j \in S_i} a_{ij} - b_i\}$. Note that for sufficiently large value of $\alpha$, an optimal solution for the new unconstrained problem with objective $c'$ is also feasible and optimal for the original GAP.

We used the standard cyclic exchange neighborhood structure for the GAP, where we consider cyclic and path exchanges involving transfer of single jobs between machines. We create an initial solution (for the unconstrained problem) by randomized rounding of the solution obtained from a linear programming relaxation of the 0-1 integer programming formulation of the GAP (Shmoys and Tardos [1993]). Our neighborhood search algorithm works as follows. We start with a small value of the penalty $\alpha$. Note that if we keep the penalty too low then a local search algorithm will always reach a solution where every job is assigned to its least costly machine. Hence, we choose $\alpha$ as the minimum value such that shifting one job from its currently assigned machine to another machine is not profitable. We perform improving iterations until we reach a local optimal solution for the current value of $\alpha$. At this point, if the current solution is feasible, we stop, otherwise we double the value of $\alpha$. We continue this process until we either reach a feasible solution or we have increased the value of the penalty parameter $\alpha$ beyond a specified threshold. Note that this procedure is not guaranteed to even generate a feasible solution. However, in our computational tests, we achieved a feasible solution in every run of the neighborhood search algorithm. We applied this algorithm to the type D and type E benchmark instances of GAP in the OR-Library, which are known to be the hardest set of instances (Yagiura
et al. [1999a]). For each of the instances, we performed 50 runs of the neighborhood search algorithm starting at solutions generated by our randomized procedure. We present the best solutions obtained by our simple local search algorithm in Table 2-9. The neighborhood search algorithm is within 0.3% of the best available solutions on average. We have not reported the results on the easy problems as our algorithm achieves the best-available results on all of them. Given that our implementation does not use any sophisticated meta-heuristics as other algorithms in the literature such as Yagiura et al. [1999b], these results indicate that the cyclic exchange neighborhood search technique is very effective.

<table>
<thead>
<tr>
<th>Problems of Type D</th>
<th></th>
<th></th>
<th>Problems of Type E</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem ID</td>
<td>Best</td>
<td>Cyclic Exchange</td>
<td>Problem ID</td>
<td>Best</td>
<td>Cyclic Exchange</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n=100K=5</td>
<td>6353</td>
<td>0.268</td>
<td>n=100K=5</td>
<td>12681</td>
<td>0.047</td>
</tr>
<tr>
<td>n=100K=10</td>
<td>6348</td>
<td>0.567</td>
<td>n=100K=10</td>
<td>11577</td>
<td>0.121</td>
</tr>
<tr>
<td>n=100K=20</td>
<td>6196</td>
<td>0.952</td>
<td>n=100K=20</td>
<td>8436</td>
<td>0.486</td>
</tr>
<tr>
<td>n=200K=5</td>
<td>12743</td>
<td>0.267</td>
<td>n=200K=5</td>
<td>24930</td>
<td>0.072</td>
</tr>
<tr>
<td>n=200K=10</td>
<td>12433</td>
<td>0.153</td>
<td>n=200K=10</td>
<td>23307</td>
<td>0.013</td>
</tr>
<tr>
<td>n=200K=20</td>
<td>12244</td>
<td>0.653</td>
<td>n=200K=20</td>
<td>22379</td>
<td>0.067</td>
</tr>
</tbody>
</table>

Table 2-8. Results of cyclic exchange neighborhood search for GAP.

2.6 Summary and Conclusions

In this chapter, we focused on the cyclic exchange neighborhood structure for partitioning problems. The neighborhood search problem for the cyclic exchange neighborhood is NP-hard in general and its size can be $\Omega((n/K)^K(K-1)!))$. We introduce new heuristic and exact algorithms for this problem using the concept of improvement graph proposed by Thompson and Orlin [1989]. In particular, we developed a dynamic programming based algorithm that can search the neighborhood in time $O(nmK2^K)$, which is only exponential in the number of partitions. We describe methods of efficiently implementing this procedure in practice using state pruning and hash tables. We also describe a preprocessing technique for improvement graphs that can improve the performance of this algorithm. We developed new neighborhood search algorithms for the capacitated minimum spanning tree (CMST) problem, where we use our proposed algorithms to perform the neighborhood search.

The computational results for our algorithms on the set of benchmark instances for the CMST problem show that cyclic exchange neighborhood structure is superior compared to the two-exchange neighborhood structure, and it can be searched in about the same time as the two-
exchange neighborhood structure. We also provide additional computational results for cyclic exchange based neighborhood structure based neighborhood search algorithm for GAP, which further indicate that it is a powerful neighborhood structure for the partitioning problems whenever it can be searched efficiently.
Chapter 3

Combined Through and Fleet Assignment Model

3.1 Introduction

The airline industry has been a pioneer in using OR techniques to solve complex business problems related to the schedule planning of the airline. Given a flight schedule, an airline's schedule planning group needs to decide the itinerary of each aircraft and each crewmember that maximizes the total revenue minus the total operating costs while satisfying all relevant operational constraints. The quality of the schedule is also measured by other attributes such as schedule reliability during operations. The entire planning problem is too large to be solved to optimality as a single optimization problem using present day technology. Hence, it is typically divided into four stages (see, for example, Barnhart and Talluri [1997] and Gopalan and Talluri [1998]): (i) fleet assignment; (ii) through assignment; (iii) maintenance routing; and (iv) crew scheduling. These problems are solved sequentially with the optimal solution of one problem becoming the input for the following problem. There has been significant effort spent in modeling and solving these individual problems using advanced optimization models. The economies of scale at a large airline are such that a relatively minor improvement in contribution results in considerable improvement in the bottom line. As a result, airlines have benefited immensely from the advances in modeling these problems.

Currently, an important research issue in airline schedule planning is to solve an integrated optimization problem that will consider the entire planning problem mentioned above and include other downstream issues that affect the overall schedule quality. Basically, the planning problem is a multi-criteria optimization problem, i.e., there are many objectives that have different metrics and different priorities. A sequential approach to solve such problems has a major drawback in that the solution at each stage does not account for the considerations of subsequent stages. This results in overall suboptimal solutions. For example, if a fleet assignment
is performed without considerations of optimizing crew scheduling, the result could be a very bad input for the crew scheduling process. On the other hand, if the fleet assignment incorporates crew issues, then it is likely to provide a better starting point to the crew scheduling optimization, resulting in overall economic benefits for the airline.

The airlines are devoting considerable effort to develop models to solve such integrated optimization problems for schedule planning. One possible strategy is to develop explicit joint optimization models with combined objective functions, combined set of constraints and combined data. Typically, these joint models are too large to be solved to optimality or near-optimality, suggesting that heuristics might be needed. Moreover, some downstream criteria cannot be represented easily in a form consistent with explicitly modeling the problem. For example, although an important criteria in schedule planning that is related to the schedule structure, airline reliability is very hard to model as a typical optimization problem. Another strategy for providing an integrated optimization approach is to exploit the multi-criteria nature of the planning problem. Typically, many solutions are close to the optimal in terms of contribution. However, these solutions can have very distinct characteristics on other criteria such as crew required, potential for through flights, schedule reliability, ground manpower requirements, etc. This implies that intelligent search techniques, when coupled with advanced optimization modeling, hold a lot of promise in solving multi-criteria schedule planning problems.

In this chapter, we explore neighborhood search as a strategy that builds upon the foundation that already existed for modeling the individual stages and can be easily scaled up to address the integrated problems. We propose an integrated approach that first solves the separate models to optimality in a stage-wise fashion followed by solving the integrated model heuristically using neighborhood search techniques. This approach guarantees that the solution obtained by our approach is no worse than the solution obtained by the current sequential approach and in practice is better. We also use neighborhood search technique as a tool to solve the multi-criteria problem, where we generate multiple good solutions that may be evaluated for later planning stages instead of just one solution.

In this thesis, we focus on integrating two of the airline scheduling models, the Fleet Assignment Model (FAM) and the Through Assignment Model (TAM), into a single model that we call the Combined Through Fleet Assignment Model (ctFAM). We next briefly describe these three models.

**Fleet Assignment Model (FAM):** In FAM, planes of different fleet type are assigned to flight legs to minimize the assignment cost, which is the sum of the operating cost of a fleet type and
the revenue lost by assigning it to a flight leg. A fleet assignment has to satisfy the following three types of constraints: (i) covering constraints: each flight leg must be assigned exactly one plane; (ii) flow balance constraints: for each fleet type, the number of planes landing at a city must be equal to the number of planes taking off from the city; and (iii) fleet size constraint: for each fleet type, the number of planes used must not exceed the number of planes available. Abara [1989] and Hane et al. [1995] give an MIP formulation for FAM. Subsequently, Clarke et al. [1996] and Subramaniam et al. [1994] provide extensions to incorporate additional operational constraints related to maintenance and crew scheduling. Barnhart et. al [1998] use a column generation approach to solve the problems of fleet assignment, through assignment, and maintenance routing jointly as one optimization problem. The approach works successfully for the (small) international schedules, however, it is not tractable for the large hub-and-spoke networks in the domestic flight schedules of US airlines. Kniker et al. [2000] provide a new itinerary based model for the fleet assignment that uses the origin-destination demands explicitly in the model.

**Through Assignment Model (TAM):** A through connection is a connection between an inbound flight leg and an outbound flight leg at a station which ensures that the same plane flies both legs. Both flights legs in a through connection get the same flight number in the airline’s flight schedule. Since a through connection allows passengers to remain onboard instead of changing gates at busy airports, passengers are willing to pay a premium for such connections; this premium is termed as the through benefit. TAM takes as an input a list of candidate pairs of flight legs that can make through connections with corresponding through benefits, and identifies a set of most profitable through connections. Observe that we can make through connections only between flights flown by the same fleet type; hence the fleet assignment limits the possible through connections. In the current implementations, TAM takes as an input the fleet assignment, identifies inbound and outbound flights at each city flown by the same fleet type, and determines through connections (that must be a subset of the candidate pairs) to maximize the through benefit. This problem can be solved as a bipartite matching problem. However, in practice the solution must satisfy some additional constraints, which yields a constrained bipartite assignment problem that can be solved using MIP techniques. We refer the reader to the papers by Bard and Hopperstad [1987], Barnhart et al. [1998], Gopalan and Talluri [1998], and Jarrah and Reeb [1997] for additional details on the TAM.

**The Combined Through Fleet Assignment Model (cTFAM):** The through assignment depends on the fleet assignment in that a through connection requires that both its flight legs have the
same fleet type. In the current systems, FAM does not take into account the through benefits, and may yield fleet assignments with limited through assignment possibilities. TAM cannot change the fleeting in order to get a better through assignment. In our model, ctFAM solves the integrated model and simultaneously determines fleet assignments and through connections. The integrated model offers opportunities to obtain better solutions compared to the current sequential approach. We first developed an integer programming formulation of ctFAM, which was too large to be solved to optimality or near-optimality for a major US airline. We then pursued the approach outlined in Figure 3-1. In our approach, we first solve FAM to obtain an optimal (or nearly optimal) fleet assignment. For this fleeting, we then solve TAM to determine optimal (or nearly optimal) through connections. We then solve ctFAM heuristically using the neighborhood search algorithm with the optimal FAM and TAM solutions as the starting solution for the neighborhood search.

**Figure 3-1. An approach to solve ctFAM.**

Neighborhood search algorithms are widely regarded as an important tool to solve difficult combinatorial optimization problems effectively. The primary reasons for the widespread application of neighborhood search techniques in practice are their intuitive appeal, flexibility and ease of implementation, and their excellent empirical results (see, for example, Aarts and Lenstra [1997], and Glover and Laguna [1997]). We decided to pursue neighborhood search algorithms for ctFAM for the following reasons:

(i) Neighborhood search algorithms have been very successful in solving a variety of large-scale combinatorial optimization problems.

(ii) Neighborhood search algorithms permit us to start with the excellent solution obtained by solving FAM first followed by solving TAM. This guarantees that the neighborhood search algorithm obtains a solution that is at least as good as that obtained by FAM followed by TAM, and possibly better.

(iii) A neighborhood search algorithm examines several solutions during execution. We can examine these solutions and generate a set of good solutions for later stages instead of just a single solution.
(iv) Neighborhood search algorithms are often flexible enough to incorporate other constraints that are difficult to model through linear constraints.

We now present a brief overview of our neighborhood search algorithm for ctFAM. An issue of critical importance in a neighborhood search algorithm is the manner in which we define the neighborhood of a solution. As in Talluri [1996], we define neighbors of a given solution by performing "A-B swaps" for two specified fleet types A and B. An A-B swap consists of changing fleet types of some legs from A to B and of some legs from B to A so that all constraints remain satisfied. A profitable A-B swap decreases the total cost of the solution, which in our case includes the costs of throughs. Identifying a profitable A-B swap is not a trivial problem because the number of possible A-B swaps is exponentially large. Hence the neighborhood search algorithm using A-B swaps falls within the category of very large-scale neighborhood (VLSN) search algorithms, as introduced in Chapter 1.

Our approach consists of determining the starting solution by first solving FAM followed by TAM. This solution is successively improved by our neighborhood search algorithm. In each iteration, the neighborhood search algorithm selects any two fleet types, which we label as A and B, and performs a profitable "A-B swap". An A-B swap consists of changing some legs flown by fleet type A to fleet type B, changing some legs flown by fleet type B to fleet type A, and changing some through connections appropriately. The number of A-B swaps can be very large and difficult to enumerate explicitly. We describe a method using A-B Improvement Graphs which allows us to obtain profitable A-B swaps quickly in practice. The A-B improvement graph is constructed in a manner that each negative cost directed cycle in the graph satisfying some constraints defines a profitable A-B swap.

The neighborhood search algorithms constructs the A-B improvement graph and solves an integer programming problem to identify a negative cost constrained directed cycle. This cycle yields a new fleet and through assignment with a lower cost. We repeat this process for every pair of fleet types A and B, and terminate when for every such pair of fleet types, we do not find improved neighbors. We developed and implemented both a local improvement algorithm (where we always perform cost-decreasing iterations) and a tabu search algorithm (where we sometimes allow cost-increasing iterations too). Our local improvement algorithm obtains a local optimal solution for ctFAM in 5-6 seconds, whereas we ran the tabu search algorithm for 20-25 minutes on our data sets, which are of realistic size data. The solutions obtained by our algorithms resulted in savings of $5 million to $25 million on an annual basis on the data provided by a major US
airline. These results suggest that neighborhood search is a useful supplement to the techniques already developed in airline operations research.

One of our major contributions in this paper is to extend Talluri's [1996] concept of A-B improvement graph so that it incorporates through constraints. A disadvantage of Talluri's neighborhood is that it relies on the existence of "banks": any plane arriving at the bank would have sufficient time to be assigned to any of the departures at the bank. This assumption is overly restrictive in practice. Our neighborhood structure does not make this assumption. Indeed, it does not even depend upon the existence of arrival and departure banks.

In addition to solving the ctFAM, we also considered a multi-criteria version of ctFAM. An optimal or near optimal solution of the ctFAM may not necessarily be good for the subsequent planning decisions models. In multi-criteria ctFAM, we consider a measure of the "goodness" of a fleet and through assignment for the subsequent manpower planning problem as an objective in addition to the contribution of the fleet and through assignment. We describe modifications to our neighborhood search algorithm to identify a set of solutions that provide a tradeoff between better contribution and better ground manpower resource consumption. Although our computational results indicated that the ground manpower resource usage was not an important criterion to be considered against contribution, the approach developed by us could be used to generate solutions that tradeoff contribution with other criterion and provide planners with multiple choices for identifying good schedules.

The rest of this chapter is organized as follows. In Section 2, we present an integer programming formulation for ctFAM. Section 3 develops our neighborhood search algorithms for ctFAM. We describe some implementation details in Section 4. We give our computational results in Section 5. In Section 6, we describe ways of modifying our neighborhood search algorithm to perform multi-criteria optimization. Section 7 gives conclusions of our research and avenues of future research.

3.2 An Integer Programming Formulation of ctFAM

In this section, we present an integer linear (IP) formulation of ctFAM. We formulate this problem as a flow problem on a network, which we call the connection network. We first present input data for ctFAM, followed by the description of the connection network, followed by the IP formulation.

Input Data

We have the following input data for ctFAM:
L : The set of all flight legs needed to be assigned planes. We use the index \( i \) to represent a particular leg.

F : The set of all fleet types. We use the index \( f \) to represent a particular fleet type.

T : The set of all candidate through connections. Each through connection is specified by a pair \((i, j)\) of flights.

\( size(f) \) : The number of planes of fleet type \( f \) available for assignment.

\( dep-time(i) \) : The departure time for flight leg \( i \).

\( arr-time(i) \) : The arrival time for flight leg \( i \). We denote by \( arr-time(i) \) as the time when flight \( i \) actually arrives plus the turn time (the time need to prepare the plane to be assigned to the next flight*). Thus, the plane released from the flight \( i \) can be assigned to any flight \( j \) with \( dep-time(j) \geq arr-time(i) \) at the same city.

\( dep-city(i) \) : The departure city for flight leg \( i \).

\( arr-city(i) \) : The arrival city for flight leg \( i \).

\( c^f_i \) : The cost incurred in assigning fleet type \( f \) to flight leg \( i \).

\( d^f_{ij} \) : The cost incurred in connecting flight leg \( i \) with the flight leg \( j \) provided \( arr-city(i) = dep-city(j) \) and both the legs are flown by the fleet type \( f \). Observe that \( d^f_{ij} < 0 \) for \((i, j) \in T\), and 0 otherwise.

\( count-time \) : A time instant on the 24-hour time scale when no plane leaves or arrives, that is, \( count-time \neq arr-time(i) \) or \( dep-time(i) \) for any \( i \in L \). We will assume here that count-time is midnight.

**Connection Network**

We now explain how to construct the connection network, which will be the basis of our integer programming formulation as well as our neighborhood search algorithm for ctFAM. We denote the connection network as \( G = (N, E) \) where \( N \) denotes the node set and \( E \) denotes the arc set. The node set \( N = \{i : i \in L\} \) is obtained by defining a node for each flight leg \( i \in L \), and the

* In practice, the turn time also depends upon the fleet type but for the simplicity of notation, we assume it to be independent of the fleet type.
arc set $E = \{(i, j): \text{arr-city}(i) = \text{dep-city}(j)\}$ consists of all possible connections between inbound and outbound flight legs. Obviously, a connection between flight legs $i$ and $j$ is possible only if the arrival city of leg $i$ is the same as the departure city of leg $j$. We give an example of connection network in Figure 3-2.

Figure 3-2. Part of the connection network at a city with arrivals 1, 2, and 3, and departures 4, 5, and 6.

A connection arc $(i, j)$ is said to be a through connection arc if $(i, j) \in T$ and a regular connection arc otherwise. We will use the following additional notation related to the connection network:

$I(i) = \{(j, i) \in E: j \in N\},$

$O(i) = \{(i, j) \in E: j \in N\},$

$S = \{(i, j) \in E: \text{arr-time}(i) < \text{count-time} < \text{arr-time}(j)\} \cup \{(i, j) \in E: \text{dep-time}(i) < \text{count-time} < \text{arr-time}(i)\}$, where the inequalities are based on the circular 24-hour time.

The set $I(i)$ denotes the set of incoming arcs at node $i$ in the connection network, the set $O(i)$ denotes the set of outgoing arcs, and $S$ denotes the set of arcs in the connection network that cross the count-time, which we assume to be midnight. We call the arcs in $S$ as overnighting arcs; it contains the set of connection arcs that cross the count-time and also those connection arcs whose arrival flights are in the air at count-time.

**Decision Variables**

We define two sets of decision variables in our integer programming formulation. The first set of decision variables $(y^f_i)$ specify the fleet assignment and the second set of decision variables $(x^f_{ij})$ specify the (regular or through) connection assignment.

$y^f_i$: This variable takes value 1 if the flight leg $i$ is assigned fleet type $f$, and 0 otherwise.
\( x_{ij} \): This variable takes value 1 if both the flight legs \( i \) and \( j \) are flown by the fleet type \( f \) and we make a (regular or through) connection between the flight legs \( i \) and \( j \), and 0 otherwise.

**Integer Programming Formulation**

We give below the integer programming formulation of ctFAM.

\[
\text{Minimize} \quad \sum_{i \in N} \sum_{f \in F} c_i^f y_i^f + \sum_{(i,j) \in E} \sum_{f \in F} d_{ij}^f x_{ij}^f \tag{3.1a}
\]

subject to

\[
\sum_{f \in F} y_i^f = 1, \quad \text{for all } i \in N \tag{3.1b}
\]

\[
\sum_{(i,j) \notin O(i)} x_{ij}^f = y_i^f, \quad \text{for all } i \in N \text{ and all } f \in F \tag{3.1c}
\]

\[
\sum_{(i,j) \notin I(j)} x_{ij}^f = y_j^f, \quad \text{for all } j \in N \text{ and all } f \in F \tag{3.1d}
\]

\[
\sum_{(i,j) \notin S} x_{ij}^f \leq \text{size}(f), \quad \text{for all } f \in F \tag{3.1e}
\]

\[
x_{ij}^f \in \{0, 1\}, \quad \text{for all } (i, j) \in E \text{ and for all } f \in F \tag{3.1f}
\]

\[
y_i^f \in \{0, 1\}, \quad \text{for all } i \in N \text{ and for all } f \in F \tag{3.1g}
\]

We represent a feasible solution of ctFAM as \((x, y)\). The first and second terms in the objective function (3.1a) represent the contributions resulting from the fleet assignment and through assignment, respectively. The constraint (3.1b) ensures that each flight leg is assigned exactly one fleet type. The constraints (3.1c) and (3.1d) together with (3.1b) imply that each flight leg is assigned to another flight leg using a connection arc, and the two flight legs and the connection arc are assigned the same fleet type. The constraint (3.1e) ensures that the total number of planes of fleet type \( f \) in the assignment, which is the sum of the flows on arcs in \( S \), is no more than the available planes given by \( \text{size}(f) \). Observe that to compute the number of planes of a particular fleet type \( f \) used in a fleet schedule, we sum the flow of planes of that fleet type on the overnighing arcs.

In practice, the solution of ctFAM must also satisfy several additional constraints. These constraints incorporate aspects of maintenance routing and crew scheduling. To simplify the
explanation, we defer the detailed description of these constraints to Section 4. We also explain there how our algorithm needs to be modified to account for these constraints.

We note that the formulation (3.1) is not the most efficient formulation of the ctFAM. One can create a more compact formulation with fewer variables by using a slightly modified connection network in which we use ground nodes to consolidate connection arcs. This idea has been used in FAM (see for example, Hane et al. [1995]). We have not presented that formulation here primarily because the description of our neighborhood structure can be given more clearly with formulation (3.1). However, we tested a substantially smaller integer programming formulation of the ctFAM than the formulation (3.1) but that problem was also too large to be solved to optimality or near-optimality (using the current integer programming technology) for the national network of a large US airline. In the data available to us for a real-life instance of ctFAM, there were 1,609 flight legs and 13 fleet types. The resulting (compact) IP formulation had approximately 100,000 integer variables and 18,000 constraints. We could not solve problems of this magnitude using the commercially available IP solvers. We then focused on neighborhood search algorithms to solve ctFAM. Additional reasons for considering neighborhood search algorithms have earlier been described in Section 1. We next describe our neighborhood structure for the ctFAM

3.3 Swap Based Neighborhood Structure for ctFAM

We define our neighborhood structure with respect to a feasible solution \((x, y)\) of ctFAM integer programming formulation (3.1). In order to define the neighborhood structure, we introduce some additional definitions here.

**A-B Solution Graph:** The A-B solution graph, \(S^{AB}(x, y)\), is a subgraph of the connection network \(G = (N, E)\) and is defined with respect to a given fleeting and connection solution \((x, y)\) and a pair of fleet types A and B. Its node set, \(N(S^{AB}(x, y))\), and arc set, \(E(S^{AB}(x, y))\), are defined as follows:

\[
N(S^{AB}(x, y)) = \{i \in N: y_i^A = 1 \text{ or } y_i^B = 1\},
\]

\[
E(S^{AB}(x, y)) = \{(i, j) \in E: x_{ij}^A = 1 \text{ or } x_{ij}^B = 1\}.
\]

In other words, the A-B solution graph \(S^{AB}(x, y)\) is the subgraph of \(G\) whose node set comprises of the flight legs that are assigned fleet types A and B in the solution \((x, y)\), and the arc set comprises of the connections between those flight legs. We shall refer to a node in the A-B
solution graph as an $A$-node if $y^A_i = 1$ and $B$-node if $y^B_i = 1$. We shall refer to an arc in the $A-B$ solution graph as an $A$-arc if $x^A_{ij} = 1$ and $B$-arc if $x^B_{ij} = 1$.

**A-B Swaps:** Our neighborhood search structure uses the concept of $A-B$ swaps to define neighboring solutions. We first define an $A-B$ swap in a very general manner, one that permits a much larger neighborhood than we subsequently search. Given a feasible solution $(x, y)$ of ctFAM, and a pair of fleet types $A$ and $B$, we say that $(x', y')$ is an $A-B$ neighbor of $(x, y)$ if it is a feasible solution that differs only in the assignment of $A$-flights and $B$-flights. The operation of obtaining an A-B neighbor is called an $A-B$ swap. Figure 3-3(a) shows a part of the solution graph $S^{AB}(x, y)$ and Figure 3-3(b) shows the same part after the A-B swap has been performed. In the figure, we show $A$-nodes and $A$-arcs using regular lines, and $B$-nodes and $B$-arcs using dashed lines.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
</tbody>
</table>

(b)

**Figure 3-3. Illustrating an A-B swap.** (a) Part of the solution graph before the A-B swap. (b) Part of the solution graph after the A-B swap.

Observe that the A-B swap changes the fleet type of nodes 4 and 10 from $A$ to $B$ and changes the fleet type of nodes 3 and 6 from $B$ to $A$. Changing the fleet types of these nodes requires changing the connections too because we can connect nodes with the same fleet type only. The A-B swap must also ensure that the connections can be feasibly made, that is, for each connection arc $(i, j)$, the arrival time of flight $i$ is less than the departure time of flight $j$.

Recall that while defining $A-B$ swaps we require that we do not violate fleet size constraints for fleet types $A$ and $B$. Figure 3-4(a) shows a part of the A-B swap where the number of planes of a particular type used can increase. Suppose that flights 1 and 3 are flown by fleet type $A$ and flights 2 and 4 are flown by fleet type $B$. Assume that flights 1 and 2 arrive at
times 2 PM and 4 PM, respectively, and the flights 3 and 4 depart at 5 PM and 3 PM, respectively. Since flight 1 connects to flight 3 which leaves three hours later, the arc (1, 3) is not an overnighting arc. However, flight 2 arrives at 4 PM and connects to flight 4, which departs at 3 PM. Thus the arc (2, 4) is an overnighting arc. If we change the fleet types of flights 1 and 3 from A to B, and of flights 2 and 4 from B to A, as shown in Figure 3-4(b), then we increase the number of planes used for type A by one and decrease the number of planes used for type B by one. Our neighborhood search algorithm does not allow such swaps if it leads to infeasibilities. Note that if we change the fleet type of flight 1 from A to B, fleet type of flight 2 from B to A, and swap their connections, as shown in Figure 3-4(c), then both the new connection arcs (1, 4) and (2, 3) are not overnight arcs. This swap will reduce the number of planes used for type B by one. Our neighborhood search algorithm allows such swaps.

![Diagram](image_url)

**Figure 3-4. Effect of swaps on the number of planes used.**

The example shown in Figure 3-4 illustrates a very simple A-B swap; on the other hand, there can exist far more complex A-B swaps, that affect many more flights and connections. In principle, we could identify an improving $A-B$ neighbor of $(x, y)$ by solving a restricted integer program. We decided that this was computationally too intensive, and adopted a more efficient approach. Further, using our approach, more complicated objectives in addition to contribution can be considered, this is useful later in our multi-criteria optimization algorithm. We search a subset of $A-B$ neighbors of $(x, y)$ using network optimization. We next define the concept of $A-B$ improvement graph, which allows us to efficiently identify profitable $A-B$ swaps over a structured subset of the $A-B$ neighborhood.

### 3.3.1 A-B Improvement Graph

Before we discuss the creation of our improvement graph, we note that the $A-B$ solution graph satisfies the following cycle-based property: The solution graph as restricted to the A nodes is a union of node-disjoint cycles, and the solution graph is also the union of node-disjoint cycles.
Equivalently, each A node \( i \) has exactly one outgoing arc and exactly one incoming arc, and both these arcs have A-nodes as the other endpoint. In our swaps, we will be changing some A-nodes to B nodes and vice-versa. We will construct our A-B network in such a way that an improving cycle leads to a new solution with the above cycle-based property.

Let us first illustrate the simplest type of swap before moving to the more complex swaps permitted below. Consider two directed paths \( P \) and \( P' \) in the A-B solution graph both starting at the same time \( t \) and the same location \( L \), and both ending at the same time \( t' \) and the same location \( L' \), and such that \( P \) consists of A-flights and \( P' \) consists of B-flights. We can swap \( P \) and \( P' \), making all the flights of \( P \) into B-flights and making all of the flights of \( P' \) into A flights. To identify such path pairs, we could look for all paths of A-flights and all paths of B-flights starting at time \( t \) at location \( L \) and ending at time \( t' \) at location \( L' \). Talluri [1996] recognized that we could find these paths in a simpler manner by reversing the direction of all B arcs, and then looking for a directed cycle. By doing so, one also identifies many other cycles, but each of the cycles (if overnight arcs are excluded from the cycles) corresponds to a valid A-B swap. In our approach, we also reverse the arcs incident to B nodes.

An A-B improvement graph, \( G^{AB}(x, y) \), is constructed for a given fleeting and through solution \( (x, y) \) and a pair of fleet types A and B. Each arc \((i, j)\) in the A B improvement graph has an associated cost \( c_{ij} \). The A-B improvement graph satisfies the property that each directed cycle in it satisfying some constraints, called the validity constraints, corresponds to an A-B swap with respect to the solution \( (x, y) \), and the cost of the directed cycle equals the change in the fleeting and through costs. Consequently, a negative cost directed cycle satisfying the validity constraints gives a profitable A-B swap. We will subsequently refer to a directed cycle in \( G^{AB}(x, y) \) satisfying validity constraints as a valid cycle.

The node set of the A-B improvement graph is identical to that of the A-B solution graph. Hence it consists of A-nodes and B-nodes. Each arc \((i, j)\) in the improvement graph signifies that we switch the fleet types of nodes \( i \) and \( j \) from B to A or from A to B (whichever is applicable) and reconnect the flights so that the connections are between flights that are assigned the same fleet types. In our approach, we add an arc \((i, j)\) to the improvement graph whenever this change can be feasibly made without increasing the total plane count at the city \textit{arr-city}(i)\) if \( i \) is A-node and \textit{dep-city}(i)\) if \( i \) is a B-node. We define the cost \( c_{ij} \) of the arc \((i, j)\) to be the change in the fleeting and through costs resulting from the change. Table 3-1 summarizes the six types of arcs that can be added to the improvement graph. In the figure, we show an A-node or an A-arc using
regular lines, and a $B$-node or $B$-arc using dashed lines. The detailed explanation of these arcs is given next.

**Type 1 Arcs:** Consider an arc $(i, j)$ in the $A$-$B$ solution graph which is an $A$-arc such that $(i, j) \notin S$. We introduce the arc $(i, j)$ in the improvement graph which corresponds to changing the plane types of both the flights $i$ and $j$ from $A$ to $B$. Both flights $i$ and $j$ become $B$ flights, and we assume that their connection is maintained. The cost of the arc $(i, j)$, $c_{ij}$, is the sum of (i) the change in the fleeting cost when plane type of flight $i$ is changed from $A$ to $B$, and (ii) the change in the through revenues of the connection $(i, j)$ due to change in fleeting types. Notice that when computing $c_{ij}$ we include the change in the fleeting cost of flight $i$ only but not flight $j$. We do it because if we include the cost of changing the fleet types of both the nodes $i$ and $j$ in the cost of arc $(i, j)$, then when we sum the cost of arcs in a valid cycle, we will be double counting the changes in the fleeting costs. Since the arc $(i, j)$ does not belong to the set $S$, it does not affect the fleet size constraint.

**Type 2 Arcs:** A type 2 arc $(j, i)$ is introduced in the improvement graph for each $B$-arc $(i, j)$ in the $A$-$B$ solution graph such that $(i, j) \notin S$. This arc corresponds to changing the plane types of both the flights $i$ and $j$ from $B$ to $A$ and preserving the connection between the two flights. Notice that contrary to the case of type 1 arcs, we introduce the arc $(j, i)$ instead of arc $(i, j)$. The arcs are reversed as per the discussion above. The cost of the arc $(j, i)$ captures the change in the fleeting cost of flight $j$ and through costs of the connection arc $(i, j)$.

**Type 3 Arcs:** A type 3 arc $(i, l)$ is introduced in the improvement graph for every pair, $(i, j)$ and $(k, l)$, of $A$-arcs in the $A$-$B$ solution graph such that the change corresponding to it does not violate the fleet size constraints. The arc $(i, l)$ signifies changing the fleet types of both the flights $i$ and $l$ from $A$ to $B$. Since we can make connections between flights flown by the same fleet type, this change requires changing the connections too; we thus need to reconnect flight $i$ with flight $l$ and flight $k$ with flight $j$. The cost of the arc $(i, l)$ captures the change in the fleeting cost of flight $i$ and the change in the through costs due to reconnections. We point out that we add the arc $(i, l)$ to the improvement graph only if the corresponding change does not increase the number of planes of type $A$ and $B$. For example, we require that (i) $(i, l) \notin S$, and (ii) if $(k, j) \in S$ then either $(i, j) \in S$ or $(k, l) \in S$ or both. Observe that in the absence of requirement (i), the number of planes of type $B$ used could increase by 1 after the addition of arc $(i, l)$. Similarly, if requirement (ii) is not satisfied than the number of planes of type $A$ could increase by 1.
**Type 4 Arcs:** We introduce a type 4 arc \((j, k)\) in the improvement graph for every pair of \(B\)-arcs \((k, l)\) and \((i, j)\) in the \(A-B\) solution graph such that flight \(k\) can connect to flight \(j\) and flight \(i\) can connect to flight \(l\), and such that the change corresponding to it does not violate the fleet size constraints. The cost of the arc \((j, k)\) includes the costs of changing fleet type of flight \(j\) and \(k\) from \(B\) to \(A\) and the change in the through costs due to the reconnections. Notice that a type 4 arc is similar to a type 3 arc except that the direction of the arc is reversed. The requirements on the connection arcs are the same as those in type 3 arcs.

<table>
<thead>
<tr>
<th>Type of Arc</th>
<th>Before the change in the solution graph</th>
<th>After the change in the solution graph</th>
<th>Corresponding arc in the improvement graph</th>
<th>Cost of the arc in the improvement graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>(i) → (j)</td>
<td>(i) → (j)</td>
<td>(i) → (j)</td>
<td>(c_{ij} = (c^A_i + d^A_{ij}) - (c^B_j + d^B_{ij}))</td>
</tr>
<tr>
<td>Type 2</td>
<td>(i) → (j)</td>
<td>(i) → (j)</td>
<td>(i) → (j)</td>
<td>(c_{ij} = (c^A_j + d^A_{ij}) - (c^B_i + d^B_{ij}))</td>
</tr>
<tr>
<td>Type 3</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (l)</td>
<td>(c_{il} = (c^B_i + d^B_{il} + d^A_{kl}) - (c^A_i + d^A_{il} + d^A_{kl}))</td>
</tr>
<tr>
<td>Type 4</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (l)</td>
<td>(c_{il} = (c^A_i + d^A_{il} + d^B_{kl}) - (c^B_i + d^B_{il} + d^B_{kl}))</td>
</tr>
<tr>
<td>Type 5</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (k)</td>
<td>(c_{ik} = (c^B_i + d^B_{il} + d^A_{kj}) - (c^A_i + d^A_{il} + d^A_{kj}))</td>
</tr>
<tr>
<td>Type 6</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (k) (j) (l)</td>
<td>(i) → (l)</td>
<td>(c_{il} = (c^A_i + d^A_{il} + d^B_{kl}) - (c^B_i + d^B_{il} + d^B_{kl}))</td>
</tr>
</tbody>
</table>

Table 3-1. Different types of arcs in A-B improvement graph.

**Type 5 Arcs:** We introduce a type 5 arc \((i, k)\) in the improvement graph for every pair of arcs \((i, j)\) and \((k, l)\) in the \(A-B\) solution graph such that \((i, j)\) is an \(A\)-arc, \((k, l)\) is a \(B\)-arc, and the change corresponding to \((i, k)\) does not violate the fleet size constraint, and such that flight \(i\) can connect to flight \(l\) and flight \(k\) can connect to flight \(j\). The arc \((i, k)\) corresponds to changing the fleet type
of leg \(i\) from \(A\) to \(B\) and of leg \(k\) from \(B\) to \(A\). Changes in the fleet types require changing the through assignments too; leg \(i\) connects to leg \(l\), and leg \(k\) connects to leg \(j\) after the swap. The cost of the arc \((i, k)\) captures the cost of the change in the fleet assignment of leg \(i\) and the change in through connection costs due to the reconnections. To ensure that the number of planes of type \(A\) and \(B\) do not increase, we use the following two requirements on the connection arcs involved: (i) if \((i, l) \in S\) then \((k, l) \in S\), and (ii) if \((k, j) \in S\) then \((i, j) \in S\).

**Type 6 Arcs:** A type 6 arc is similar to a type 5 arc but with its orientation reversed. We introduce the arc \((j, l)\) in the improvement graph for every pair of arcs \((i, j)\) and \((k, l)\) in the \(A-B\) solution graph such that \((i, j)\) is a \(B\)-arc, \((k, l)\) is an \(A\)-arc, and the change corresponding to it does not violate the fleet size constraints. In addition, we require that flight \(i\) can connect to flight \(l\) and flight \(k\) can connect to flight \(j\). The cost of the arc \((j, l)\) includes change in the fleeting cost of flight \(j\) and the change in through costs due to reconnections. To ensure that the number of planes of type \(A\) and \(B\) do not increase, the requirements on the connection arcs are the same as those in type 5 arcs.

We will identify \(A-B\) swaps by finding valid cycles, which we define next. Recall that in the \(A-B\) solution graph, each node \(i\) is connected to a unique node \(j\) through the arc \((i, j)\) and is also connected from a unique node \(k\) through the arc \((k, i)\). For each node \(i\), we define its "mate" as follows: (i) if \(i\) is an \(A\)-node and \((i, j)\) is an arc in the \(A-B\) solution graph, then \(mate(i) = j\); and (ii) if \(i\) is a \(B\)-node and \((k, i)\) is an arc in the \(A-B\) solution graph, then \(mate(i) = k\).

**Valid Cycles:** A directed cycle \(W\) in the \(A-B\) improvement graph is said to be a valid cycle if it satisfies the following property for every node \(i \in W\): \(mate(i) \not\in W\) unless \((i, mate(i)) \in W\).

The intuitive reason we do not allow valid cycles to contain both the nodes \(i\) and \(mate(i)\) in the valid cycles unless \((i, mate(i)) \in W\) is as follows. The purpose of constructing the improvement graph is that a directed cycle in it defines an \(A-B\) swap and that the cost of the cycle equals the cost of the \(A-B\) swap. A directed cycle, which is not a valid cycle, cannot ensure this property. Consider, for example, a directed cycle \(W\) in the improvement graph which contains a Type 5 arc \((i, k)\) (see Table 3-1). Let node \(j = mate(i)\) and \(l = mate(k)\). The arc \((i, k)\) signifies the change that flight \(i\) reconnects to flight \(l\) and flight \(k\) reconnects to flight \(j\), and the cost of the arc \((i, k)\) captures the cost of these changes. If we allow the cycle \(W\) to visit node \(j\) or node \(l\), then we will not be able to preserve the change indicated by arc \((i, k)\) and its cost will become incorrect. Thus, if we make arc \((i, k)\) part of the cycle, then we must disallow the mates of these nodes from being a part of the cycle. This difficulty arises when we include arcs of Type 3, 4, 5, or 6 in the cycle \(W\). This difficulty does not arise when we make an arc of Type 1 or Type 2 to be the part of the
cycle in which case we include both the node $i$ and its mate. Hence the "unless" clause in the definition of the valid cycle.

Figure 3-5. Valid cycle and the corresponding A-B swap.

We will now give a numerical example that a valid cycle in the improvement graph gives an A-B swap; this example will be followed by a formal proof of the general result. Consider the part of the A-B solution graph shown in Figure 3-5(a). When we construct the improvement graph, it will contain the valid cycle $W = 3-4-10-7-8-6-3$ shown in Figure 3-5(b). This cycle denotes the A-B swap, which when performed, produces the solution shown in Figure 3-5(c). Observe that all the nodes in the cycle switch their fleeting types. The arc (3, 4) in the valid cycle $W$ is a Type 6 arc, this arc signifies that nodes 3 and 4 switch their fleeting types and the inbound flights into these nodes swap their connections. The next arc (4, 10) in the cycle is a Type 3 arc which changes fleeting types and connections. The next arc (10, 7) is a Type 1 arc; it only changes the fleeting. The next arc (7, 8) in the cycle is a Type 5 arc which captures the fact that the outbound flights from these two nodes swap their flights. Finally, the two arcs (8, 6) and (6, 3) are Type 2 arcs which change the fleeting types but not the connections. Figure 3-5(c) shows the
same part of the solution graph when the corresponding A-B swap has been performed. We now prove the general result regarding the relation between valid cycles and A-B swaps.

**Theorem 3.1.** Each valid cycle in the A-B improvement graph \(G^{AB}(x, y)\) gives an A-B swap with respect to the solution \((x, y)\).

**Proof:** We note that any A-B swap results in a solution satisfying the constraints (3.1b) since any flight leg that has fleet types A or B assigned to them will have a fleet type (A or B) after the swap. The constraints (3.1e) are also satisfied since the changes corresponding to each arc in the A-B improvement graph ensure that the fleet size constraints (3.1e) are satisfied. We shall now show that the constraints (3.1c) and (3.1d) are also satisfied. This amounts to showing that the cycle-based property is maintained by the swap. Let \(W\) denote the valid cycle. Let \(i\) be a node of the A-B solution graph. We assume inductively that node \(i\) has one incoming arc and one outgoing arc in the current solution, and these arcs join node \(i\) to nodes of the same fleet type. We want to prove that this property is satisfied after the A-B swap. Our proof relies on the consideration of a number of cases. We show that the property holds for the A-nodes affected by the swap. A similar argument can be made for the B-nodes.

Suppose first that \(i \in W\) and that \(i\) is an A-node. We consider first the node that directly follows node \(i\) in \(W\). We will show that after the swap, there is a B-node that directly follows node \(i\) in the resulting A-B solution graph. If \((i, j)\) is of type 1, then arc \((i, j)\) is a B-arc in the A-B solution graph after the swap. If \((i, l)\) is of type 3, then arc \((i, l)\) is a B-arc in the solution graph after the swap. If arc \((i, k)\) is of type 5, then \((i, l)\) is a B-arc in the solution graph after the swap. We also note that cases 2, 4 and 6 are not applicable to the arcs leaving an A-node.

We now consider the node that directly precedes an A-node \(r\) in \(W\). We will show that after the swap, there is a B-node that directly precedes node \(r\) in the resulting A-B solution graph. If \((i, j)\) is of type 1 (in this case, \(r = j\) in Table 3-1), then \((i, r)\) is a B-arc in the A-B solution graph after the swap. If \((i, l)\) is of type 3, (in this case, \(r = l\) in Table 3-1), then \((i, r)\) is a B-arc in the A-B solution graph after the swap. If \((j, l)\) is of type 6, (in this case, \(r = l\) in Table 3-1), then \((i, r)\) is a B-arc in the A-B solution graph after the swap. We have just established that for an A-node in \(W\), there is exactly one outgoing B-arc and exactly one incoming B-arc after the swap. A similar argument can be made for the B-nodes in the cycle \(W\).

We now consider nodes that are not in \(W\) and are affected by the swap. In cases 1 and 2, there are no such nodes. In case 3, node \(j\) has its incoming arc changed from \((i, j)\) to \((k, j)\), and node \(k\) has its outgoing arc changed from \((k, l)\) to \((k, j)\), and the cycle property remains satisfied.
after the swap. (We know that \( j \not\in W \), and \( k \not\in W \) because \( W \) is valid). In case 4, node \( l \) has its incoming arc changed from \((k, l)\) to \((i, l)\), and node \( i \) has its outgoing arc changed from \((i, j)\) to \((i, l)\), and the cycle property remains satisfied after the swap. (We know that \( i \not\in W \), and \( l \not\in W \), because \( W \) is valid.) In case 5, the \( A \)-node \( j \) has its incoming arc changed from \((i, j)\) to \((k, j)\), and the \( B \)-node \( l \) has its incoming arc change from \((k, l)\) to \((i, l)\), and the cycle property remains satisfied after the swap. (We know that \( j \not\in W \), and \( l \not\in W \), because \( W \) is valid.) Finally, in case 6, the \( B \)-node \( i \) has its outgoing arc changed from \((i, j)\) to \((i, l)\), and the \( A \)-node \( k \) changes its outgoing arc from \((k, l)\) to \((k, j)\), and the cycle property remains satisfied after the swap. (We know that \( i \not\in W \), and \( k \not\in W \), because \( W \) is valid.) This completes the proof of the theorem.

We have shown that we can identify \( A-B \) swaps by finding valid cycles in the \( A-B \) improvement graph. However, this problem is in general NP-hard for an arbitrary graph. We show this result next.

3.3.2 Complexity of finding valid cycles

In order to show the hardness result, we first formalize the problem of finding a negative cost cycle on a graph as follows.

**Negative Cost Valid Cycle Problem**

**Input:** A graph \( G = (N, A) \), arc costs \( c : A \to R \), and a function \( \text{mate} : N \to N \) such that for \( i \in N \), \((i, \text{mate}(i)) \in A \) and \( \text{mate}(i) \neq \text{mate}(j) \) for \( j \neq i \).

**Question:** Is there a negative cost valid cycle \( W \) in \( G \) (that is, is there a cycle \( W \) such that for every node \( i \in W \): \( \text{mate}(i) \not\in W \) unless \((i, \text{mate}(i)) \in W \))? 

We refer to an input instance of the problem as a *yes instance* if the answer to the question is yes. It is easy to see that the negative cost valid cycle problem is in NP since a negative cost valid cycle is a succinct certificate for the yes instances. In order to prove the NP-completeness of the negative cost valid cycle problem, we shall provide a polynomial time transformation from another problem called the *path through forbidden pairs*, which is known to be an NP-complete problem (see, for example, Garey and Johnson [1979]).

**Path through Forbidden Pairs Problem**

**Input:** A graph \( G' = ([1, 2, ..., 2n], A') \) with \( 2n+1 \) nodes for some \( n > 0 \).

**Question:** Is there a path \( P \) from node 1 to node \( 2n+1 \) in \( G' \) satisfying the following property: for each \( k = 1, 2, ..., n-1 \), the nodes \( 2k, 2k+1 \) cannot simultaneously belong to the path \( P \)?
Given an input graph $G'$ for the path through forbidden pairs problem, we construct an input instance of the negative cost valid cycle problem as follows. The input graph for our instance is $G = (N, A)$, where $N = \{0, 1, 2, ..., 2n\}$, $A = A' \cup S \cup \{(2n, 1)\} \cup \{(0, 2n)\}$ and $S = \{(2k, 2k+1): k = 1, ..., n-1\} \cup \{(2k+1, 2k): k = 0, ..., n-1\}$. The arcs in the set $S$ and the arc $(0, 2n)$ are added so that we can define the mate function in the desired manner. For $k = 1, 2, ..., n-1$, we define $mate(2k) = 2k+1$ and $mate(2k+1) = 2k$. We set $mate(0) = 2n$, $mate(2n) = 1$, and $mate(1) = 0$. The reader can verify that our mate function satisfies the required conditions. We define the arc costs $c$ as follows:

$$
    c_{ij} = \begin{cases} 
        0 & \text{if } (i, j) \in A' \\
        1 & \text{if } (i, j) \in S \\
        -1 & \text{if } (i, j) = (2n, 1) 
    \end{cases}
$$

It is easy to see that the instance above can be constructed in time polynomial in the size of $G'$.

**Theorem 3.2.** The instance $(G, c, mate)$ is a yes instance for the negative cost valid cycle problem if and only if the graph $G'$ is a yes instance for the path through forbidden pairs problem.

**Proof:** If a path $P$ exists in $G'$ from node 1 to node $2n$ such that it satisfies the required condition for path through forbidden pairs problem, then by the definition of $c$, the cost of the path $P$ is 0 in the instance $(G, c, mate)$. Further, by the construction of the mate function, the cycle obtained by adding the arc $(2n, 1)$ to path $P$ is a valid cycle. Hence, if $G'$ is a yes instance of the path through forbidden pairs problem, then $(G, c, mate)$ is a yes instance of the negative cost valid cycle problem.

Conversely, if $W$ represents a negative cost valid cycle in the instance $(G, c, mate)$ then it must contain the arc $(2n, 1)$ and none of the arcs $(i, j) \in S$ can be in the cycle $W$. By the construction of the mate function, this implies that for $k = 1, ..., n-1$, the nodes $2k$ and $2k+1$ cannot be in the cycle $W$ simultaneously. Hence, the path obtained by removing the arc $(2n, 1)$ from the cycle $W$ satisfies the condition for the path through forbidden pairs problem in the graph $G'$. This proves our result.

Using the previous theorem and our construction of the instance $(G, c, mate)$, we observe that there is a polynomial time transformation from any instance of the path through forbidden pairs problem to an instance of the negative cost valid cycle problem. Therefore, the negative cost valid cycle problem is an NP-complete problem.
3.3.3 Identifying A-B swaps

We have shown that we can identify A-B swaps by finding negative cost valid cycles in the A-B improvement graph. However, this problem is hard problem in general for an arbitrary graph as shown in the last section. Fortunately, this problem was typically solved in a fraction of second using CPLEX in our benchmark cases. We will next model the problem of finding a union of node-disjoint valid cycles as an integer programming problem.

We first introduce some notation related to the integer program. Let $N' = N(G^{AB}(x, y))$ denote the set of nodes and $E' = E(G^{AB}(x, y))$ denote the set of arcs in the A-B improvement graph. We associate a binary variable $w_{ij}$ with each arc $(i, j) \in E'$. This variable takes value 1 if arc $(i, j)$ is present in some valid cycle, and takes value 0 otherwise. We give the IP formulation next followed by its explanation.

Minimize $\sum_{(i,j) \in E'} c_{ij}w_{ij}$ (3.2a)

subject to

$\sum_{\{j \mid (j,i) \in E'\}} w_{ji} - \sum_{\{j \mid (i,j) \in E'\}} w_{ij} = 0, \quad \text{for all } i \in N'$, (3.2b)

$\sum_{\{j \mid (j,i) \in E' \backslash \{(i,\text{mate}(i))\}\}} w_{ij} + \sum_{\{j \mid (\text{mate}(i),j) \in E'\}} w_{\text{mate}(i),j} \leq 1, \quad \text{for all } i \in N'$, (3.2c)

$w_{ij} \in \{0,1\}, \text{ for } (i,j) \in E'$.

(3.2d)

In the above formulation (3.2), the constraints (3.2b) and (3.2d) imply that the solution is a 0-1 circulation. This 0-1 circulation can be decomposed into unit flows along directed cycles. The constraints (3.2c) ensure that the flow passing through each node $i$ plus the flow passing through the node $\text{mate}(i)$ is at most 1, which implies that the resulting flow will not pass through both the nodes $i$ and $\text{mate}(i)$. An exception to this rule occurs when flow takes place over the arc $(i, \text{mate}(i))$ in which case both the nodes $i$ and $\text{mate}(i)$ can be visited. It is easy to see that a feasible solution of (2) gives a set of valid cycles. If the improvement graph does not contain any negative cost valid cycles, then $w = 0$ will be an optimal solution of (3.2). If the improvement graph contains a negative cost valid cycle, then an optimal solution $w^*$ of (3.2) will give a collection of valid cycles with the minimum total cost. Using flow decomposition (see, for example, Ahuja, Magnanti, and Orlin [1993]), we can decompose $w^*$ into a set of node-disjoint cycles. Each of these cycles has a negative cost or a cost of 0. The negative cost cycles include an associated profitable A-B swap.
3.3.4 Neighborhood Search Algorithms

We are now in a position to describe our neighborhood search algorithm for ctFAM. Figure 3-6 describes the generic version of our algorithm. Our neighborhood search algorithm for ctFAM performs passes over all fleet pairs A and B and performs profitable A-B swaps. The algorithm terminates when in one complete pass it finds that no profitable swap exists for any pair of fleet types A and B.

\textbf{algorithm} ctFAM neighborhood search; \\
\textbf{begin} \\
\hspace{1em} solve FAM to determine the optimal fleet assignment y; \\
\hspace{1em} solve TAM to determine the optimal connections x for the fleet assignment y; \\
\hspace{1em} \textbf{repeat} \\
\hspace{2em} \textbf{for} each pair of the fleet types A and B \textbf{do} \\
\hspace{3em} \textbf{begin} \\
\hspace{4em} construct the A-B solution graph \( S^{AB}(x, y) \); \\
\hspace{4em} construct the A-B improvement graph \( G^{AB}(x, y) \); \\
\hspace{4em} \textbf{while} the A-B improvement graph \( G^{AB}(x, y) \) contains negative cost valid cycles \textbf{do} \\
\hspace{5em} \textbf{begin} \\
\hspace{6em} determine a set \( W \) of negative cost valid cycles in the A-B improvement graph; \\
\hspace{6em} perform A-B swaps corresponding to \( W \); \\
\hspace{6em} update the A-B solution graph \( S^{AB}(x, y) \); \\
\hspace{5em} \textbf{end}; \\
\hspace{4em} \textbf{end}; \\
\hspace{2em} \textbf{until} for every pair of fleet types A and B, \( G^{AB}(x, y) \) contains no negative cost valid cycle; \\
\textbf{end};

\textbf{Figure 3-6. The neighborhood search algorithm for ctFAM.}

3.3.5 Alternate Definition of A-B Improvement Graph

In this section, we present an alternative definition of the A-B improvement graph. This definition is motivated by the following observation. In our definition of the arcs in the improvement graph, we only perform reconnections between the flights whose connections have been affected. However, it may be possible to achieve better connections once the fleet type of some arrival or departure flights has been modified. For example, consider an arc \((i, j)\) of Type 1 in the A-B improvement graph. In a swap involving this arc, we change the fleet types of flight legs \(i\) and \(j\) from A to B and keep the connection \((i, j)\). It is possible that there is another B-arc \((k, l)\) between flights \(k\) and \(l\) such that breaking connections \((i, j)\) and \((k, l)\) and adding connections \((i, l)\) and \((k, j)\) gives a better through revenue. In this case, the benefit of swapping the fleet types of flights \(i\) and \(j\) is higher than that represented by the cost of the arc in the A-B improvement graph.

One possibility to measure the benefit of changing the fleet types of flights \(i\) and \(j\) is to perform the change and then solve a bipartite assignment problem to find a new optimal set of through connections at \(\text{arr-city}(i) = \text{dep-city}(j)\). However, the problem with using this measure
is that in order to ensure the *correctness* of arc costs in a swap, we have to define valid cycles such that they are incident to each city at most once. The restriction is necessary because the arc costs are computed assuming that only $i$ and $j$ are modified and all the other arrivals and departures at the city are unchanged. This results in a very restricted neighborhood structure.

We considered another possibility which is less restrictive and provides about the same accuracy of change in through benefit as that of solving the through connection problem optimally at a city. A *bank* is defined as time interval at a city such that a sequence of consecutive arrivals is followed by a sequence of departures during this time interval such that the arrivals can only connect to the departures. We note that if the schedule at the *arr-city*$(i)$ is split into several banks, then we only need to re-compute the through assignment for the arrivals and departures in the bank where flights $i$ and $j$ belong. Further, we only need to restrict the cycles to make at most one change in each bank.

In real-life schedules, the set of arrivals and departures at a city does not exactly follow a bank structure. However, it is quite close to it. Typically, a set of arrivals is followed by a set of departures but the number of arrivals is not always equal to the number of departures in consecutive sets. Therefore, a small number of arrivals have to connect to a departure in a later set and a small number of departures receive planes from earlier set of arrivals. We analyzed the real-life data provided to us by a major US airline and partitioned the day into several periods. The boundary of each period is a time when there are very few planes on the ground. Therefore, each period “approximately represents” a bank. We illustrate this through the example in Figure 3-7. We have numbered the flights in the chronological order of their arrival or departure times. For example, flight 4 departs before the arrival of flight 5. We assume that all these flights have the same fleet type so that they can connect any subsequent departure. Note that in Figure 3-7, although there are clear clusters of arrivals and departures but there is no exact bank structure. However, we can treat the flights from 1 to 7 as one approximate bank as there is only one arrival flight in this set that will need to connect to a subsequent set of departures.

$$
\begin{array}{cccccccc}
1 & 2 & 3 & 5 & 8 & 10 & 11 & \text{Arrivals} \\
\end{array}
$$

$$
\begin{array}{cccccccc}
4 & 6 & 7 & 9 & 12 & 13 & 14 & \text{Departures} \\
\end{array}
$$

*Figure 3-7. Illustration of an approximate bank structure.*
Using our analysis, we partitioned the set of arrivals and departures at major hub cities for the airline into approximate banks. We construct the alternative A-B improvement graph, $G^{bank}$, using the approximate bank information as follows.

The set of nodes in $G^{bank}$ is the same as the A-B improvement graph. Note that the changes corresponding to each of the Type 1, ..., Type 6 arcs in the A-B improvement graph only modify connections at a single city. Therefore, we can associate each arc with the city where it changes the connections. We consider the Type 1, ..., Type 6 arcs of the A-B improvement graph that only modify connections at cities that are not major hubs (where we do not define approximate banks). For these arcs, we use the arc costs as described in their definitions in Section 3.2. For the hub cities, we consider arcs of Type 1, ..., Type 6 only between arrivals and departures belonging to the same approximate bank. For example, suppose that in Figure 3-7 the approximate bank contains flights 1 to 7, all flights have fleet type A, flight 3 connects to flight 9, flight 5 connects to 7, and flight 8 connects to 12. In this case we shall consider the Type 3 arc (3, 7) but not consider (3, 12). Even though in both the cases, we change connections between flights in separate banks, in the first case we are swapping fleet types between flights in the same bank where as in the second case the flights belong to different banks. The other difference in the construction is in the way we compute the change in through benefits resulting from the fleet type changes corresponding to each of these arcs. Instead of computing the benefit by just considering re-connections between affected arrivals/departures, we setup at most two bipartite assignment problems (one for fleet type A and another for fleet type B), using which we find the optimal set of through connections only between arrivals and departures within the approximate bank. We associate the arcs between nodes in an approximate bank with the bank.

One can define valid cycles for this construction of the improvement graph such that the cycles correspond to A-B swaps and the cost of the cycle is equal to the change in objective function from the swap. We implemented a neighborhood search algorithm using this idea of improvement graph. Our computational results indicated that even using approximate banks makes the definition of valid cycles too restrictive. The resulting neighborhood structure is not as large as the one defined by A-B improvement graph and the definition of valid cycles given in Section 3.2. Hence, the solutions obtained by this alternative approach were not satisfactory.

### 3.4 Implementation Details

We now describe some important details of the implementation of our neighborhood search algorithm.
**Identifying Negative Cost Valid Cycles:** To identify a negative cost valid cycles in the $A$-$B$ improvement graph, we solve the IP problem (2) using the commercial solver CPLEX 6.5 and do not run it up to optimality as it takes too much time. The solver solves the IP problem using a branch and bound algorithm. We keep track of the number of integer solutions found by the branch and bound algorithm and stop it as soon as it finds an optimal solution or finds 10 integer solutions, whichever occurs earlier. We use the best integer solution found, decompose it into node-disjoint profitable valid cycles, and perform $A$-$B$ swaps corresponding to each valid cycle. Our neighborhood search needs only one negative cost valid cycle to improve the current solution and it need not be the best valid cycle. Consequently, we may terminate the IP branch-and-bound whenever it has found a negative cost valid cycle.

**Updating Flight Connections:** Our neighborhood search algorithm starts with a solution where the flight connections (given by the solution $x$) are optimal for the specified fleet assignment (given by the solution $y$). Each $A$-$B$ swap performed by the algorithm changes the fleet assignment of some flight legs and may also change flight connections. As noted in Section 3.5, the modified flight connections $x'$ may not be optimal for the modified fleet assignment $y'$. Hence, a possibility to improve the solution value exists by changing connections without changing the fleet assignment. Our algorithm checks for these possibilities and makes switches when improvements are possible. It solves a TAM for the fleet types $A$ and $B$ at every city where $A$-$B$ swap has changed the fleet assignment. This step takes only a small proportion of the overall computational time, and occasionally improves the solution value substantially.

**Tabu Search:** The algorithm described in Figure 3.7 is a pure local search algorithm. We also implemented a tabu search algorithm (see Glover and Laguna [1997] for details on tabu search). We implemented a version of the tabu search that incorporated the short-term memory aspect of tabu search; that is, we used tabu lists. Whenever we perform an A-B swap that worsens the objective value, we choose a random flight whose fleet type was modified, and restrict our algorithm not to modify its fleet type again for next $p$ iterations. This can be done by removing the node corresponding to the flight in the A-B improvement graph. We selected the value $p = 5$ after some experimentation. To ensure that the tabu search approach would generate at least one valid cycle at each iteration, we added the constraint $\sum_{(i,j) \in E} w_{ij} \geq 1$ to (2.2). In order to cut down on some of the unproductive searching, we restricted our search to a small subset of “promising” $A$-$B$ pairs of fleet types. In addition, when we solved (2.2) by the IP solver, we enumerated 100 integer solutions only and the best solution among them determined the set of $A$-$B$ swap performed. Each flight involved in an $A$-$B$ swap is made tabu for the next 5 iterations. For any
pair of fleet types $A$ and $B$, we apply the tabu search algorithm for 100 iterations and record the best solution found. We use this best solution as the starting solution for the next pair of fleet types.

**Handling Additional Constraints:** In Section 2, we noted that the solutions of ctFAM also need to satisfy a set of three additional constraints. We briefly discuss these three kinds of additional constraints that are enforced on a fleet assignment, and are closely related to constraints faced in other fleet scheduling problems as well. They are related to maintenance and crew scheduling. The rational behind these constraints is discussed in detail in Clarke et. al. [1996]. Our ctFAM includes these additional constraints and solutions obtained by our algorithms satisfy these constraints.

1. **Service Maintenance Constraints.** For each fleet type $f$, the service maintenance constraints specify a set of maintenance stations at which a certain desired percentage of aircraft of fleet type $f$ must be on the ground at midnight. These constraints can be easily incorporated into the integer programming formulation of the ctFAM. Let $S_f$ denote the set of connection arcs such that a plane using one of these arcs is on the ground at midnight at one of the maintenance stations for fleet type $f$. Let $p_f$ denote the desired percentage of aircraft of fleet $f$ at its maintenance stations. Using this notation, we can write the service constraint for fleet type $f$ in the integer programming formulation (1) as: \[ \sum_{(i,j) \in S_f} x_{ij}^f \geq \frac{p_f}{100} \text{size}(f). \]

2. **Aircraft Balance Check Constraints.** These constraints model the longer balance check maintenance (10-12 hours) done on the aircraft. An aircraft balance check constraint specifies a station $s$, an interval of the day $(a, b)$, duration of the check $D$, a list $L$ of fleet types, and a number $K$ such that $K$ planes from the fleet type list $L$ must receive balance check of duration $D$ within the interval $(a, b)$ at station $s$. In order to incorporate this constraint in the integer program (3.1), let $Q$ denote the set of connection arcs such that a plane taking any of the connections in $Q$ is on the ground at station $s$ for a duration of at least $D$ units between the interval $(a, b)$. The aircraft balance check constraint can then be specified as: \[ \sum_{(i,j) \in Q} \sum_{f \in L} x_{ij}^f \geq K. \] There may be several such constraints, one per type of balance check constraint.

3. **Crew Block Hour Constraints.** The block hour of a flight is the time that elapses between the flight leaving the gate at the departure city and entering the gate at the arrival city. The crew block hour constraint requires that the total block hours of all the flight legs that are assigned to a given subset of fleet types should be bounded. Since each crew is typically trained for a subset of
aircraft types, these constraints ensure that none of the crews is over or under-utilized. For each flight node \(i\), let \(b_i\) denote the block time of the flight leg associated with it. Let \(L\) represent the set of fleet types involved in the crew block hour constraint and \(m, M\), respectively, denote the lower and upper bounds on the total block hours allowed for all the planes belonging to the fleet types in \(L\). The crew block hour constraint can be incorporated in the formulation (3.1) as
\[
m \leq \sum_{f \in L} \sum_{i \in N^f} b_i y_t^f \leq M.
\] There may be several such constraints, one per type of balance check constraint.

To incorporate these additional constraints, we first added these constraints to FAM and to TAM so that the initial solution constructed by using these models satisfies these constraints. Subsequently, we ensured that each A-B swap performed by the algorithm maintains these additional constraints. These constraints (in particular, service maintenance and crew block hour) involve arrivals and departures at several cities. Since the arcs in our A-B improvement graph only look at changes at one city, these constraints cannot be handled in the improvement graph itself. Instead, they can be incorporated into the integer program (2.2) to ensure that the A-B swaps identified by the algorithm do not violate these constraints. Unfortunately, this method makes the integer program (2) substantially much more difficult to solve, and not practical for a neighborhood search approach. However, we also made the following discovery: approximately 60% of all the improving valid cycles for the integer program (2.2) in our algorithm satisfy the additional constraints. Using this observation, we solve (2.2) using the IP solver with no additional constraints. We perform an A-B swap as determined by the solution of the integer program when it also satisfied these additional constraints.

3.5 Computational Testing for FAM and ctFAM

In this section, we present computational results of our neighborhood search algorithms for the ctFAM. We programmed our algorithms in the C programming language and tested them on a Pentium 4 1.4 GHz processor computer with 512MB RAM and a Linux operating system. We tested our algorithms on the real-life data provided by a major US airline.

Recall that our swap based neighborhood structure is applicable to the fleet assignment model as well, where we assume that the through connection benefits are zero for all possible connections. We tested our local improvement and tabu search algorithms on four problems: (i) FAM without maintenance constraints; (ii) ctFAM without maintenance constraints; (iii) FAM with maintenance constraints; and (iv) ctFAM with maintenance constraints. The starting solutions for these problems were obtained by solving FAM followed by TAM as done in the
current sequential approach. The integer programming model for FAM was run up to 30 minutes as is the practice at that airline. The best integer solution obtained became the starting point of our neighborhood search algorithms.

Table 3-2 gives the changes in FAM and through contributions by the use of local improvement algorithm. Our objective in the neighborhood search algorithms was to maximize the total fleet assignment and through contribution. The improvements obtained are reported on an annual basis. The results for tabu search algorithm are reported in Table 3-3. We also investigated some other properties of the algorithms. For the local improvement algorithm, we noted the number of improving A-B swaps performed, average cost of each swap, the average number of flights involved in each swap, the total number of flights whose fleeting was changed, and the number of passes over all pairs of fleet types (A, B) performed by the algorithm. For the tabu search algorithm, we also noted the number of swaps that worsened or improved the objective function as all the A-B swaps do not improve the objective function in this case. We report the results for local improvement algorithm in Table 3-4 and tabu search algorithm in Table 3-5.

We observe that both the local and tabu search algorithms improve the integer programming solutions quickly in a fairly reasonable time. In particular, the local improvement algorithm terminates in a matter of seconds. These algorithms are able to improve the FAM solution. The reason that they can improve the FAM solution is that the FAM solution was not solved to optimality, but terminated with a nearly optimal solution. Typically, the integer programming method for FAM obtains a near optimal solution quickly but takes a long time to improve it after that. The neighborhood search algorithms found the possible improvements quickly. This suggests that neighborhood search algorithms can be used as a supplement to the integer programming techniques for FAM.

Our algorithms for ctFAM with maintenance constraints improved the integer programming solution by a substantially larger amount than ctFAM without the maintenance constraints. Maintenance constraints make the integer programming problem harder and the solution produced by the IP software leaves more room for possible improvement which our neighborhood search algorithm is able to obtain. The results also highlight the inherent dependency between through and fleet assignment problems. We note that in the case of ctFAM instances, our algorithms were able to substantially improve the through contribution by degrading the fleet contribution by a small amount. The current sequential approach is not able to
do this. Given the size of the combined problem, a neighborhood search approach seems suitable for solving the ctFAM.

<table>
<thead>
<tr>
<th>Model</th>
<th>Local Improvement Algorithm</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Changes in fleeting contribution (in millions)</td>
<td>Changes in though contribution (in millions)</td>
<td>Changes in total contribution (in millions)</td>
<td>Running time (sec.)</td>
<td></td>
</tr>
<tr>
<td>FAM without maintenance</td>
<td>0.55</td>
<td>0</td>
<td>0.55</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FAM with maintenance</td>
<td>3.84</td>
<td>0</td>
<td>3.84</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>ctFAM without maintenance</td>
<td>-5.25</td>
<td>22.40</td>
<td>17.15</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>ctFAM with maintenance</td>
<td>-0.94</td>
<td>27.80</td>
<td>26.86</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-2. Improvements obtained by the local improvement algorithm.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tabu Search Algorithm</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Changes in fleeting contribution (in millions)</td>
<td>Changes in though contribution (in millions)</td>
<td>Changes in total contribution (in millions)</td>
<td>Running time (sec.)</td>
<td></td>
</tr>
<tr>
<td>FAM without maintenance</td>
<td>1.77</td>
<td>0</td>
<td>1.77</td>
<td>144</td>
<td></td>
</tr>
<tr>
<td>FAM with maintenance</td>
<td>3.88</td>
<td>0</td>
<td>3.88</td>
<td>299</td>
<td></td>
</tr>
<tr>
<td>ctFAM without maintenance</td>
<td>-10.00</td>
<td>35.20</td>
<td>25.20</td>
<td>1543</td>
<td></td>
</tr>
<tr>
<td>ctFAM with maintenance</td>
<td>-2.12</td>
<td>31.77</td>
<td>29.65</td>
<td>380</td>
<td></td>
</tr>
</tbody>
</table>

Table 3-3. Improvements obtained the tabu search algorithm.
Table 3-4. Behavior of the local improvement algorithm.

<table>
<thead>
<tr>
<th>Model</th>
<th>Local Improvement Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of A-B swaps</td>
</tr>
<tr>
<td>FAM without maintenance</td>
<td>4</td>
</tr>
<tr>
<td>FAM with maintenance</td>
<td>40</td>
</tr>
<tr>
<td>ctFAM without maintenance</td>
<td>34</td>
</tr>
<tr>
<td>ctFAM with maintenance</td>
<td>39</td>
</tr>
</tbody>
</table>

Table 3-5. Behavior of the tabu search algorithm.

<table>
<thead>
<tr>
<th>Model</th>
<th>Tabu Search Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of A-B swaps</td>
</tr>
<tr>
<td>FAM without maintenance</td>
<td>4163</td>
</tr>
<tr>
<td>FAM with maintenance</td>
<td>1963</td>
</tr>
<tr>
<td>ctFAM without maintenance</td>
<td>15354</td>
</tr>
<tr>
<td>ctFAM with maintenance</td>
<td>4998</td>
</tr>
</tbody>
</table>

3.6 Multi-criteria Optimization for ctFAM

So far we have discussed the ctFAM with the objective of maximizing through and fleet contribution. However, the fleet and through assignment of the schedule also affects the
subsequent decisions such as crew scheduling and manpower usage. These decisions are quite complex problems themselves and an integrated model, which can handle the constraints involved of all these decision and their objectives, is not solvable with the current technology.

We noticed in our computational results for ctFAM that an optimal or near optimal solution for the fleet assignment model may not be a good input for the through assignment model. Similarly, an optimal or near optimal solution for the ctFAM may not be a good input for the subsequent stages of the scheduling planning process. However, there are often many solutions that are nearly optimal and can be good for a subsequent stage. Suppose that we are given a measure of the “goodness”, $G(x, y)$, of a ctFAM solution $(x, y)$ for a subsequent stage such as manpower planning. The goodness measure specifies the likelihood that the solution is a good input for the subsequent stage. Given such a measure, a search procedure can evaluate several near optimal solutions and obtain a set of pareto optimal solutions, where each solution in the set is better than any other solutions either in terms of contribution or the goodness measure. Such a set shows the tradeoff between the fleet and through contribution and the goodness of solutions. The planners can use a subset of these solutions, which are “satisfactory” on both criteria, as inputs for the subsequent stage and choose one depending on the overall attractiveness of the solution. Typically, obtaining the pareto optimal set of solutions is a very hard problem. Therefore, we can try to find instead a set of solutions such that each solution is pareto-optimal with respect to the rest of the solutions in the set.

Since our neighborhood search algorithms examine a large number of solutions that are nearly optimal, they could be used to find a good candidate set of solutions for the planners. One straightforward approach is to evaluate all the solutions encountered by the neighborhood search algorithm in each iteration and select a set of pareto-optimal solutions from it. The difficulty with this approach is that the solutions considered by neighborhood search have no relation with the goodness criterion; hence the quality of pareto-optimal set obtained by this method is likely to be poor. If the goodness measure has a form that can be optimized using the neighborhood search algorithm, then we can use other approaches which may be more promising. We were provided with a goodness measure $G(x, y)$ for the manpower planning stage by the airline we worked with. This measure is a sum of the expected manpower power usage at each city given the fleet assignment of arrivals and departures at the city. The dependence of this function on the fleet assignment was such that it we could use our neighborhood structure to optimize the ctFAM solution for the function $G(x, y)$.
In this section, we describe extensions of our neighborhood search algorithm to solve the multi-criteria ctfAM, where we use G(x, y) as the additional objective function. For notational convenience, we use TF(x, y) to represent the objective function (2.1a) of the ctfAM. We investigated two possible approaches of extending the neighborhood search algorithms to solve the multi-criteria ctfAM. We describe them next.

**Minimizing the Weighted Sum**

This is the most commonly used technique in multi-criteria optimization. In this approach, a weighted sum of the individual criterion is optimized as a single objective function. We modified our neighborhood search algorithms for the ctfAM to maximize the weighted sum: \( \lambda \text{TF}(x, y) + (1-\lambda)\text{G}(x, y) \) for some \( 0 \leq \lambda \leq 1 \) (we set it up as a minimization problem over \( -\lambda \text{TF}(x, y) - (1-\lambda)\text{G}(x, y) \)). This involved defining the arc costs in the A-B improvement graph such that the cost of a valid cycle is equal to the change in the objective \( \lambda \text{TF}(x, y) + (1-\lambda)\text{G}(x, y) \). We run the neighborhood search algorithms on this objective for different values of \( \lambda \), starting from \( \lambda = 0 \) and then incrementing it until \( \lambda = 1 \). Note that high values of \( \lambda \) put more weight on through and fleet assignment, where as small values of \( \lambda \) put more weight on the manpower goodness function. Hence, by varying \( \lambda \) we can direct the neighborhood search into looking for solutions with high contribution or high goodness value.

**Sequential Minimization**

In this approach the objective functions are assumed to have an order of their relative importance. The optimization problem is first solved with the most important objective. After that, the second objective is minimized over the set of solutions that have the optimal value for the first objective and so on. Since the number of solutions that have the optimal value on the first objective may be just one, this technique may lead to just a single solution. We use a slight relaxation of this approach and it is only applicable to the tabu search algorithm for the ctfAM. We first use our neighborhood search algorithms with the objective function \( \text{TF}(x, y) \) to maximize the through and fleet contribution. Once a local optimal solution is obtained, we restrict the search space of the tabu search algorithm so that it only generates neighbors that are within \( \alpha \) units of the through and fleet contribution for the local optimal solution. The number \( \alpha \) represents the amount that the planner is willing to for go in fleet and through contributions to get a solution with better value of goodness.

We next describe our computational results for the neighborhood search algorithms based on these two approaches.
3.6.1 Computational Testing

We implemented two local improvement algorithms based on the first approach. In the first algorithm, which we denote by Local1, we use the same starting solution for all values of the weight λ. In the second algorithm, Local2, we use the local optimum obtained by the last value of λ as the starting solution for the next value. Similarly, we tested two tabu search algorithms, Tabu1 and Tabu2, based on the first approach. In each of these algorithms, we start with λ = 0, and increment its value by 0.1 after each execution of the neighborhood search algorithms.

In Figure 3-8, we compare the set of solutions obtained Local1 and Local2. We have plotted the solutions obtained by the four algorithms with the contribution along vertical axis and the manpower index along horizontal axis. Since we solved these problems as minimization problems, the set of points closer to the origin represent good solutions. We note that the set of solutions obtained by Local2 performs slightly better than Local1. Figure 3-9 shows the results for Tabu1 and Tabu2. In this case, the set of solutions obtained by Tabu2 is substantially better than Tabu1. This suggests that starting from the best solution of the previous run is a better strategy than starting from same solution always.

We also implemented the sequential optimization approach in a tabu search algorithm. We found that it always produced a small number of pareto-optimal solutions compared to the other techniques but the quality of these solutions is better if we choose a small value of α. We provide comparison of one such run with the Tabu2 and Local2 algorithms in Figure 3-10.

Figure 3-8. Solutions obtained from Local1 and Local2.
Figure 3-9. Solutions obtained from Tabu1 and Tabu2.

Figure 3-10. Comparison of solutions from Tabu2, Local2, and Sequential.

3.7 Summary and Conclusions

In this chapter, we study the combined through and fleet assignment model (ctFAM) which integrates the fleet assignment model (FAM) and the through assignment model (TAM). We give an integer programming formulation of ctFAM which, unfortunately, is too large to be
solved to optimality or near-optimality using the state-of-art commercial IP solvers. One of the major contributions of this chapter is the development of a swap-based neighborhood search algorithm for cTFAM that proceeds by swapping the fleet types of flights flown by two fleet types. Our swap based neighborhood structure extends the previous neighborhoods suggested by Berge and Hopperstad [1993] and Talluri [1996] to incorporate through benefits. We propose the idea of an A-B improvement graph to search our swap based neighborhood structure. We consider an alternate definition of the improvement graph where we use approximate banks to define arc costs, however, the resulting neighborhood structure is too restrictive and does not provide good results. We searched our neighborhood heuristically using an integer programming solver. We implemented two versions of our basic algorithm – a local improvement algorithm and a tabu search algorithm. We also examine methods of extending neighborhood search algorithms to provide a good set of solutions to the multi-criteria version of the cTFAM.

Computational results of our algorithms show that a neighborhood search based approach can be used as an effective supplement to the current integer programming techniques to solve airline scheduling problems. The results also highlight the inherent dependency between the different models in airline scheduling. We could obtain large improvements in the through contribution by slightly worsening the fleeting contribution of a nearly optimal fleet assignment solution. In conclusion, through the work in this thesis we have established neighborhood search as an important tool for the airlines to integrate and build upon the current model and solve advanced planning problems.
Chapter 4

Extended Neighborhoods

4.1 Introduction

Neighborhood search has become a popular method to solve difficult optimization problems (Aarts and Lenstra [1997]). This is primarily because of its intuitive appeal and empirical success in solving problems. An important part of the design of any neighborhood search algorithm is the definition of the neighborhood structure N. The size of the neighborhood of a solution \( x \), defined as the number of solutions in \( N(x) \), is of particular concern. As noted in the survey paper of Ahuja, Ergun, Orlin, and Punnen [2001], typically the larger the size of the neighborhood of each solution the better is the quality of the local optimum obtained by the neighborhood search algorithm. However, searching a larger neighborhood structure usually requires longer time. Hence a larger neighborhood structure does not always lead to a better algorithm. In recent years, there has been lot of interest in designing very large-scale neighborhood (VLSN) structures for combinatorial optimization problems. Such neighborhood structures have a large number of neighbors for each solution, often exponential in the input size of the problem. The neighborhood search algorithms that are used to search such very large-scale neighborhood structures usually rely on some implicit enumeration approach because explicitly calculating the objective value of each neighbor of the current solution may not be practical. The recent survey papers of Ahuja et al. [2001] and Deineko and Woeginger [1997] describe several VLSN structures and methods of performing implicit searches on these neighborhood structures.

The rule of thumb that larger size neighborhoods lead to a better set of local optima is clearly violated in circumstances in which two neighborhood structures differ dramatically in size but have the same set of local optima regardless of the problem instance. One example of such a situation is the 2-opt (Flood [1956] and Croes [1958]) and the independent 2-opt (Potts and Velde [1995]) neighborhood structures for the traveling salesman problem (TSP). It is well-known that
for a traveling salesman problem on a graph with \( n \) vertices, the size of the 2-opt neighborhood of any solution is \( O(n^2) \). Potts and Velde [1995] showed that the size of the independent 2-opt neighborhood structure is \( \Omega(1.75^n) \). They also showed that the set of local optima for the independent 2-opt neighborhood structure is the same as that of the 2-opt neighborhood structure for all instances of the traveling salesman problem.

These two neighborhoods are equivalent in the sense that the problem of finding a local optimal solution for the 2-opt neighborhood is the same as the problem of finding a local optimal solution for the independent 2-opt neighborhood. From the perspective of very large scale neighborhood search, the equivalence merits further analysis. Although the 2-opt neighborhood is small, in some important technical sense it is equivalent to a very large scale neighborhood.

More generally, we say that two neighborhood structures are "LO-equivalent" if they have the same set of local optima, regardless of the instance of the problem. The extended neighborhood of a neighborhood \( N \) is the largest neighborhood \( N^* \) that is LO-equivalent to \( N \). We establish in Section 4.2 that such a neighborhood exists.

In this chapter, we introduce the concept of \( \text{extended neighborhood} \) of a neighborhood structure and develop some of its basic theory. The chapter is organized as follows. In Section 4.2, we provide our notation, some additional definitions, and some elementary results concerning extended neighborhoods. In Section 4.3, we provide the characterization of the extended neighborhood in the case of combinatorial optimization problems with general linear costs and prove some additional properties of extended neighborhoods for this case. In Section 4.4, we present results relating to the size and other properties of the extended neighborhood of the 2-opt neighborhood for the TSP, which we denote as the 2-opt*-neighborhood. We summarize contributions and provide future research directions in Section 4.5.

4.2 Extended Neighborhood

We define an instance of a combinatorial optimization problem by a pair \( I = (S,f) \), where \( S \) denotes a finite set of feasible solutions and \( f: S \rightarrow \mathbb{R} \) is a real-valued objective function selected from some class \( \mathcal{F} \) of objective functions. We shall assume the combinatorial optimization problems to be minimization problems. A neighborhood structure is a function \( N: S \rightarrow 2^S \) that maps each feasible solution in \( S \) to a set of feasible solutions, which are called its neighbors. We say that a solution \( x \in S \) is \( \text{locally optimal} \) with respect to \( N \) if \( f(x) \leq f(x') \) for all \( x' \in N(x) \). We denote the set of locally optimal solutions with respect to a neighborhood structure \( N \).
in the instance \( I \) by \( LO^N_I \subseteq S \). We say that two neighborhood structures \( N^1 \) and \( N^2 \) are \( LO \)-equivalent if and only if \( LO^{N^1}_I = LO^{N^2}_I \) for all instances \( I \) of the combinatorial optimization problem, i.e., \( N^1 \) and \( N^2 \) have the same set of local optima for all instances of the combinatorial optimization problem. The following property follows directly from the definitions above.

**Proposition 4.1.** If two neighborhood structures \( N^1 \) and \( N^2 \) are \( LO \)-equivalent then the neighborhood structure defined as \( N(x) = N^1(x) \cup \bar{N}^2(x) \) for \( x \in S \) is \( LO \)-equivalent to both \( N^1 \) and \( N^2 \).

We denote the set of neighborhood structures \( LO \)-equivalent to a given neighborhood structure \( N \) by \( LO^N \). We define the extended neighborhood of \( N \) as the neighborhood structure \( N^*(x) = \bigcup_{N \in LO^N} N'(x) \), for all \( x \in S \). Note that since the set \( S \) is finite, the number of possible neighborhood structures is finite. In particular, the set \( LO^N \) is finite as well. Hence, the extended neighborhood always exists and is uniquely defined. Proposition 1 implies that the neighborhood structure \( N^* \) is \( LO \)-equivalent to \( N \). In fact, it has the largest size neighborhood set for each solution among all neighborhood structures \( LO \)-equivalent to \( N \). We note that any two \( LO \)-equivalent neighborhood structures have the same extended neighborhood.

A neighborhood structure is called exact for a combinatorial optimization problem if any locally optimal solution for an instance is also an optimal solution for the instance. The following proposition follows directly from the definition of the extended neighborhood of a neighborhood structure.

**Proposition 4.2.** A neighborhood structure \( N \) is exact for a combinatorial optimization problem if and only if for any instance \((S, f)\), \( N^* \) satisfies the condition: \( N^*(x) = S \) for \( x \in S \).

Glover and Punnen [1997] introduced a concept to measure the effectiveness of heuristics, which is somewhat related to extended neighborhoods. They defined the domination ratio of a heuristic algorithm \( \alpha \) for a combinatorial problem as:

\[
\text{dom}(\alpha) = \inf_{(S, f)} \frac{\left| \{ x \in S : f(x) \geq f(x_{\alpha}) \} \right|}{|S|}, \text{ where } x_{\alpha} \text{ is the solution obtained by the heuristic } \alpha \text{ on instance } (S, f).
\]

The domination ratio represents the fraction of solutions that are guaranteed to be worse than the solution obtained by the heuristic \( \alpha \) for any instance of the problem. The concept applies to any heuristic including neighborhood search techniques. Glover and Punnen offer the concept
in part as an alternative to the size of the neighborhood, and in part as an alternative metric for the quality of the solution. The concept of domination number sometimes relates to the size of the extended neighborhood, but in general the domination number is an independent concept. For example, if we generate a random solution from the set S, the domination number of our heuristic is half on average, even though there is no neighborhood structure involved. Further, two neighborhood structures for a combinatorial optimization problem could have the same domination ratio even if they do not have different set of local optima.

We have provided the definition of extended neighborhoods for a generic combinatorial optimization problem. In the rest of this chapter we consider combinatorial optimization problems where the set of feasible solutions S is contained in $\mathbb{R}^n$ (often $S \subseteq \{0, 1\}^n$) and the objective function $f$ is an arbitrary linear function, i.e. $f(x) = cx$, for some vector $c$ in $\mathbb{R}^n$. We note that in some cases the set of cost vectors allowed for a problem might be required to be non-negative or satisfy additional constraints such as triangle inequality in the case of traveling salesman problem. We refer to the problems where any linear cost function is allowed as combinatorial optimization problems with *general linear objectives*. We provide a polyhedral characterization of the extended neighborhood for such problems. Using this characterization, we prove some additional properties of extended neighborhoods for the neighborhood structures for combinatorial optimization problems with general linear costs.

Before concluding this section, we briefly introduce the concept of neighborhood graph of a neighborhood structure here. This concept is commonly used to analyze some properties of the associated neighborhood structure (Aarts and Lenstra [1997]). We shall analyze some properties of the neighborhood graph of 2-opt* in Section 4.4.

Given a neighborhood structure $N$ and an instance $I = (S, f)$ of a combinatorial optimization problem, we define the *neighborhood graph* as a directed graph $G^N(I) = (S, A)$. The set of nodes in the neighborhood graph is the same as the set of feasible solutions in the instance. We create an arc $(x, y)$ in the graph if $y \in N(x)$. We define the *distance* from a solution $x$ to another solution $x'$ as the length of the shortest path from $x$ to $x'$ in $G^N(I)$. If the graph contains no path from $x$ to $x'$, then the distance is $\infty$. We define the *diameter* of the neighborhood graph as the maximum distance between any pair of solutions in the neighborhood graph. We say that a directed path $x_1 \rightarrow x_2 \rightarrow \ldots \rightarrow x_i$ in the neighborhood graph is *monotone* if $f(x_1) > f(x_2) > \ldots > f(x_i)$. We say that a solution $x'$ is *reachable* from solution $x$ if there is a monotone path from $x$ to $x'$ in $G^N(I)$.

### 4.3 Combinatorial Optimization Problems with Linear Objectives

99
In this section, we consider combinatorial optimization problems with linear objectives: minimize \( \{cx : x \in S \subseteq \mathbb{R}^n\} \), where \( c \) is a vector in \( \mathbb{R}^n \). We represent an instance as the pair \((S, c)\) to emphasize the linearity of the objective function. Many interesting combinatorial optimization problems can be formulated as problems with linear objectives. For example, any 0-1 combinatorial optimization problem has linear objectives.

Let \( P \) denote a combinatorial optimization problem with linear objectives. Let \( N \) be a neighborhood structure for \( P \). For a given instance \((S, c)\) and a solution \( x \in S \), let \( N(x) = \{x^1, \ldots, x^K\} \). We define the neighborhood vectors of \( x \) as \( N^V(x) = \{v^i \in \mathbb{R}^n: v^i = x^i - x \text{ for } i = 1, \ldots, K\} \). We define the polyhedral extension of \( N \) for the solution \( x \), denoted by \( N^E(x) \), as:

\[
N^E(x) = \{x' \in S: x' = x + \sum_{i=1}^K \lambda_i v^i, \lambda_i \geq 0 \text{ for } i = 1, \ldots, K\}.
\]

In other words, \( N^E(x) \) is the set of solutions that can be obtained by adding a non-negative linear combination of neighborhood vectors in \( V(x) \) to the solution \( x \). Clearly, the mapping \( N^E: S \to 2^S \) defines a neighborhood structure for \( P \) and \( N(x) \subseteq N^E(x) \) for any solution \( x \in S \). In this section, we study some properties of the polyhedral extension for any neighborhood structure defined for combinatorial optimization problems with linear objectives. We provide a sufficient condition, under which, the polyhedral extension is the same as the extended neighborhood of the neighborhood structure.

**Lemma 4.1.** The neighborhood structure \( N^E \) is LO-equivalent to \( N \).

**Proof.** Consider any arbitrary instance \((S, c)\) of the combinatorial optimization problem with linear costs. We noted that for any solution \( x \in S \), \( N(x) \subseteq N^E(x) \). Therefore, if a solution is locally optimal with respect to \( N^E \), it must be locally optimal with respect to \( N \). We now prove the converse. Let \( x^* \) be a locally optimal with respect to \( N \). Let \( N^V(x^*) = \{v^{*1}, v^{*2}, \ldots, v^{*K}\} \). By the definition of local optimality, we know that \( c v^i \geq 0 \) for \( i = 1, \ldots, K \). Consider any \( x' \in N^E(x^*) \), i.e.,

\[
x' = x + \sum_{i=1}^K \lambda_i v^i,
\]

for some \( \lambda_i \geq 0 \) for \( i = 1, \ldots, K \). In this case, \( c x' = c x^* + \sum_{i=1}^K \lambda_i v^i \geq c x^* \).

Therefore, \( x^* \) is locally optimal with respect to \( N^E(x^*) \). Since we picked \((S, c)\) arbitrarily, for any instance of the combinatorial optimization problem, a solution is locally optimal with respect to \( N \) if and only if it is locally optimal with respect to \( N^E \).

Since the extended neighborhood \( N^* \) is the largest neighborhood structure for each solution such that it is LO-equivalent to \( N \), \( N^E(x) \subseteq N^*(x) \) for each solution \( x \in S \). Therefore, the size of the polyhedral extension of \( N \) is a lower bound on the size of the extended neighborhood of \( N \). A natural question is the following: under what conditions will the polyhedral extension of
N also be the extended neighborhood of N? We offer a partial answer to this question by providing a sufficient condition for \( N^E \) to be the extended neighborhood.

We say that a combinatorial optimization problem P has \textit{general linear objectives} if it satisfies the following condition: If \((S, c)\) is an instance of P for some \( c \in \mathbb{R}^n \) then for any \( c' \in \mathbb{R}^n \) the pair \((S, c')\) is an instance of P.

\textbf{Theorem 4.1.} Let P be a combinatorial optimization problem with general linear objectives and N be a neighborhood structure for P. Then \( N^*(x) = N^E(x) \) for \( x \in S \).

\textbf{Proof.} Suppose that there exists an instance \((S, c)\) of P and a solution \( x' \in S \) such that \( N^E(x') \neq N^*(x') \). Let \( x^* \) be a solution such that \( x^* \in N^*(x') \setminus N^E(x') \). Since \( x^* \notin N^E(x') \), the linear system \( \{ \sum_{i=1}^{K} \lambda_i y^i = x^* - x', \lambda_i \geq 0 \text{ for } i = 1, \ldots, K \} \) must be infeasible. Farkas Lemma (see for example Papadimitriou and Steiglitz [1982]) implies the existence of a vector \( w \in \mathbb{R}^n \) such that \( w(x^* - x') < 0 \) and \( w y^i \geq 0 \) for \( i = 1, \ldots, K \). Consequently, in the instance \((S, w), x'\) is a locally optimal with respect to N but not locally optimal with respect to \( N^* \), which is a contradiction since N and \( N^* \) are LO-equivalent. Hence, \( N^*(x) = N^E(x) \) for all \( x \in S \). ✷

The previous two results provide a polyhedral characterization of the extended neighborhood when the combinatorial optimization problem allows general linear costs as objective functions. We now illustrate this result using two examples of neighborhood structures in the literature.

\textbf{Example 1:} A linear program: \( \min \{ cx : x \in Q \subseteq \mathbb{R}^n \} \) over a polytope \( Q \) induces a combinatorial optimization problem where the set of feasible solutions S is the set of extreme points of the polytope. The simplex method for the linear programming can be viewed as a neighborhood search algorithm where the neighbors of an extreme point x are those extreme points that share an edge with x in the polytope \( Q \). It can be shown that for this neighborhood structure, \( Q \subseteq N^E(x) \) for all \( x \in S \). Theorem 4.1 implies that the extended neighborhood for the neighborhood structure of the simplex method is equal to S for all solutions \( x \in S \). Proposition 4.1 shows that the neighborhood structure of the simplex method is exact for linear programming problems, which is well known. In this case, the concept of extended neighborhood provides an alternate proof of exactness of this neighborhood structure.

\textbf{Example 2:} The traveling salesman problem is to find a minimum weight Hamiltonian cycle in a graph \( G = (V, E) \). It can be formulated as a combinatorial optimization problem with general linear costs as follows. Each Hamiltonian cycle can be represented as a 0/1 vector \( x \in \{0, 1\}^{|E|} \).
where \( v_e = 1 \) means that the edge \( e \) is in the cycle. The set of feasible solutions of an instance is given by \( S = \{ x \in \{0, 1\}^{|E|} : x \) is the incidence vector of a Hamiltonian cycle in \( G \} \). A 2-opt move on a Hamiltonian cycle removes two edges from it and adds two edges to it in order to get a new Hamiltonian cycle. A 2-opt move can be represented as a vector in \( \{0, 1, -1\}^{|E|} \), where \( v_e = 1 \) means that the edge \( e \) is added to the cycle, \( v_e = -1 \) means that the edge is removed from the cycle, and \( v_e = 0 \) otherwise. The 2-opt neighborhood of a Hamiltonian cycle consists of all the Hamiltonian cycles that can be obtained from it by a 2-opt move. From Theorem 4.1, the extended neighborhood of the 2-opt neighborhood structure contains all the Hamiltonian cycles that can be obtained by non-negative combinations of 2-opt moves. In particular, the Hamiltonian cycles obtained by 0/1 combinations of 2-opt moves are part of the extended neighborhood. We note that the independent 2-opt neighborhood of Potts and Velde [1995] is a subset of such Hamiltonian cycles.

We next observe that if \( P \) is a combinatorial optimization problem with general linear costs and \( N \) is a neighborhood structure for \( P \) that can be searched in polynomial time, then for any instance \( (S, c) \) of \( P \) and solutions \( x, x' \in S \), one can determine in polynomial time whether \( x' \in N^E(x) (= N^e(x)) \).

**Theorem 4.2.** If the neighborhood structure \( N \) can be searched to find a better solution in polynomial time then there is a polynomial time algorithm to determine the membership \( x' \in N^E(x) \) for any \( x' \in S \).

**Proof.** Given that \( x' \in S \), it follows that \( x' \in N^E(x) \) if and only if the linear system \( P^1 = \{ \lambda \in \mathbb{R}^K : \sum_{i=1}^K \lambda_i v^i = x' - x, \lambda_i \geq 0 \text{ for } i = 1, \ldots, K \} \) is feasible, where \( N^V(x) = \{ v^1, v^2, \ldots, v^K \} \). Using Farkas Lemma, we know that the linear system \( P^1 \) is feasible if and only if the linear system \( P^3 = \{ w \in \mathbb{R}^n : (x' - x)w < 0 \text{ and } v^i w \geq 0 \text{ for } i = 1, \ldots, K \} \) is infeasible. The system \( P^3 \) is feasible if and only if \( P^3 = \{ w \in \mathbb{R}^n : (x' - x)w \leq -1 \text{ and } v^i w \geq 0 \text{ for } i = 1, \ldots, K \} \) is feasible. We now show that if the neighborhood structure \( N \) is searchable in polynomial time then the separation problem for the linear system \( P^3 \) can be solved in polynomial time, i.e., given a vector \( w \in \mathbb{R}^n \) we can determine in polynomial time whether \( w \in P^1 \), and if \( w \in P^3 \) we can find a vector \( v \in \mathbb{R}^n \) and scalar \( b \in \mathbb{R} \) such that \( v w < b \) and \( vw \geq b \) for all \( w \in P^3 \). It is well known that if the separation problem for a linear system can be solved in polynomial time, then we can determine the feasibility of the linear system in polynomial time. This will complete the proof.
In order to solve the separation problem, we check in polynomial time whether \( \tilde{w} \) satisfies all the inequalities of system \( P^3 \). First of all, if \((x' - x) \tilde{w} > -1\) then \( \tilde{w} \notin P^3 \) and \( v = x - x' \) and \( b = 1 \) are the outputs. In order to check the constraints \( v_i w_i \geq 0 \) for \( i = 1, \ldots, K \) we search the neighborhood \( N(x) = \{x + v^1, x + v^2, \ldots, x + v^K\} \) using \( \tilde{w} \) as the objective cost vector. If \((x + v^i) \tilde{w} < x \tilde{w}\) for some \( i = 1, \ldots, K \) then \( \tilde{w} \notin P^3 \) because \( v_i \tilde{w} < 0 \) and \( v = \sqrt{v}, b = 0 \) is the output. If all the constraints are satisfied then \( \tilde{w} \) is a feasible solution for the linear system \( P^3 \). Since all the constraints can be checked in polynomial time, the lemma follows.

In the two examples provided in this section, the extended neighborhoods are much larger than the neighborhood structures themselves. However, for certain combinatorial optimization problems with general linear objective, it can be shown that the extended neighborhood is always equal to the neighborhood structure itself.

Let \( Q = \text{Conv}(S) \) denote the convex hull of the feasible solution set of a combinatorial optimization problem \( P \) with general linear costs. We show the following result for the case when \( S \subseteq \{0, 1\}^n \); however, it holds for any situation where every feasible solution in \( S \) is an extreme point of \( Q \).

**Theorem 4.3.** Suppose \( P \) is a combinatorial optimization problem with general linear costs such that for any instance \((S, c) \) of \( P \), \( S \subseteq \{0, 1\}^n \) and the diameter of the polytope \( Q = \text{conv}(S) \) is one. Then for any neighborhood structure \( N \) for the combinatorial optimization problem \( P \), \( N^*(x) = N(x) \) for all \( x \in S \).

**Proof.** Consider the possibility that for some neighborhood structure \( N \) and a feasible solution \( x \), \( N(x) \neq N^*(x) \), i.e., the extended neighborhood is larger than the neighborhood structure itself. Let \( x' \) be a solution in \( N^*(x) \setminus N(x) \). Since \( Q \) is a polytope of diameter one and \( x, x' \) are extreme points in it, they must share a common edge in the polytope. It is well known (see for example Papadimitriou and Steiglitz [1982]) that in this case there must exist a cost vector \( w \) satisfying: \( wx' < wx \) and \( wx \leq wx'' \) for all \( x'' \in S \setminus \{x, x'\} \). However, in the problem instance \((S, w)\), the solution \( x \) is locally optimal with respect to \( N \) but not with respect to \( N^* \), which is a contradiction. Hence it must be the case that \( N^*(x) = N(x) \) for all \( x \in S \).

One of the applications of Theorem 4.3 is the Graph Bipartition problem. In this problem, we are given a complete undirected graph \( G = (V, E) \) containing \( 2n \) nodes and weights \( w_{ij} \) associated with each edge \((i, j) \in E \). The Graph Bipartition problem is to partition the set of nodes \( V \) into two subsets \( V^1 \) and \( V^2 \) such that \( |V^1| = |V^2| \) and the weight of edges crossing the partition,
\[ \sum_{(i, j) \in V^1 \text{ and } j \in V^2} w_{ij}, \] is minimum. This problem can be formulated as a combinatorial optimization problem with general linear cost function as follows. We represent each node partition \((V^1, V^2)\) as a vector \(x \in \{0, 1\}^{|E|}\) such that \(x_{ij} = 1\) if \(i \in V^1\) and \(j \in V^2\) and \(x_{ij} = 0\) otherwise. In other words, \(S = \{x \in \{0, 1\}^{|E|}: x\) represents a node partition\}. The linear objective function associated with each solution is \(f(x) = wx\). In this formulation, the diameter of the polytope \(\text{Conv}(S)\) of the feasible solutions is 1 as shown below in Proposition 4.3. Therefore, the extended neighborhood of any neighborhood structure for the Graph Bipartition problem is the same as the neighborhood structure for all instances of the problem and for any neighborhood structure.

**Proposition 4.3.** The diameter of the polytope of feasible solutions for any instance of Graph Bipartition is 1.

**Proof.** Let \((S, w)\) be an instance of the Graph Bipartition problem and \(x\) be a solution representing the node partition \((V^1, V^2)\). Consider any other solution \(x' \in S\). We shall construct a weight function \(w\) satisfying: \(wx' = wx\) and \(wx < wx''\) for all \(x'' \in S \setminus \{x, x'\}\). The existence of such a weight function implies that \(x\) and \(x'\) share an edge in the polytope \(\text{Conv}(S)\), proving the proposition.

Suppose the solution \(x'\) represents the partition \((\tilde{V}^1, \tilde{V}^2)\). Let \(A = V^1 \cap \tilde{V}^1, B = V^2 \cap \tilde{V}^2, C = \tilde{V}^1 \cap \tilde{V}^2,\) and \(D = \tilde{V}^2 \cap \tilde{V}^1\). Therefore, \(V^1 = A \cup C, V^2 = B \cup D, \tilde{V}^1 = A \cup D,\) and \(\tilde{V}^2 = B \cup C,\) that is, \((\tilde{V}^1, \tilde{V}^2)\) is obtained from \((V^1, V^2)\) by moving the nodes in \(C\) from \(V^1\) to \(V^2\) and nodes in \(D\) from \(V^2\) to \(V^1\). We note that the following must hold: \(|A| = |B|\) and \(|C| = |D|\). We assign the weights in the graph as follows:

1. \(w_{ij} = \infty\) if \(i, j \in A\) or \(i, j \in B\) or \(i, j \in C\) or \(i, j \in D\).
2. \(w_{ij} = 1\) if \(i \in A \cup B\) and \(j \in C \cup D\).
3. \(w_{ij} = 0\) for all other edges.

If \(|A| \neq |C|\), there are only two possible partitions with finite weights: \((V^1, V^2)\) (represented by \(x\)) and \((\tilde{V}^1, \tilde{V}^2)\) (represented by \(x'\)). Let \(|A| = |B| = k_1\) and \(|C| = |D| = k_2\). It is easy to see that \(wx = |A||D| + |B||C| = 2k_1k_2\) and \(wx' = |A||C| + |B||D| = 2k_1k_2\). In the case \(|A| = |C|\) (\(= |B|\) = \(|D|\)), the partition: \((A \cup B, C \cup D)\) has finite weight in addition to the other two partitions. Let \(x''\) represent the solution corresponding to the partition \((A \cup B, C \cup D)\). It can be easily verified that \(wx'' = |A|(|C| + |D|) + |B|(|C| + |D|) = 4k_1k_2\). Hence in both cases \(w\) satisfies the property that \(wx' = wx\) and \(wx < wx''\) for all \(x'' \in S \setminus \{x, x'\}\).
We next study the extended neighborhood for the well-known 2-opt neighborhood structure for the traveling salesman problem.

4.4 Extended Neighborhood for TSP 2-opt

In this section, we analyze some properties of the extended neighborhood 2-opt* of the 2-opt neighborhood structure for the TSP. In particular, we provide a lower bound on the number of solutions in the extended neighborhood. We also compare the neighborhood graphs for 2-opt and 2-opt* and analyze their properties such as reachability, diameter, and the complexity of searching the best solution in the neighborhood structure.

Let $G = (V, E)$ be a weighted complete undirected graph with $V = \{1, \ldots, n\}$ and weights $c_{ij}$ associated with each edge $(i, j) \in E$. The traveling salesman problem (TSP) is to find the minimum weight Hamiltonian cycle (tour) in $G$. We represent a tour as a sequence $i_1 - i_2 - \ldots - i_n - i_1$, where edges $(i_k, i_{k+1})$ for $k = 1, \ldots, n-1$ and $(i_n, i_1)$ belong to the tour. We assume without loss of generality that $i_1 = 1$. Let $T$ be a tour. The *incidence vector* for $T$ is the vector $x \in \{0, 1\}^{|E|}$ where $x_{ij} = 1$ for edges $(i, j) \in T$ and $x_{ij} = 0$ otherwise. The traveling salesman problem can be formulated as a combinatorial optimization problem with linear objective as follows. We define the set of feasible solutions as $S = \{x \in \{0, 1\}^{|E|}, x$ is incidence vector of some tour in $G\}$ and the linear objective function associated with each feasible solution $x$ is $f(x) = cx$.

Let $T = i_1 - i_2 - \ldots - i_n - i_1$ be a given tour and $x$ be its incidence vector. We represent a 2-opt move on the tour using an unordered pair $\{k, l\}$, where $k, l \in \{1, \ldots, n\}$ such that $i_k$ and $i_l$ are not adjacent in the tour, i.e., $k \neq l-1$ and $k \neq l+1$. The move $\{k, l\}$ represents the following changes to tour $T$:

1. Remove the edges $(i_k, i_{k+1})$ and $(i_l, i_{l+1})$
2. Add the edges $(i_k, i_l)$ and $(i_{k+1}, i_{l+1})$.

In this definition we assume that $i_{n+1}$ represents $i_1$. We denote the set of 2-opt moves corresponding to tour $T$ by $\text{2-opt}^\text{move}(T)$. The 2-opt neighborhood for the tour $T$ (solution $x$) is defined as all the tours (incidence vectors of tours) that can be obtained from $T$ by performing a 2-opt move $\{k, l\}$ for some $\{k, l\} \in \text{2-opt}^\text{move}(T)$. We use $\nu^x = \{0, 1\}^{|E|}$ to denote the neighborhood vector in 2-opt$^x$ corresponding to the 2-opt neighbor obtained by performing the move $\{k, l\}$, i.e.,

105
\[
\nu^k(e) = \begin{cases} 
-1 & \text{if } e = (i_k, i_{k+1}) \text{ or } e = (i_l, i_{l+1}) \\
1 & \text{if } e = (i_k, i_l) \text{ or } e = (i_{k+1}, i_{l+1}) \\
0 & \text{otherwise}
\end{cases}
\]

We note that adding the vector \( \nu^k \) to \( x \) gives the incidence vector of the tour obtained by performing the move \( \{k, l\} \) on \( T \). The polyhedral extension of the 2-opt neighborhood of solution \( x \) is given by 2-optE(\( x \)) = \{ \( x' \in S \colon x' = x + \sum_{\{k, l\} \in 2\text{-optmove}(T)} \lambda_{kl} \nu^k \), \( \lambda_{kl} \geq 0 \) for \( \{k, l\} \in 2\text{-optmove}(T) \} \). Theorem 4.1 implies that for TSP with general linear objectives the polyhedral extension 2-optE is equal to the extended neighborhood 2-opt* for the 2-opt neighborhood structure. Often, the TSP is restricted to instances where the edge weights \( c \) satisfy the triangle inequality, that is: for any \( i, j, k \in V \), \( c_{ij} + c_{jk} \geq c_{ik} \). We refer to this special case of the TSP as TSPTI. We first show that for any instance \( I \) of TSP, the instance obtained by adding a constant to each edge weight has the same set of 2-opt local optima as the instance \( I \). Using this property, we show that if TSP is restricted to cost vectors satisfying triangle inequality, the polyhedral extension 2-optE is still the extended neighborhood of 2-opt neighborhood structure.

**Lemma 4.2.** Let \( G = (V, E) \) be an input graph and \( I = (S, c) \) be an instance for TSP on \( G \) with cost vector \( c \). Given an integer \( M \), the instance \( I' = (S, c') \) where \( c'_ij = c_{ij} + M \) for \( (i, j) \in E \) satisfies \( LO^{2\text{-opt}}_I = LO^{2\text{-opt}}_{I'} \).

**Proof.** A solution \( x \) corresponding to a tour \( T = i_1 - i_2 \ldots - i_n - i_1 \) is in \( LO^{2\text{-opt}}_I \) if and only if the following condition is satisfied: for every \( \{k, l\} \in 2\text{-optmove}(T) \), \( c_{ik} + c_{kj} + c_{jl} \leq c_{ij} + c_{ik} + c_{kj} \).

Clearly, if the condition is satisfied for \( c \), it is also satisfied for \( c' \). Hence, the solution \( x \) is in \( LO^{2\text{-opt}}_{I'} \) if and only if it is in \( LO^{2\text{-opt}}_I \).

**Corollary 4.1.** Given an instance \( (S, c) \) of TSP, there exists an instance \( (S, c') \) such that \( c' \) satisfies triangle inequality and \( LO^{2\text{-opt}}_{I'} = LO^{2\text{-opt}}_I \).

**Proof.** Let \( G = (V, E) \) be the graph corresponding to the instance \( (S, c) \) and \( M = \max_{(i, j, k) \in \nu^k} \{ c_{ik} - (c_{ij} + c_{jk}) \} \). We can obtain \( c' \) by adding the constant \( M \) to each edge weight in \( c \).

**Theorem 4.4.** For the TSPTI, the polyhedral extension 2-optE of 2-opt is the same as the extended neighborhood of 2-opt.

**Proof.** Let 2-opt* denote the extended neighborhood for 2-opt neighborhood structure on the restricted problem TSPTI. Suppose \( I = (S, c) \) is an instance of TSPTI with solutions \( x, x' \in S \) such
that 2-opt\textsuperscript{E}(x) ≠ 2-opt\textsuperscript{TT}(x) and x' ∈ 2-opt\textsuperscript{TT}(x)2-opt\textsuperscript{E}(x). Since 2-opt\textsuperscript{E} is the extended neighborhood of 2-opt for the TSP (with general linear objectives), there must exist an instance I' = (S, c') of the TSP such that LO\textsuperscript{2-opt\textsuperscript{E}} \textsubscript{I'} ≠ LO\textsuperscript{2-opt\textsuperscript{TT}} \textsubscript{I'} (otherwise 2-opt\textsuperscript{TT} would be the extended neighborhood of TSP). Using Corollary 4.1, there exists an instance I'' = (S, c'') such that c'' satisfies triangle inequality and LO\textsuperscript{2-opt\textsuperscript{E}} \textsubscript{I''} = LO\textsuperscript{2-opt\textsuperscript{E}} \textsubscript{I'}. Since LO\textsuperscript{2-opt\textsuperscript{E}} \textsubscript{I'} = LO\textsuperscript{2-opt\textsuperscript{TT}} \textsubscript{I'} by Lemma 4.1, our initial assumption implies that LO\textsuperscript{2-opt\textsuperscript{TT}} \textsubscript{I'} ≠ LO\textsuperscript{2-opt\textsuperscript{TT}} \textsubscript{I''}, which is a contradiction.

Let x denote the incidence vector of the tour T. Lemma 4.1 implies that 2-opt\textsuperscript{E}(x) consists of tours obtained by adding a non-negative linear combination of set of neighborhood vectors \textsuperscript{v} \textsuperscript{hl} to x. In particular, the polyhedral extension 2-opt\textsuperscript{E}(x) of 2-opt(x) contains tours obtained by adding 0/1 combinations of neighborhood vectors to the solution x. In other words, if a set of 2-opt moves for the tour T results in creating a new tour T' then the incidence vector of T' belongs to 2-opt\textsuperscript{E}(x). However, Figure 4-1 shows that performing an arbitrary set of 2-opt moves on the current tour does not always create a tour.

![Figure 4-1. (a) a tour, (b) 2-opt moves \{1, 5\}, \{3, 8\} (c) 2-opt moves \{1, 5\}, \{3, 8\}, and \{6, 10\}.](image)

In the example illustrated above, Figure 4-1(a) represents a tour and Figure 4-1(b) represents the set of edges obtained after 2-opt moves \{1, 5\} and \{3, 8\}. This set of edges represents two subtours: 1 → 5 → 4 → 9 → 10 → 1 and 2 → 3 → 8 → 7 → 6 → 2. Hence performing the moves \{1, 5\} and \{3, 8\} does not produce a tour. However, if we perform one additional move: \{6, 10\}, the resulting set of edges is a tour and is shown in Figure 4-1(c). The example also illustrates a result regarding the reachability of local optimum solutions that we discuss next.

### 4.4.1 Reachability in G\textsuperscript{2-opt} and G\textsuperscript{2-opt\textsuperscript{*}}

2-opt and 2-opt\textsuperscript{*} always have the same set of local optima; however, given an initial tour T there may be a locally optimal tour T' reachable from T by performing a sequence of cost
improving iterations with 2-opt* neighborhood but not with 2-opt neighborhood structure. We use the example in Figure 4-1 to construct an instance of the TSP with this property.

Consider the following weights for the edges in the traveling salesman problem illustrated in Figure 4-1. We set the weight of all the edges in the tour 1-...-10-1 in Figure 4-1(a) equal to 0. We set the weights of edges (1, 5), (2, 6) and (3, 8), (4, 9) equal to −1. This ensures that the 2-opt moves {1, 5} and {3, 8} each improve the weight of the tour in Figure 4-1(a) by −2. We set the weights of edges (6, 10) and (7, 1) to 0. This ensures that the move {6, 10} makes no change in the weight of the tour. Finally, for all the remaining edges, we set the weight to ∞. This weight assignment ensures that only 2-opt moves {1, 5} and {3, 8} improve the cost of the tour in Figure 4-1(a), the 2-opt move {6, 10} does not modify the tour cost, and all other 2-opt moves increase the tour cost to ∞. We note that the tour in Figure 4-1(c) is the optimal tour under this weight assignment, and hence it is locally optimal as well. We choose this tour as T*. If we choose the tour in Figure 4-1(a) to be T then T* is in 2-opt*(T) because it can be reached from T by performing the set of 2-opt moves {1, 5}, {3, 8}, and {6, 10}. Therefore, T* can be reached from T in one step in 2-opt* neighborhood. In the case of 2-opt neighborhood structure only one of the two improving moves {1, 5} and {3, 8} can be chosen. The tours resulting from making these moves are shown in Figure 4-2(a) and Figure 4-2(b) respectively.

![Figure 4-2](image)

Figure 4-2. (a) Tour obtained by move {1, 5} (b) Tour obtained by move {3, 8}.

It can be verified that the tours in Figure 3(a) and Figure 3(b) are both locally optimal under the weight assignment above. Hence, the tour T* cannot be reached from the tour T using a sequence of improving 2-opt moves (corresponding to a monotone path in G\textsuperscript{2-opt}). We now state the result as a theorem.

**Theorem 4.5.** There exist instances I = (S, f) of TSP with tours T, T* ∈ S such that T* ∈ LO\textsuperscript{2-opt}_I and T* is reachable from T in G\textsuperscript{2-opt*}(I) but not in G\textsuperscript{2-opt}(I).

Since 2-opt neighbors of a solution are contained in 2-opt*, any local optimum reachable from a solution in G\textsuperscript{2-opt} is also reachable in G\textsuperscript{2-opt*}. The neighborhood search algorithms typically follow monotone paths in the neighborhood graph. Therefore, the set of local optima reachable from a solution in a neighborhood search algorithm based on 2-opt* is at least as large as the one
based on 2-opt neighborhood structure. We now analyze the size of the extended neighborhood of a solution.

4.4.2 Size of 2-opt*

We noted in Section 4.1 that the size of 2-opt* is $\Omega(1.75^n)$ because the independent 2-opt neighborhood structure is LO-equivalent to the 2-opt neighborhood structure. In this section, we show that the size of the 2-opt* neighborhood of a solution is at least $(n-3)(\lceil n/2 \rceil - 3)!$. We shall establish this result by constructing an enumeration tree that enumerates a subset of 2-opt* neighbors of a tour $T$. Our enumeration tree satisfies the following properties:

1. The root of the tree is $T$.
2. Each node of the tree is in the 2-opt* neighborhood of $T$.
3. If $T'$ is a node in the tree then the children of $T'$ are obtained from $T'$ by performing a 2-opt move that is also valid for $T$.
4. There are at least $(n-3)(\lceil n/2 \rceil - 3)!$ nodes in the tree.

We describe the construction of the tree later in the section after introducing some preliminary results. Without loss of generality, we assume that $T$ is given by the sequence $1 - 2 - 3 - \ldots - n - 1$. We define a 2-opt move for $T$ using an unordered pair $\{i, j\}$ of nodes such that $j \not\in \{i-1, i+1\}$. The pair $\{i, j\}$ represents the following change to the tour $T$:

1. Remove edges $(i, i+1)$ and $(j, j+1)$.
2. Add edges $(i, j)$ and $(i+1, j+1)$.

Let $T'$ be a tour such that an edge $(i, i+1) \in T'$ for some $i = 1, \ldots, n$ (where $n+1$ represents 1). Let the tour $T'$ be represented as the sequence $i_1 - i_2 - \ldots - i_n - i_1$. We say that the edge $(i, i+1)$ is a forward edge in the sequence if node $i+1$ appears after node $i$ in the sequence, otherwise $(i, i+1)$ is a backward edge. We say that a 2-opt move $\{i, j\}$ for $T$ is feasible for the tour $T'$ if $T'$ contains edges $(i, i+1)$ and $(j, j+1)$ and either both the edges appear as forward edges or both appear as backward edges in the sequence representing $T'$.

**Lemma 4.3.** If a move $\{i, j\}$ for $T$ is feasible for $T'$ then performing the move on $T'$ generates a new tour.
**Proof.** Let $T'$ be represented by the sequence $i_1 - i_2 - \ldots - i_n - i_1$. Suppose $\{i, j\}$ is a feasible move for $T'$. Then $T'$ contains edges $(i, i+1)$ and $(j, j+1)$. Let $i = i_k$ and $j = i_l$ for some $k, l \in \{1, \ldots, n\}$. We also assume, without loss of generality, that $k > l$. There are two possible cases:

Case 1. $(i, i+1)$ and $(j, j+1)$ are forward edges in the sequence for $T'$. In this case, $T'$ is given by sequence: $i_1 - i_2 - \ldots - i_{k-1} - i - (i+1) - i_{k+2} - i_{k+3} - \ldots - i_{l-1} - i_j - (j+1) - i_{l+2} - i_{l+3} - \ldots - i_n - i_1$, i.e., $i_k = i, i_{k+1} = i+1, i_i = j$, and $i_{k+1} = j+1$. Performing the move $\{i, j\}$ for $T$ on the tour $T'$ generates a new tour: $i_1 - i_2 - \ldots - i_{k-1} - i - j - i_{j+1} - i_{j+2} - \ldots - i_{k+3} - i_{k+2} - i_{k+3} - \ldots - i_n - i_1$, which contains the sequence $(i+1) - i_{k+2} - i_{k+3} - \ldots - i_{l+1}$ in reverse order.

Case 2. $(i, i+1)$ and $(j, j+1)$ are backward edges in the sequence for $T'$. In this case, $T'$ is given by sequence: $i_1 - i_2 - \ldots - i_{k-2} - i_{k-1} - (i+1) - i - i_{k+1} - i_{k+2} - \ldots - i_{l+3} - i_{l+2} - (j+1) - i - i_{l+1} - i_{l+2} - \ldots - i_n - i_1$, i.e., $i_{k-1} = i+1, i_k = i, i_{k+1} = j+1$, and $i_i = j$. Performing the move $\{i, j\}$ for $T$ on the tour $T'$ generates a new tour: $i_1 - i_2 - \ldots - i_{k-2} - (i+1) - i - i_{i+1} - i_{i+2} - \ldots - i_{k+3} - i_{k+2} - i_{k+3} - \ldots - i_n - i_1$, which contains the sequence $i - i_{k+1} - i_{k+2} - \ldots - i_{j+1}$ in reverse order.

In both the cases, we obtain a new tour after performing move $\{i, j\}$ on $T'$.

We let $F(T')$ denote the set of edges $(i, i+1)$ (where $n+1$ represents 1) in $T'$ that are forward edges in its sequence. We let $B(T')$ denote the set of edges $(i, i+1)$ in $T'$ that are backward edges in the sequence for $T'$. We are now ready to state our algorithm to generate an enumeration tree of tours.

**Procedure Tree-of-Tours;**

begin
1. Let $T$ be the root of the tree;
2. Let the children of $T$ be the n-3 tours $T'$ is obtained from $T$ by performing move $\{1, j\}$ for $j = 3, \ldots, n-1$;
3. For each node $T'$ in the tree such that $T' \neq T$ do
4. begin
5. if $|F(T')| \geq |B(T')|$ then
6. Let $(i, i+1) \in F(T')$ such that $(i-1, i) \notin F(T')$. Generate children of $T'$ by performing moves $\{i, j\}$ of $T$ feasible for $T'$;
7. else Let $(i, i+1) \in B(T')$ such that $(i-1, i) \notin B(T')$. Generate children of $T'$ by performing moves $\{i, j\}$ of $T$ feasible for $T'$;
8. end;
end.

**Figure 4-3. Procedure to generate tree of some tours in 2-opt.**

The tree contains the tour $T$ as the root node. The root node has $(n-3)$ children that are obtained using the $(n-3)$ moves $\{1, j\}$ for $j = 3, \ldots, n-1$, on $T$. For each node $T' \neq T$, the children are generated as follows. If $|F(T')| \geq |B(T')|$ then we find an forward edge $(i, i+1)$ in the sequence.
of $T'$ such that $(i-1, i)$ is not a forward edge, and generate the children of $T'$ as tours obtained by performing those 2-opt moves $(i, j)$ of $T$ on $T'$ that are feasible for $T'$. If $|F(T')| < |B(T')|$ then we find a backward edge $(i, i+1)$ such that $(i-1, i)$ is not a backward edge in $T'$. In this case, we again obtain children of $T'$ by performing those 2-opt moves $(i, j)$ of $T$ on $T'$ that are feasible for $T'$.

**Theorem 4.6.** The tree generated by procedure Tree-of-Tours has size at least $(n - 3)[\lceil n/2 \rceil - 3]!$ for a TSP with $n$ nodes.

**Proof.** We first show that any two nodes in the tree are distinct tours. Let $T^1$ and $T^2$ be the tours corresponding to two distinct nodes in the tree generated by Tree-of-Tours. Let $T^0$ be the tour corresponding to the node that is the least common ancestor of the nodes for $T^1$ and $T^2$ in the tree. We assume that $|F(T^0)| \geq |B(T^0)|$; the case for $|F(T^0)| < |B(T^0)|$ is identical. In the procedure Tree-of-Tours, we select an edge $(i, i+1)$ in $F(T^0)$ such that $(i-1, i) \notin F(T^0)$ (which implies that $(i-1, i) \notin T^0$ because otherwise $(i-1, i)$ must belong to $F(T^0)$). Let the second edge incident on $i$ in $T^0$ be $(k, i)$. Based on our assumption on $T^0$, $T^1$ and $T^2$ must have distinct children of $T^0$ as ancestors. Let the ancestor of $T^1$ that is a child of $T^0$ be obtained by performing a move $(i, j_1)$ and the ancestor of $T^2$ that is a child of $T^0$ be obtained by performing a move $(i, j_2)$ where $j_1 \neq j_2$. In the tour $T^1$, the edges $(k, i)$ and $(i, j_1)$ must be incident to node $i$ as no move after the move performed at node $T^0$ can affect the edges touching node $i$. Similarly, in tour $T^2$, the edges $(k, i)$ and $(i, j_2)$ are incident to node $i$. Hence, $T^1 \neq T^2$.

We now note that for a node $T'$, each child of $T'$ is obtained by performing a 2-opt move for $T$ on $T'$. This results in removing of two edges of $T$ from $T'$. Therefore, for a node $T'$ at a level $k$ in the tree (where $T$ is at level 0), $|F(T')| + |B(T')| = n - 2k$. The number of children generated for the node $T'$ is at least $\max\{|F(T')|, |B(T')|\} - 2 \geq \lceil n/2 \rceil - k - 2$. The number of children of the root node is $(n-3)$. Thus the number of leaf nodes in the tree is at least $(n - 3)[\lceil n/2 \rceil - 3]!$. Hence the total number of nodes in the tree is at least $(n - 3)[\lceil n/2 \rceil - 3]!$ as well.

### 4.4.3 Complexity of searching 2-opt

Since the 2-opt neighborhood of a tour contains a better solution if and only if the 2-opt neighborhood of the tour contains a better solution, the complexity of finding a better solution in the 2-opt neighborhood is the same as that for the 2-opt neighborhood. However, in this section we show that the complexity of finding the best neighbor in the 2-opt neighborhood of a solution is strongly NP-hard.
In order to show our complexity result, we first show a result regarding the set of tours present in the 2-opt* neighborhood of a tour \( T = 1 - 2 - 3 - \ldots - n - 1 \).

**Lemma 4.4.** Let \( n = 4p \) for some \( p > 1 \). For any ordering \( P \) of the nodes \( \{3, 7, \ldots, 4r-1, \ldots, 4p-1\} \) there exists a tour \( T' \in \text{2-opt}^*(T) \) such that \( 1-2-P \) is the beginning of the sequence representing \( T' \).

**Proof.** We divide the set of edges in \( T \) into \( p \) groups of four edges each, where the first group is \( \{(1,2), (2,3), (3,4), (4,5)\} \) and in general the \( r^{th} \) group is \( \{(4r-3, 4r-2), (4r-2, 4r-1), (4r-1, 4r), (4r, 4r+1)\} \). Note that the \( r^{th} \) group represents the section \( (4r-3) - (4r-2) - (4r-1) - 4r - (4r+1) \) of the tour \( T \) and the middle node in the section is \( 4r-1 \), \( r = 1, \ldots, p \). Given an arbitrary ordering \( P \) of the middle nodes \( (4r-1), r = 1, \ldots, p \), we can constructively obtain a tour in \( \text{2-opt}^* \) such that \( 1-2-P \) appears as the beginning of the sequence for the tour. Suppose the given ordering is \( P = (4r_1-1) - (4r_2-1) - \ldots - (4r_p-1) \). We create a sequence of tours \( T'_j, j = 1, \ldots, p \), satisfying the following properties:

1. The beginning of the sequence representing \( T'_j \) is \( 1 - 2 - (4r_1-1) - (4r_2-1) - \ldots - (4r_j-1) - s \), where \( s \) is equal to \( 4r_1 \) or \( 4r_j-2 \).

2. All the edges in groups \( r_{j+1}, \ldots, r_p \) are present in the tour \( T'_j \).

We now describe the construction of these tours. If \( r_1 = 1 \), the \( T_1 = T \) otherwise \( T_1 \) can be obtained by performing the 2-opt move \( \{2, 4r_1-1\} \) on \( T \). Note that since the move \( \{2, 4r_1-1\} \) only affects the edges incident on nodes 2, 3, 4r_1-1, and 4r_1, the Property 2 is satisfied for \( T_1 \). Given that we have constructed \( T'_j \) for some \( j \geq 1 \), we now show how to construct \( T'^{j+1} \). There are four possible cases for the form of \( T'_j \):

Case 1. \( (4r_j-1, 4r_j) \in T'_j \) and edges in the group \( r_{j+1} \) belong to \( F(T'_j) \). In this case, the sequence for \( T_j \) starts with \( 1 - 2 - (4r_1-1) - (4r_2-1) - \ldots - (4r_j-1) - 4r_j \) by Property 1, and \( (4r_j-1, 4r_j) \notin F(T'_j) \). The move \( \{4r_j-1, 4r_{j+1}-1\} \) for \( T \) is feasible for \( T'_j \) because \( (4r_{j+1}-1, 4r_{j+1}) \notin F(T'_j) \). Lemma 4.3 shows that we can apply the move \( \{4r_j-1, 4r_{j+1}-1\} \) to obtain a new tour. We set the new tour as \( T'^{j+1} \). The move \( \{4r_j-1, 4r_{j+1}-1\} \) on \( T'_j \) removes edge \( 4r_j-1, 4r_j \) and adds the edge \( 4r_{j+1}-1, 4r_{j+1} \) to \( T'_j \). Hence \( 4r_{j+1}-1 \) is immediately after \( 4r_j \) in the sequence for \( T'^{j+1} \) and it satisfies Property 1. The move ensures that the Property 2 is satisfied as none of the edges in groups \( r_k, k > j+1 \) are affected by it.

Case 2. \( (4r_j-1, 4r_j) \in T'_j \) and edges in the group \( r_{j+1} \) belong to \( B(T'_j) \). Similar to Case 1, we note that \( (4r_j-1, 4r_j) \notin F(T'_j) \). However, \( (4r_{j+1}-1, 4r_{j+1}) \in B(T'_j) \). In this case, we perform two 2-opt moves for \( T \). We note that the move \( \{4r_{j+1}-3, 4r_{j+1}\} \) for \( T \) is feasible for \( T' \) because edges in group
Let $r_{j+1}$ belong to $B(T')$. Let the tour obtained by performing this move on $T'$ be $T'$. Since the move reverses the direction of the sequence $4r_{j+1} - 4r_{j+1} - 1 - 4r_{j+1} - 2$ in $T'$, $(4r_{j+1} - 1, 4r_{j+1}) \in F(T')$. In addition, $(4r_{j-1}, 4r_j) \in F(T')$ because the move does not affect its direction. Therefore, the move \{4r_{j-1}, 4r_{j+1}\} for $T$ is feasible for $T'$. We perform this move on $T'$ to obtain the tour $T''$. Property 1 and Property 2 are satisfied for $T''$ by argument similar to Case 1.

Case 3. $(4r_{j-1}, 4r_{j+1}) \in T'$ and edges in the group $r_{j+1}$ belong to $F(T')$. In this case, $(4r_{j-1} - 2, 4r_{j-1}) \in B(T')$ and $(4r_{j+1} - 2, 4r_{j+1} - 1) \in F(T')$. Similar to Case 2, we perform two 2-opt moves for $T$. We first perform the move \{4$r_{j+1} - 3$, 4$r_{j+1}$\} for $T$ on $T'$ to obtain the tour $T'$ such that $(4r_{j+1} - 2, 4r_{j+1} - 1) \in B(T')$ and $(4r_{j-1} - 2, 4r_{j-1}) \in B(T')$. We then perform the move \{4$r_{j-1} - 2$, 4$r_{j+1} - 2$\} for $T$ on $T'$. The move \{4$r_{j-1} - 2$, 4$r_{j+1} - 2$\} results in removing the edge $(4r_{j-1} - 1, 4r_{j+1} - 2)$ and adding the edge $(4r_{j-1} - 1, 4r_{j+1} - 1)$. Therefore, the tour $T''$ satisfies Property 1. Since the two moves performed in this case do not affect edges in groups $r_k$, $k > j + 1$, Property 2 is satisfied by $T''$.

Case 4. $(4r_{j-1}, 4r_{j+1}) \in T'$ and edges in the group $r_{j+1}$ belong to $B(T')$. We note that $(4r_{j-1} - 2, 4r_{j-1}) \in B(T')$ and $(4r_{j+1} - 2, 4r_{j+1} - 1) \in B(T')$. Hence, the move \{4$r_{j-1} - 1$, 4$r_{j+1} - 2$\} for $T$ is feasible for $T'$. We apply this move to obtain $T''$. Using argument similar to Case 1, it is easy to see that $T''$ satisfied Property 1 and Property 2.

Based on our construction, the tour $T^*$ satisfies the property claimed in the lemma.  

We now formulate the search problem for 2-opt$^*$ as a decision problem as follows.

**Input:** An undirected complete graph $G = (V, E)$, where $V = \{1, \ldots, n\}$, an integer $K$, and integer edge weights $c_{ij}$ for $(i, j) \in E$, and a tour $T$.

**Question:** Is there a tour $T'$ in 2-opt$^*$ neighborhood of $T$ such that $\sum_{(i, j) \in T'} c_{ij} \leq K$?

We refer to this decision problem as 2-opt$^*$ Search Decision Problem. To establish the hardness of finding the best solution in the 2-opt$^*$ neighborhood of a tour, we perform a polynomial transformation from the Hamiltonian Path problem to the 2-opt$^*$ Search Decision Problem. The Hamiltonian Path problem can be described as follows:

**Input:** An undirected graph $G = (V, E)$, where $V = \{1, \ldots, n\}$.

**Question:** Is there a Hamiltonian path from node 1 to node $n$ in $G$?

This problem is known to be NP-complete (Garey and Johnson [1979]). We now provide a polynomial transformation from Hamiltonian Path problem to the problem of finding the best tour in 2-opt$^*$. 

113
**Theorem 4.7.** The 2-opt* Search Decision Problem is NP-Complete.

**Proof:** Clearly, the 2-opt* Search Decision Problem is in the class NP. We now show that it is NP-complete by providing a transformation from the Hamiltonian Path Problem. Let \( G = (V, E) \) be an instance of the Hamiltonian Path problem. For notational convenience, we will assume that the \( n \) nodes of \( V \) are labeled 3, 7, ..., 4n-1, and the question is whether there is a Hamiltonian path in \( G \) from node 3 to node 4n-1. We now create a complete graph \( G' = (V', E') \) with node set \( V' = \{1, 2, ..., 4n\} \) and let \( T \) be the tour \( 1 - 2 - 3 - ... - 4n - 1 \). Using our notation, we may view \( V \) as a subset of \( V' \). We construct the edge weights \( c \) for \( G' \) as follows:

1. If \( i \in V \setminus\{3, 4n-1\} \) and \( j \in V \setminus V \), then \( c_{ij} = 1 \)
2. If \( i \in V \) and \( j \in V \) and \( (i, j) \in E \), then \( c_{ij} = 1 \).
3. For all other arcs \( c_{ij} = 0 \).

We now claim that \( G \) has a Hamiltonian path from node 3 to node 4n-1 if and only if there is a tour \( T' \) in 2-opt*(T) such that \( \sum_{(i, j) \in T'} c_{ij} = 0 \). Suppose first that there is a Hamiltonian Path \( P \) from node 3 to node 4n-1 in \( G \). By Lemma 4.4, there is a tour \( T' \) in 2-opt*(T) such that the first \( n+2 \) nodes of \( T' \) are 1-2-P. All nodes following \( P \) are in \( V \setminus V \). Regardless of how the remaining nodes in \( T' \) are ordered, the cost of \( T' \) is 0 by our construction of the edge weights.

We now consider the case that there is a tour \( T' \) in 2-opt*(T) with cost 0. At least two nodes of \( V \) are incident to nodes of \( V \setminus V \) in \( T' \). However, \( c_{ij} = 1 \) for \( i \in V \setminus\{3, 4n-1\} \) and \( j \in V \setminus V \). Since none of these arcs can be in \( T' \), we conclude that exactly two nodes of \( V \) are incident to nodes of \( V \setminus V \) and these nodes are 3 and 4n-1. This means that the nodes in \( V \) are consecutive in \( T' \). Let \( P \) denote the subpath formed by nodes in set \( V \) in tour \( T' \). We now claim that \( P \) is Hamiltonian Path in \( G \). To see this, note that any arc of \( P \) that is not in \( E \) must have a cost of 1 in \( G' \). This completes the proof.

Although it is NP-hard to search the 2-opt* neighborhood of a solution, there are some known large subsets of the 2-opt* neighborhood that are searchable in polynomial time. These include the independent 2-opt neighborhood of Potts and Velde [1995] and the twisted neighborhood structure of Aurenhammer [1988].

**4.4.4 Diameter of G^{2-opt*}**

The diameter of a neighborhood graph is another measure that is sometimes used to study the quality of different neighborhood structures. The diameter of the neighborhood graph \( G^{2-opt} \) of
the 2-opt neighborhood structure is at least \( \lfloor n/2 \rfloor \) for a TSP instance with \( n \) nodes. This can be shown as follows. Let \( T' \) be a tour of nodes \( \{1, \ldots, n\} \) such that a node \( i \) is not adjacent to nodes \( i+1 \) and \( i-1 \) for \( i = 1, \ldots, n \). One example of such a tour is the tour obtained by ordering all the odd indexed nodes in increasing order and placing an even indexed node \( i = 2j \) between \( 2j+3 \) and \( 2j+5 \) using circular order, where a number \( n+k \) is the same as \( k \). Figure 4-4 illustrates such a tour for \( n = 10 \).

![Figure 4-4. A tour such that no node \( i \) is adjacent to \( i+1 \) or \( i-1 \).](image)

It would take at least \( \lfloor n/2 \rfloor \) 2-opt moves to create the tour \( T = 1 - 2 - \ldots - n - 1 \) from the tour \( T' \) because each 2-opt move can remove at most two edges of \( T' \) from the current tour. We now show that the diameter of the \( G^{2-opt^*} \) is \( O(\log n) \). In order to show this result, we prove that the distance of an arbitrary tour \( T' \) from the tour \( T = 1 - 2 - \ldots - n - 1 \) is \( O(\log n) \).

Let the tour \( T' = i_1 - i_2 - \ldots - i_n - i_1 \). We note that performing a 2-opt move \( \{k, l\}, k < l \), results in generating a new tour in which the sequence of nodes \( i_{k+1}, \ldots, i_l \) is reversed, i.e. the new tour \( T'' \) has the form:

\[
i_1 - i_2 - \ldots - i_k - i_l - i_{k+1} - \ldots - i_{k+2} - i_{k+1} - i_{l+1} - i_{l+2} - \ldots - i_n - i_1
\]

Any move \( \{k', l'\} \) for \( T' \) where \( k < k' < l' < l \) can be applied in addition to \( \{k, l\} \) to obtain another tour. This tour has the form:

\[
i_1 - i_2 - \ldots - i_k - i_l - i_{k+1} - \ldots - i_{k'+1} - i_{l'-1} - i_l - i_{k'} - i_{k'-1} - \ldots - i_{k+2} - i_{k+1} - i_{l+1} - i_{l+2} - \ldots - i_n - i_1
\]

The move \( \{k', l'\} \) results in reversing the direction of the sequence \( i_{k'+1} - \ldots - i_n \) within \( T' \). In general, for numbers \( 1 \leq k_1 < k_2 < \ldots < k_p < l_1 < \ldots < l_1 \leq n \), the set of 2-opt moves \( \{k_1, l_1\}, \{k_2, l_2\}, \ldots, \{k_p, l_p\} \) results in a new tour. Such a set of moves is called a nested set. We use a nested set of moves to now show that given \( j, k, l \in \{1, \ldots, n\} \) such that \( j < k < l \) and \( i_j < i_k < i_l \), we can modify the subpath \( i_j - \ldots - i_l \) in \( T' \) in to a new subpath where all nodes with indices less than or equal to \( i_k \) appear before the nodes with indices greater than \( i_k \) in the subpath. We generate this set using the algorithm given in Figure 4-5.

The algorithm proceeds as follows. It identifies the first node in the the subpath \( i_{j+1}, \ldots, i_{l-1} \) of the tour \( T' \) with index greater than \( i_k \) as \( i_{r'} \) and the last node in the subpath such that its index is less than or equal to \( i_k \) as \( i_r \). It then performs the 2-opt move \( \{i_{r'-1}, i_{r'}\} \) on \( T^0 = T' \) to obtain \( T^1 \). This move results in putting \( i_{r''} \) after \( i_{r'-1} \) and \( i_{r'} \) before \( i_{r+1} \) in the tour \( T^1 \). Note that in \( T^1 \), all the
nodes between \(i_j\) and \(i_r\) are less than or equal to \(i_k\) and all the nodes between \(i_r\) and \(i_l\) are greater than \(i_k\). The algorithm then starts from positions \(j''\) and \(l''\). In each iteration, the algorithm picks a pair of nodes where the first node has index greater than \(i_k\) and the second node has index less than or equal to \(i_k\) but the second node appears after the first node in the current tour \(T^p\). The algorithm performs a nested 2-opt move to ensure that the second node appears before the first node in the new tour.

**Algorithm**  
NestedSet\((j, k, l)\);

**begin**

Let \(S \leftarrow \phi, p \leftarrow 0, j' \leftarrow j, \) and \(l' \leftarrow l, T^p \leftarrow T'\);

**while** \(j' \leq l'\) **do**

**begin**

if \(i_r\) occurs before \(i_r\) in \(T^p\) **then**

let \(j''\) be the smallest index greater than \(j'\) such that \(i_r > i_k\) and let \(l''\) be the largest index less than \(l'\) such that \(i_r \leq i_k\)

else let \(j''\) be the smallest index greater than \(j'\) such that \(i_r \leq i_k\) and let \(l''\) be the largest index less than \(l'\) such that \(i_r > i_k\)

if \(j'' \geq l''\) **then** stop and return the current set \(S\);

else add \(\{i_{r+1}, i_r\}\) to \(S\), set \(j' \leftarrow j''\), \(l' \leftarrow l''\), \(p \leftarrow p+1\), and \(T^p\) to the tour obtained by applying move \(\{i_{r+1}, i_r\}\) to \(T^p\)

**end**

**end.**

**Figure 4-5. Algorithm to generate nested set.**

We note that using the nested set of moves we can convert the tour \(T'\) into a tour \(T''\) such that all the nodes with indices less than or equal to \(\lfloor n/2 \rfloor\) appear before the nodes with indices greater than \(n/2\). Clearly this tour is in the 2-opt* neighborhood of \(T'\). Consider the last node in the tour \(T''\) that has index less than or equal to \(\lfloor n/2 \rfloor\). If the index of this node is not \(\lfloor n/2 \rfloor\), we can perform one additional 2-opt move to convert \(T''\) into a tour such that all the nodes before the node \(\lfloor n/2 \rfloor\) have index less than it and all the nodes after \(\lfloor n/2 \rfloor\) have index greater than it. Therefore, in at most two steps, we can reach a tour \(T^1\) in \(G^{2-opt*}\) starting from \(T'\) which satisfies this property. Further, our construction of the nested set of moves can be applied to the two sections of the new tour: (i) \(1 - i_2 - \ldots - \lfloor n/2 \rfloor\) and (ii) \(\lfloor n/2 \rfloor - \ldots - i_n - 1\) to further partition the nodes within each section. Using the previous argument, in at most two steps we can obtain a tour \(T^2\) such that all nodes between 1 and \(\lfloor n/4 \rfloor\) are less than \(\lfloor n/4 \rfloor\), nodes between \(\lfloor n/4 \rfloor\) and \(\lfloor n/2 \rfloor\) are less than \(\lfloor n/2 \rfloor\) and greater than \(\lfloor n/4 \rfloor\), nodes between \(\lfloor n/2 \rfloor\) and \(\lfloor 3n/4 \rfloor\) are less than \(\lfloor 3n/4 \rfloor\) and greater than \(\lfloor n/2 \rfloor\), and nodes between \(\lfloor 3n/4 \rfloor\) and \(n\) are greater than \(\lfloor 3n/4 \rfloor\).
If we continue like this, in $O(\log n)$ steps we can reach a tour $T$ in which all the nodes appear in sorted order.

### 4.5 Summary and Conclusions

In this chapter, we have introduced the concept of extended neighborhood of a neighborhood structure for combinatorial optimization problems. This concept is motivated by the observation that in certain cases two neighborhoods with very different sizes can have the same set of local optima for all instances of a problem. Hence the size of the neighborhood structure itself is not a sufficient measure of the quality of local optima. The size of the extended neighborhood provides an alternative measure of a neighborhood structure. Many small neighborhood structures can be viewed as large-scale neighborhood structures because their extended neighborhoods are large. The definition of extended neighborhood also provides an alternative way of proving exactness of neighborhood structures.

We showed that in the case of combinatorial optimization problems with general linear objective there is a geometric characterization of the extended neighborhood for any neighborhood structure. Using this characterization, we showed that for certain problems such as Graph Bipartition problem, the extended neighborhood is equal to the neighborhood structure. We also used the geometric characterization to study the size and other properties for the extended neighborhood for the 2-opt neighborhood structure for TSP. We showed that the number of neighbors of each solution in the extended neighborhood of a solution is at least $(n/2)!$, which is much larger than the size of 2-opt neighborhood itself. This result provides some explanation of the excellent empirical performance of the 2-opt neighborhood structure. We believe that in general the concept of extended neighborhood can be useful in the analysis of the neighborhood structures and perhaps also understanding the empirical behavior of neighborhood search algorithms.
Chapter 5

Conclusions and Future Work

Neighborhood search has emerged as an important tool to solve difficult combinatorial optimization problems. In this thesis, we have focused on neighborhood search algorithms based on very large-scale neighborhood structures. These neighborhood structures have extremely large number of neighbors for each solution and cannot be searched by explicit enumeration. We develop approaches to search these neighborhood structures efficiently using heuristic and exact search techniques. In addition to the work on development of very large-scale neighborhood search algorithms, we have looked more deeply at the structure of neighborhoods in general. Based on this investigation, we have proposed the concept of extended neighborhood, which provides more insight into the design of neighborhood structures in general. The contributions in this thesis can be divided into three areas. We next outline the contributions and possible future research directions in each of these areas.

Cyclic Exchange Neighborhood Structure for Partitioning Problems

We studied the Cyclic exchange neighborhood structure for partitioning problems. It is a very large neighborhood structure. We use the idea of an improvement graph, proposed by Thompson and Orlin [1989], to efficiently search this neighborhood structure. Using improvement graph, we can transform the problem of searching for a better solution in the cyclic exchange neighborhood into the problem of finding a subset-disjoint cycle in it. However, finding a subset-disjoint cycle in a graph is NP-hard in general. In this thesis, we provided following contributions.

- We develop new heuristic and exact algorithms for this problem using the concept of improvement graph. In particular, we developed a dynamic programming based algorithm that can search the neighborhood in time $O(nmK^2)$, which is only exponential in the number
of partitions. We describe methods of efficiently implementing this procedure in practice using state pruning and hash tables. We also describe a preprocessing technique for improvement graphs that can improve the performance of this algorithm.

- We developed new neighborhood search algorithms for the capacitated minimum spanning tree (CMST) problem, where we use our proposed algorithms to perform the neighborhood search. Our neighborhood search algorithms achieved or improved the best available solutions for a commonly used set of benchmark instances for CMST. We also applied cyclic exchange neighborhood search algorithms to the generalized assignment problem. Our computational results show that the cyclic exchange neighborhood is substantially more powerful that the 2-exchange, however, it takes about the same amount of time to search as the 2-exchange.

- We outline important research issues in applying the cyclic exchange neighborhood search to any partitioning problem and analyze these issues for capacitated minimum spanning tree problem. This provides a template and some insights for future application of this neighborhood structure to other partitioning problems.

Our research work shows that cyclic exchange neighborhood search is a powerful approach to solve partitioning problems. There are several problem specific research issues that need to be addressed in any application of the cyclic exchange neighborhood search. We outlined these issues in Section 2.2.4. In addition to these issues, one of the interesting research directions is to increase the scale of problem sizes that are currently solvable using cyclic exchange. In our computational testing, we have solved partitioning problems with a few hundred elements. However, several interesting partitioning problems have much larger sizes and effective implementations of cyclic exchange neighborhood search could be beneficial for them.

**Neighborhood Search for Combined Through and Fleet Assignment Model**

Airline schedule planning is a huge task involving several decisions, constraints, and objective functions. An integrated model that can take into account all these factors into account is not solvable using current technology. Therefore, the planning problem is broken into several problems that are solved in a sequential manner. Given the dependency of decisions in each stage on the decisions made in previous stages, this sequential approach inherently leads to sub-optimal solutions for the over all integrated model. Therefore, airlines are interested in integrating some of the stages in order to improve the quality of solutions produced. In this thesis, we studied the combined through and fleet assignment model (ctFAM), which integrates two of the airline
planning models: the fleet assignment model (FAM), and the through assignment model (TAM). We formulated the ctFAM as an integer-programming problem. Unfortunately, it is too large to be solved using current integer programming solves. We investigated neighborhood search approaches for solving the ctFAM. We now highlight the primary contributions of this thesis in solving airline schedule planning problems.

- We developed a swap-based neighborhood search algorithm for ctFAM that proceeds by swapping the fleet types of flights flown by two fleet types. Our neighborhood structure extends the previous neighborhoods suggested by Berge and Hopperstad [1993] and Talluri [1996] to incorporate through benefits.

- We propose the idea of an A-B improvement graph to efficiently search our swap based neighborhood structure. We investigated an alternate definition of the improvement graph where we use approximate banks to define arc costs, however, the resulting neighborhood structure is too restrictive and does not provide good results. We searched our neighborhood heuristically using an integer-programming solver.

- We implemented our neighborhood search algorithms and tested them on real-life data from a major US airline. Our computational results show that the neighborhood search can be a very effective supplement to the current integer-programming techniques being used in airline schedule planning.

- We investigated extensions of our neighborhood search algorithms to solve ctFAM with a multi-criteria objective function. Although our results with the specific criteria we studied turned out to be of low importance for airlines, the approach developed in our thesis could be useful in addressing other multi-criteria problems involving ctFAM.

Our computational results highlight the inherent dependency between the different models in airline scheduling. We could obtain large improvements in the through contribution by slightly worsening the fleeting contribution of a nearly optimal fleet assignment solution.

In conclusion, through the work in this thesis we have established neighborhood search as an important tool for the airlines to integrate and build upon the current models and solve advanced planning problems.

**Theoretical understanding of Neighborhood Structures**

The commonly accepted rule of thumb in the design of a neighborhood structure is that the larger the size of the neighborhood of a solution, the better is the quality of local optima.
There are some clear exceptions to this rule. We noted that the well-known 2-opt neighborhood structure and the independent 2-opt neighborhood structure proposed by Potts and Velde [1995] for TSP have the same set of local optima for all instances of TSP even though the 2-opt neighborhood of a solution contains only $O(n^2)$ solutions where as the independent 2-opt neighborhood contains $\Omega(1.75^n)$ solutions. Hence, the size of the neighborhood structure itself is not a sufficient measure of the quality of local optima. One of the main contributions of our work is the development of the concept of extended neighborhood. We defined two neighborhood structures to be LO-equivalent if they have the same set of local optima for all instances of the combinatorial optimization problem. Using this definition, the extended neighborhood of a neighborhood structure is defined as the largest neighborhood structure that is LO-equivalent to it. The extended neighborhood structure provides an alternative measure of the size of a neighborhood structure. Many small neighborhood structures can be viewed as large-scale neighborhood structures in terms of their extended neighborhoods. In this thesis, we analyzed some theoretical properties of the extended neighborhood. Following are the other contributions of our work.

- We provide a geometric characterization of the extended neighborhood of any neighborhood structure for those combinatorial optimization problems where the objective function can be any arbitrary linear function. Using this characterization, we showed that for certain problems such as Graph Bipartition, the extended neighborhood of any neighborhood structure is always equal to it. This suggests that a small neighborhood structure is unlikely to perform very well for these problems.

- We analyzed the extended neighborhood for the 2-opt neighborhood structure for TSP. In particular, we showed that the number of neighbors of each solution in the extended neighborhood of a solution is $(n/2)!$, which is much larger than the size of 2-opt neighborhood itself. This result provides some explanation of the excellent empirical performance of the 2-opt neighborhood structure.

We believe that in general the concept of extended neighborhood can be quite useful in the design and understanding of the neighborhood structures. We note that our definition of the extended neighborhood of a neighborhood structure is dependent implicitly on the class of objective functions that are allowed for a problem. In the future, it would be useful to analyze the extended neighborhoods for other specific neighborhood structures that are of practical interest as well as other classes of objective functions.
Bibliography


Ahuja, R. K., N. Boland, and I. Dumitrescu. 2001b. Exact and heuristic algorithms for the subset-disjoint minimum cost cycle problem. To be submitted to *Networks*.


