Algorithms for the Shortest Path Problem with Time Windows and Shortest Path Reoptimization in Time-Dependent Networks

by

Andrew M. Glenn

Submitted to the Department of Electrical Engineering and Computer Science

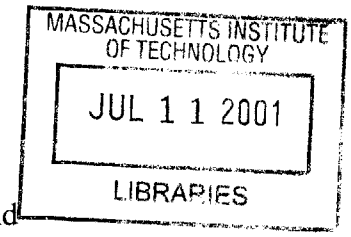
in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June 1, 2001



BARKER

Author _____
Department of Electrical Engineering and Computer Science
June 1, 2001

Certified by _____
Ismail Chabini
Assistant Professor
Department of Civil and Environmental Engineering
Massachusetts Institute of Technology
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Algorithms for the Shortest Path Problem with Time Windows and Shortest Path Reoptimization in Time-Dependent Networks

by

Andrew M. Glenn

Submitted to the
Department of Electrical Engineering and Computer Science
June 1, 2001

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

We consider the shortest path problem in networks with time windows and linear waiting costs (SPWC), and the minimum travel time reoptimization problem for earlier and later departure times. These problems are at the heart of many network flow problems, such as the dynamic traffic assignment problem, the freight transportation problem with scheduling constraints, and network routing problems with service costs at the nodes. We study these problems in the context of time-dependent networks, as such networks are useful modeling tools in many transportation applications.

In the SPWC, we wish to find minimum cost paths from the source node and all other nodes in the network while respecting the time window constraints associated with each node. We develop efficient solution algorithms to the SPWC in the cases of static and dynamic network travel times. We implement these algorithms, and we provide computational results as a function of various network parameters.

In the minimum travel time path reoptimization problem, we wish to utilize previously computed shortest path trees in order to solve the shortest path problem for different departure times from the source. We develop algorithms for the reoptimization problem for earlier and later departure times in both FIFO and non-FIFO networks. We implement these algorithms and demonstrate an average savings factor of 3 based on using reoptimization techniques instead of re-running shortest path algorithms for each new departure time.

Thesis Supervisor: Ismail Chabini
Title: Assistant Professor, Department of Civil and Environmental Engineering,
Massachusetts Institute of Technology

Acknowledgements

I would like to thank many people for their encouragement, and advice throughout the process of researching and writing this thesis. I wish to thank my thesis advisor, Professor Ismail Chabini, for his patience, insight, and support over the past year. Without his help, this entire thesis would not have been possible. His genuine concern for me, as well as for my research, was reassuring and invaluable.

I am also greatly indebted to Professor Stefano Pallottino from The Università di Pisa, for all of the time he spent discussing his proposed algorithms with me. Indeed, several of the key concepts within originated from Professor Pallottino.

I am grateful to my friends in the MIT Algorithms and Computation for Transportation Systems (ACTS) research group for their insight and advice throughout this past year.

I wish to thank my parents for always reassuring me that this thesis would eventually be completed, and for gently nudging me to always do work that I would be proud of. I would like to thank Stanley Hu for knowing every possible thing there ever is to know about computers. Finally, I wish to thank my brother Jonathan, and all of my friends for keeping me sane throughout this entire process. Most especially, thank you Eli, Elsie, and Wes for making MIT fun for the past four years.

Table of Contents

Chapter 1

Introduction

The problem of finding shortest paths in a network has been extensively studied. The most well known version of the problem is, given a network with static arc travel times, compute the shortest paths from a given source node to all other nodes in the network. However, in practical situations, many variations of the standard problem arise. These problem variants have real-world use in Intelligent Transportation Systems (ITS) and other situations that arise in the field of transportation networks analysis and operation.

The goal of this thesis is to investigate two discrete-time shortest path problem variants that are motivated by their use as sub-problems of larger transportation network flow problems. The two problems we investigate are the shortest path problem with time windows and linear waiting costs, and the problem of determining shortest paths in a time-dependent network for a set of departure times, when the shortest paths are already known for a given departure time.

For the shortest path problem with time windows and linear waiting costs, we prove several properties of optimal solutions. We utilize these properties to develop efficient

algorithms for this problem in the case of static as well as dynamic arc travel times. For the shortest path problem for varying departure times, we investigate this problem under a reoptimization framework. We assume a solution for a given departure time is known, and we wish to use this solution to efficiently compute the shortest paths for earlier or later departure times. We develop algorithms for the reoptimization of earlier and later departure times in both FIFO and non-FIFO networks.

For each algorithm presented in this thesis, we provide theoretical worst-case running time bounds. Additionally, we implement the algorithms in the C++ programming language to investigate how the running times of our algorithms would be affected by various network parameters in practical applications.

The material in this thesis is organized as follows:

Chapter 2 introduces network notation that we will be using throughout the thesis. Chapter 2 also includes a description of the time-space network. The understanding of the time-space network is useful in understanding how to solve shortest path problems in time-dependent networks.

In Chapter 3, we describe the shortest path problem with time windows and linear waiting costs. We state the problem mathematically and present optimality conditions. We then prove several properties about labels in the time-space network. These

properties lead to the development of efficient, increasing order of time, solution algorithms for both static and dynamic networks. We provide implementation details and pseudocode, as well as the theoretical worst-case running times of the algorithms described. Additionally, we present computational results based on computer implementations of these algorithms.

In Chapter 4, we investigate the shortest path reoptimization problem for earlier and later departure times from the source. We examine properties of FIFO networks in order to obtain lower and upper bounds on the shortest paths for new departure times. Using these bounds, we develop an increasing order of time solution algorithm. We develop variations of this algorithm for the reoptimization of earlier and later departure times in both FIFO and non-FIFO networks. For each of these algorithms, we provide implementation details and pseudocode, as well as a worst-case running time analysis. Finally, we provide computational results based on the computer implementations of this class of algorithms.

In Chapter 5, we summarize the main contributions of this thesis, and we suggest directions for future research.

Additionally, Appendix A provides techniques for solving the additional reoptimization variants described in Section 4.1. Appendix B provides a quick-reference glossary of the terminology used in Chapters 3 and 4 of this thesis, as well as in related literature.

Chapter 2

Notation for Traditional and Time-Expanded Networks

In this chapter we describe the formulation of discrete-time networks, both static and dynamic. We also present the time-space expansion network as a way of visualizing dynamic networks. Portions of this chapter are based on the description presented in Chabini and Dean [2].

2.1 General Network Notation

Let $G = (N, A)$ be a directed network with n nodes and m arcs. The network G is said to be dynamic if the value of network data, such as arc travel times or arc travel costs, depend upon the time at which travel along those arcs takes place.

Let $A(i)$ represent the set of nodes that come after node i in the network. That is, $A(i)$ is the set $\{j: (i, j) \in A\}$. Similarly, let $B(j)$ represent the set of nodes that come before node j in the network, such that $B(j)$ is the set $\{i: (i, j) \in A\}$. We refer to $A(i)$ as the forward star of node i, and $B(j)$ as the backward star of node j.

Let $d_{ij}(t)$ be the travel time along arc (i, j) for a departure time of t from node i. Let c_{ij} denote the constant cost of traveling along arc (i, j). In this thesis, we require that all values of $d_{ij}(t)$ be positive and integral, while c_{ij} is unrestricted in sign and value.

2.2 The Time-Space Network

The time-space network is a useful tool for visualizing and solving discrete, time-dependent shortest path problems. The time-space network is constructed by expanding the network G in the time dimension in such a way that there exists a separate copy of each node in G, one for each time t over the range we wish to investigate. We depict a sample time-expansion of a network with n nodes below:

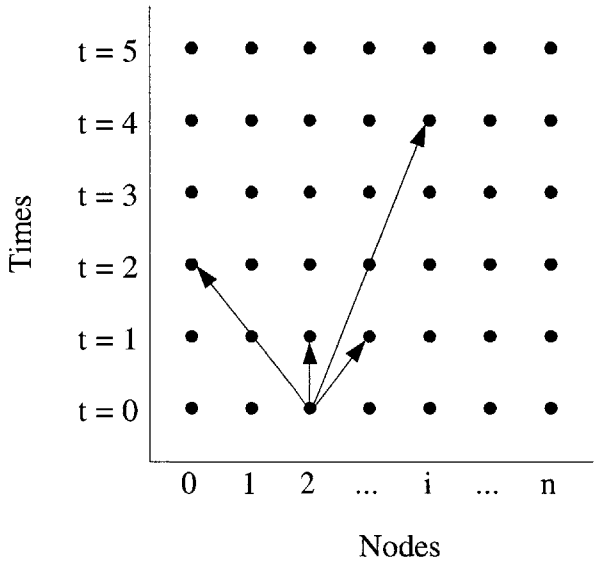


Figure 2.1 A sample time-space network. The arcs leaving node 2 at time $t = 0$ are drawn in. The arc from node 2 at time 0 to node 2 at time 1 represents waiting at node 2 for one unit of time.

We refer to points in the time-space network as node-time pairs, denoted in the form (i, t). Formally, we define the time-space network $G^* = (N^*, A^*)$ as follows. The set of nodes $N^* = \{(i, t) : i \in N, t \in \{0, 1, 2, ...\}\}$. The set of arcs A^* is the set of arcs from every node-time pair (i, t) to every other node time pair $(j, \min \{T, t + d_{ij}(t)\})$, such that $j \in A(i)$ and arriving at node-time pair $(j, t + d_{ij}(t))$ is permissible. (For example, in the case of time windows, we will not permit arcs to node-time pair $(j, t + d_{ij}(t))$ in the time-space network if $t + d_{ij}(t)$ is greater than the window upper bound for node j.) Finally, if waiting at nodes is permitted, we can create arcs of the form $((i, t), (i, t + 1))$ in the time-space network, as illustrated in Figure 2.1 by the arc $((2,0), (2,1))$.

Chapter 3

Minimum-Cost Paths In Networks With Time Windows and Linear Waiting Costs

This chapter contains the problem statement of the one-to-all minimum-cost problem in networks with time windows and linear waiting costs (referred heretofore as the SPWC problem). We present optimality conditions and a solution algorithm to solve this problem efficiently in the case of static arc travel times and in the case of dynamic arc travel times. We provide theoretical time bounds and experimental running time analysis based on several C++ implementations of the algorithm.

In Section 3.1 we provide an introduction to the problem, and we review previous work on the subject of shortest path problems with time windows. In Section 3.2 we provide a mathematical formulation and optimality conditions for the SPWC. We investigate the case of static travel times in Sections 3.3 through 3.6. In Section 3.7, we investigate the more real-world network model of time-dependent arc travel times. In Section 3.8, we provide computational results based on C++ implementations of the algorithms described in this chapter.

3.1 Problem Background and Introduction

Time windows have become a popular tool for modeling scheduling and routing problems in networks. They have been used to solve preferred delivery-time problems and the time constrained vehicle routing problem [7], as well as the Urban Bus Scheduling Problem (UBSP) and various freight transportation scheduling problems [5].

The shortest path problem with time windows (SPPTW) was formulated in Desrosiers, Pelletier, and Soumis [8]. Desrochers and Soumis [9] solve the SPPTW for the one-to-all problem with an increasing order of time permanent labeling algorithm. Desrochers [6] solves the SPPTW with additional resource constraints by a similar approach. Additional work in the field has been conducted in Ioachim et al. [16], where a dynamic programming solution algorithm for the SPPTW with linear node costs was developed. The most recent work on time windows has been the formulation of the SPWC by Desaulniers and Villeneuve, described in [5]. They examine the single origin to single destination shortest path problem with soft time windows and linear waiting costs at the nodes. The problem is formulated in terms of those in [7] and [17] through a linear programming analysis of the problem.

In this chapter, we reexamine the SPWC problem proposed in [5] to show how it can very simply be seen as an extension to the SPPTW, and thus solved by a similar approach as in [7]. We examine the one-to-all minimum cost problem for both static as well as

dynamic arc travel times. (Only the single source single destination problem for static networks was studied in [5].)

3.2 Mathematical Formulation

In this section, we mathematically formulate the one-to-all minimum-cost problem in a network with time windows and linear waiting costs at the nodes. Each arc in the network has an associated positive constant travel time, denoted as $d_{ij}(t)$, where this value is a constant for a given arc (i, j) in the case of static arc travel times. Additionally, each arc in the network has an associated travel cost, denoted as c_{ij}, which is unrestricted in sign and value.

3.2.1 Time Windows

In the networks discussed in this chapter, each node i in the network has an associated time window, denoted by $[l_i, u_i]$, such that l_i and u_i respectively represent the lower and upper bounds of the time window corresponding to node i. We refer to a time window as "hard" when arrival and departure at a node is permitted only within the time window of that node. In this chapter, we will focus however on a second type of time window, known as a soft time window. A time window is said to be "soft" if departure from a node is permitted only within the time window, but arrival at a node is permitted either within the range of the time window, or at any time before the lower bound of the time

window. In the case of soft time windows, if a commodity arrives at a node i before time l_i, it must wait at node i until time l_i before departure is permitted.

Since all nodes in the network have soft time windows, we permit waiting at any node in the network before and within its time window. We impose a cost of w per unit of waiting at any node, where w is a positive constant that is a characteristic of the network (i.e. it is not node-dependent). This waiting penalty is imposed regardless of whether the waiting occurs before or within the time window of a node. (The special case of zero-cost waiting at the origin node is addressed in sub-section 3.4.2.)

3.2.2 Path and Schedule Feasibility

A set of nodes $\{i_0, i_1, \ldots, i_{d-1}, i_d: i_k \in N\}$ comprise a path from node i_0 to i_d if and only if:

$$(i_k, i_{k+1}) \in A \qquad \forall k, 0 \leq k \leq d-1 \qquad (3.1)$$

Let a schedule S for a particular path P be a set of departure times $\{t_0, t_1, \ldots, t_{d-1}, t_d\}$ such that t_k is a departure time corresponding to node $i_k \in P$. A schedule S along a path P is feasible if and only if there exists departure times T_i such that:

$$T_i + d_{ij}(T_i) \leq T_j, \qquad \forall (i, j) \in P \qquad (3.2)$$

$$l_i \leq T_i \leq u_i, \qquad \forall i \in P \qquad (3.3)$$

A path P is feasible if and only if there exists some feasible schedule corresponding to it.

3.2.3 Label Optimality

The one-to-all SPWC problem is the problem of finding a set of minimum-cost feasible paths from a given source node s to all other nodes in the network. In a standard minimum-cost problem, a set of feasible paths is optimal if and only if they obey the well-known Bellman optimality conditions. If we denote the cost of the minimum-cost path from s to j as C_j, then Bellman's conditions say that the minimum-cost path to node j is optimal if and only if:

$$C_j = \min_{\forall i \in B(j)} (C_i + c_{ij})$$

In the case of time windows, linear waiting costs, and dynamic arc travel times, we must alter these optimality conditions slightly. Let a label for node i be defined as a node-time pair in the time-space network within the time window associated with node i. A label for node i has an associated cost, representing the cost of a feasible path that departs node i at the time associated with that label. For a given node i, let L_i^1 be the label (T_i^1, C_i^1) and let L_i^2 be the label (T_i^2, C_i^2). (To make the notation less cumbersome, we drop the node subscripts from the notation when it is clear to which node we are referring. Additionally, we drop the numerical superscripts if only one label is being discussed for a particular node.) Bellman's optimality conditions can be restated for a label (T_j, C_j) as follows. Label (T_j, C_j) is optimal if and only if it satisfies Equations 3.2 and 3.3, and:

$$C_j = \min_{\forall i \in B(j)} (C_i + c_{ij} + w(max(0, T_j - T_i - d_{ij}(T_i)))$$

The minimum arrival cost at node i is then equal to:

$$\min_k \{ C_i^k : (T_i^k, C_i^k) \text{ is an optimal label corresponding to node } i \}$$

3.3 Properties of Labels in Static Networks

To solve the SPWC problem in static networks, we propose an algorithm that implicitly searches through the nodes of the time-space network in a chronological order. We refer to this algorithm as the SPWC-Static Algorithm. In searching through the nodes of the time-space network, it creates labels that represent the current minimum-cost path to a particular node at a specific time. In order to make our SPWC-Static Algorithm efficient, we would like to find conditions on a label that allow us to determine if that label should be kept, because it may be part of a minimum-cost path, or if the label may be discarded, because discarding it will not increase the cost of the minimum-cost path to any node in the network.

The method we use to eliminate labels is to identify those labels that are dominated. We say that a label is dominated if removing that label from the network does not increase the cost of the minimum-cost path from the source node to any other node in the network. In the remainder of this section, we formalize the condition of dominance and develop

several lemmas that will allow us to exploit this characteristic. The reader may skip over the following proofs, and refer back to them later, as desired. We begin by formalizing the notion of dominance in the following lemma.

Lemma 3.1: *For a given node i, if L_1 and L_2 are distinct labels such that $T^1 \leq T^2$ and $C^1 + w(T^2-T^1) \leq C^2$, then L^1 dominates L^2.*

Proof: There are two cases. If L^2 is not on a minimum-cost path to any node, then removing L^2 will not increase the cost of any minimum-cost path, and L^2 is dominated. Otherwise, L^2 must be a part of some minimum-cost path P^2. Let S^2 be the schedule associated with path P^2 that achieves this minimum cost. Then, we can construct a path P^1 with a corresponding schedule S^1 that has an equal or smaller cost by using the label L^1 instead of the label L^2. To construct P^1 and S^1, we take the given path (and associated schedule) from the source to L^1, such that we arrive at node i at the time T^1 with the cost C^1. Assume that node j immediately follows node i along the path P^2. Then, we let the next node along P^1 be the node j, with an associated departure time from node i of time T^1. For all nodes after node j, let path P^1 and schedule S^1 be identical to path P^2 and schedule S^2.

The cost to reach any node d after node i along path P^2 using schedule S^2 is then equal to the cost to reach node i along P^2 using S^2, plus the cost of the arc (i, j), plus the cost to travel from node j to node d along P^2 using S^2. If we let C' be the cost of traveling from

node j to node d along P^2 using S^2, then cost to reach node d along P^2 using S^2, denoted by $C_d(P^2, S^2)$, is:

$$C_d(P^2, S^2) = C^2 + c_{ij} + C' \qquad (3.4)$$

Similarly, the cost to reach node d along P^1 using S^1 is equal to the cost to reach node i along P^1 using S^1, plus the cost to travel from node i to node j, plus the cost to wait at node j until schedule S^2 departs from node j, plus the cost to travel from node j to node d along P^2 using S^2. Then, the cost to reach node d along P^1 using S^1, denoted as $C_d(P^1, S^1)$, is:

$$C_d(P^1, S^1) = C^1 + c_{ij} + w(T^2-T^1) + C' \qquad (3.5)$$

Given that $C^1 + w(T^2-T^1) \leq C^2$, it follows from Equations 3.4 and 3.5 that $C_d(P^1, S^1) \leq C_d(P^2, S^2)$. Thus we have found a path P^1 that arrives at node d at the same time as path P^2, and at an equal or smaller cost. Since this holds regardless of the destination node d, we may always use L^1 instead of L^2, and thus label L^2, and any path that utilizes L^2, may be discarded without increasing the cost of any minimum-cost path. Label L^2 is therefore dominated. ■

The algorithm we describe in Section 3.4 works in increasing order of time to identify the next label to be examined. Although the concept will be fully addressed in Section 3.4, we say that a label L is "examined" when it is selected from the set of all non-permanent

labels, and a decision is made as to if L should be discarded, or added to set of permanent labels. As suggested in Pallottino and Scutella [20] for the Chrono-SPT algorithm for solving the SPPTW, when a label corresponding to a node i is examined during the course of the Chrono-SPT algorithm, it is sufficient to check only if this label is dominated by the last label that was examined and designated as permanent for node i. (If no such last-label exists, then the label being examined is trivially non-dominated.) In Lemma 3.2, we prove that the claim in [20] holds for the more general case of the SPWC (thereby also proving it holds for the SPPTW).

Lemma 3.2: *Let L^1, L^2, and L^3 be different labels for node i, such that $T^1 \leq T^2 \leq T^3$. If L^3 is not dominated by L^2 and L^2 is not dominated by L^1, then L^3 is not dominated by L^1.*

Proof: Since L^2 is not dominated by L^1 and L^3 is not dominated by L^2, we have the following equations:

$$C^1 + w(T^2 - T^1) > C^2 \qquad (3.5)$$

$$C^2 + w(T^3 - T^2) > C^3 \qquad (3.6)$$

Summing Equations 3.5 and 3.6, and rearranging terms, we have:

$$C^1 + wT^2 - wT^1 > C^3 - wT^3 + wT^2$$

which simplifies to

$$C^1 + w(T^3 - T^1) > C^3.$$

Thus, L^1 does not dominate L^3. ■

The following lemma is an interesting, if perhaps obvious, result of the dominance criteria, since it states that throwing out dominated labels in increasing order of time does not result in any loss of information about dominated labels that may exist later in time.

Lemma 3.3: *Let L^1, L^2, and L^3 be different labels for node i. If L^1 dominates L^2, and L^2 dominates L^3, then L^1 dominates L^3.*

Proof: Since L^1 dominates L^2 and L^2 dominates L^3, we have the following conditions:

$$C^1 + w(T^2 - T^1) \leq C^2 \qquad (3.7)$$

$$C^2 + w(T^3 - T^2) \leq C^3 \qquad (3.8)$$

Adding Equations 3.7 and 3.8 and simplifying, we have:

$$C^1 + w(T^2 - T^1) \leq C^3.$$

Thus, L^1 dominates L^3. ■

Lemma 3.4: *Let P be a finite, minimum cost path from the source s to a destination node d. If a schedule S along this path contains non-zero waiting within the time window of any intermediate node i along path P ($i \neq s$ and $i \neq d$), then there exists another schedule*

of equal cost along some path from s to d such in which waiting takes place within the time window of a node only at node d.

Proof: We provide two proofs of Lemma 3.4. The first is a simple logical deduction from Lemma 3.1 that proves that discarding any "waiting labels" does not increase the minimum cost for any node in the network. We also present a second proof, which actually constructs a new schedule of equivalent cost that traverses the same path P along a new schedule such that no waiting within the time windows of intermediate nodes takes place.

Let a waiting label L_i for node i be a label (T_i, C_i) such that predecessor of that label is a label L_i' for node i written as $(T_i - 1, C_i')$. Observe that the cost C_i of label L_i is equal to $C_i' + w$. By Lemma 3.1, the label L_i is a dominated label, and thus removing all paths that go through it cannot increase the cost of the minimum cost path to any node in the network.

To gain further insight as to how waiting within the time window of any intermediate node of path P can always be avoided, we provide a second proof of Lemma 3.4 by constructing a new schedule along the path P that achieves the goal of no unnecessary waiting at intermediate nodes. (We refer to waiting within the time window of a node as "unnecessary waiting.") Assume that there exists a feasible path P from the source node s to a destination node d, and a schedule S of departure times from each node in P such

that non-zero waiting takes place at a node $j \in P$ within the time window of node j. Furthermore, assume that w is strictly greater than zero, because otherwise all feasible schedules along a given path will have an equivalent cost. Let the time T_d correspond to the departure time from node d that is achieved using this schedule S. We can construct a schedule S' of equivalent cost by considering a schedule which does not wait at node j, and instead "pushes" this waiting to the next node along the path, say node k, while maintaining a cost of S' equivalent or smaller to that of S. Assume that we depart along S from node j at a time T_j and with a cost of C_j. Then, along S', we depart from j at some time T_j' such that $T_j' < T_j$. Let A_k and A_k' denote the arrival times at node k along schedules S and S', respectively. Then, we have the following equations for node j:

$T_j' < T_j$	(given)
$C_j' = C_j - w(T_j - T_j')$	(C_j is larger than C_j' in proportion to the unnecessary waiting at node j along schedule S)

And the following equations for node k:

$A_k = T_j + d_{jk}$	(arrival time at k using S)
$A_k' = T_j' + d_{jk}$	(arrival time at k using S')
$T_k = A_k + t$	(t represents the waiting time at node k along S)
$C_k = C_j + c_{jk} + wt$	(cost of departing node k along S)

Putting the above together to solve for C_k', we have that C_k' is equal to the cost of departing node j along schedule S', plus the cost of traversing arc (j, k), plus the cost of

waiting at node k until the lower bound of the time window for node k, which is no greater that time T_k.

Thus,

$$\begin{aligned} C_k' &\leq C_j' + c_{jk} + w(T_k - A_k') \\ &= C_j - w(T_j - T_j') + c_{jk} + w(T_k - A_k') \\ &= C_j + c_{jk} + wt. \\ &= C_k \end{aligned}$$

Therefore, the cost of taking schedule S' (zero waiting within the time window of node j) is no greater than taking schedule S (non-zero waiting within the time window of node j). Since this procedure can be repeated iteratively for a path consisting of any finite number of nodes, we conclude that given a path P consisting of a finite number of nodes, and a schedule S that has non-zero waiting at some of the nodes along P, we can always construct a schedule S' of equivalent or smaller cost in which there does not exist any waiting within the time window of the nodes in P, except for at the final destination node d. ■

3.4 The SPWC-Static Algorithm

In this section, we describe the SPWC-Static Algorithm. We first present a detailed description of the SPWC-Static Algorithm based on the version of the problem presented

in this thesis. We then show how to modify this algorithm to solve the SPWC problem presented in [5].

3.4.1 The SPWC-Static Algorithm for Our Network Model

The SPWC-Static Algorithm makes use of the lemmas in Section 3.3 to efficiently handle labels by not wasting computational time exploring dominated labels. The SPWC-Static Algorithm maintains an array of buckets into which candidate labels may be placed. The algorithm proceeds in increasing order of time, examining candidate labels in chronological order. When a candidate label $L_i = (T_i, C_i)$ is examined, it is removed from its time-bucket B_t (where $T_i = t$), and if it is non-dominated, it is marked as permanent.

The algorithm continues by visiting the node-time pairs in the time-space network that are reachable from the (now permanent) label L_i by a single feasible arc (i, j). We say that an arc (i,j) is feasible if there exists a time T_j such that Equations 3.2 and 3.3 are satisfied. Note that, by Lemma 3.4, waiting at node i after time l_i is never useful, and thus, when exploring from the label $L_i = (T_i, C_i)$, we do not consider the label (T_i+1, C_i) as part of the neighbor-set. For each node-time pair that is visited, the cost of arriving at that node-time pair is computed, and a candidate label is inserted in to the corresponding time-bucket.

This procedure continues for each time-bucket, until no non-empty time-buckets remain (that is, until no candidate labels remain). At this point, the algorithm terminates, and a

final search through each node's set of permanent labels is performed to determine the minimum-cost label (and thus the minimum-cost path) to that node.

To determine if a label is non-dominated, the algorithm maintains a value *last-label(i)* for each node i in the network, where *last-label(i)* holds the most recent, permanently labeled (and thus, non-dominated), label for node i. By Lemma 3.2, it is sufficient to check only if the label for node i that is currently being examined is dominated by the previous non-dominated label for node i, because if the current label is not dominated by *last-label(i)*, then it is not dominated by any permanent label for node i. Additionally, by Lemma 3.3, discarding dominated labels in increasing order of time does not result in a loss of detection of dominated labels for any labels that may exist for a later time.

The following is pseudocode for the SPWC-Static Algorithm. It was adapted from the pseudocode for the Generalized Permanent Labelling Algorithm (GPLA) in Desrochers and Soumis [7]:

Step 1: Initialize

//P_i is the set of permanent labels for node i
//B_t is a time-bucket corresponding to nodes with
//minimum arrival time t

$T = max\{ u_i : i \in N \}$
$P_i = \emptyset, \ \forall i \in N$
$C_i = \infty, \ \forall i \in N, \ \forall (T_i, C_i) : T_i \in [l_i, u_i]$
$B_0 = \{ L_{source} = (0,0) \}$
$B_t = \emptyset \ \forall t, \ 1 \leq t \leq T$
$t = 0$

Step 2: Find the next label to be examined

if $B_t != \emptyset$ then
  select a label $(T_i, C_i) \in B_t$
else if $B_{t'} = \emptyset$ for all $t' > t$, then stop
else let $t = min\{ t' > t : B_{t'} != \emptyset \}$

Step 3: Examine Label (T_i, C_i) (note $t = T_i$)

if $(last\text{-}label(i) != \emptyset)$ then
  let $(T', C') = last\text{-}label(i)$
  if $C_i \geq C' + w(T_i - T')$ then go to step 2
else
  $B_t = B_t \setminus \{ (T_i, C_i) \}$
  $P_i = P_i \cup \{ (T_i, C_i) \}$
  for all $j \in A(i)$ do
    if $T_i + d_{ij} \leq u_j$ then
      $T_j = max(l_j, T_i + d_{ij})$
      $C_j = C_i + c_{ij} + w(T_j - T_i - d_{ij})$
      if B_{T_j} contains a label $L' = (T_j, C_j')$
        $C_j' = min\{ C_j', C_j \}$
      else $B_{T_j} = B_{T_j} \cup \{ (T_j, C_j) \}$

Step 4: Compute Minimum Costs

For each node i, find the minimum cost label in P_i

Figure 3.1 The SPWC-Static Algorithm solves the one-to-all minimum cost problem in a static network with soft time windows and linear waiting costs at the nodes.

3.4.2 The SPWC-Static Algorithm for Networks with Zero-Cost Waiting at the Source

The SPWC problem as proposed in [5] allows for waiting at the source node without the imposition of any waiting penalty. To model this situation under the implementation presented in section 3.4.1, we may simply modify Step 1 of the pseudocode given in Figure 3.1 to initialize all feasible labels for the source node to have a cost of zero. This initialization procedure permits a departure from the source at any time within the time window of the source node without the imposition of any waiting penalty. The modified Step 1, which allows for zero-cost waiting at the source node, is provided in Figure 3.2 below:

Step 1: Initialize
//P_i is the set of permanent labels for node i
//B_t is a time-bucket corresponding to nodes with
//minimum arrival time t

$T = max\{ u_i : i \in N \}$
$P_i = \emptyset \ \forall i \in N$
$C_i = \infty, \ \forall i \in N, \ \forall (T_i, C_i) : T_i \in [l_i, u_i]$
$B_t = \{ L_{source} = (0,0) \} \ \forall t, \ l_{source} \leq t \leq u_{source}$
$B_t = \emptyset \ \forall t, \ u_{source} < t \leq T$
$t = 0$

Figure 3.2 The modified version of Step 1 from Figure 3.1 can be used to solve the SPWC-Static problem where zero-cost waiting is permitted at the source node.

3.5 Implementation Details for the SPWC-Static Algorithm

In the following subsections we discuss the implementation details for the SPWC-Static Algorithm. We suggest various useful data structures, and we describe several dominance strategies that could be employed by minor changes to the pseudocode in Figure 3.1.

3.5.1 Data Structures

The implementation of the SPWC-Static Algorithm is straightforward from the pseudocode of Figure 3.1. To efficiently maintain the time-buckets B_t, one may use a queue or a variety of other extensible data structures for each bucket. Each set P_i can be stored in any structure that permits *O(1)* insertion time. The cost of the *last-label(i)* may simply be stored in an array of size n, indexed by node.

The costs of all labels discovered by the algorithm can be stored efficiently by taking advantage of the fact that all labels (T_i, C_i) which will be explored over the course of the algorithm have times T_i within the time window of the node i. Thus, for each node, we can maintain an array of costs of size $u_i - l_i + 1$. We can maintain a similar array to efficiently store predecessor information.

Under this storage system, for each node i, we may also wish to maintain a cost *cost-best(i)* and a predecessor *pred-best(i)*, which hold the cost and predecessor label of the

path of cheapest cost that arrives at node i at a time t, where t need not be within the bounds of the time window for node i. We may maintain this additional information in order to relax the constraint imposed by Equation 3.2 for the final node on any path. This extra information can be easily maintained using arrays of size n.

3.5.2 Dominance Strategies

Although we have presented the algorithm as one that checks the dominance of labels upon examination of those labels, this is not the only strategy for checking dominance that one could employ. Strategies may range from the most relaxed (never checking any dominance of labels) to the most rigorous (rechecking the dominance of every single label after every single insertion or examination of a candidate label). We investigate a few of these possibilities here. (Strategies 1, 2, 4, and 5 have all been implemented, and their running times on various types of networks are illustrated in Section 3.8.)

Strategy 1: Never Check. In a worst-case example, all labels created will be non-dominated. In a scenario of this type, a strategy that does not waste time checking the dominance of any labels will actually do better than one that attempts to find dominated labels. However, in the general case, no savings will be gained by exploiting the dominance of labels, and many unnecessary labels will be explored.

Strategy 2: Check Upon Insertion. When a candidate label $L_i^k = (T_i^k, C_i^k)$ is inserted into the data structure, one could check it against *last-label(i)*. This strategy takes advantage of the last-label concept, although it may still permit dominated labels to be explored. For example, L_i^k may not be dominated by *last-label(i)* at the time of insertion, but this last-label value might change before L_i^k is selected and its forward star is explored. Thus, since the status of the *last-label(i)* is not set, L_i^k may be designated as non-dominated, even though its status might change later (undetected by the algorithm). This strategy should fare very well in practice, as most of the dominated labels will be detected, and few unnecessary labels will be created and added to the set of candidate labels.

Strategy 3: Check All Upon Insertion. To ensure that only non-dominated labels are explored, one could use a modified version of Strategy 2. In this third strategy, we first check if the new label $L_i^k = (T_i^k, C_i^k)$ is dominated by the last-label(i), before L_i^k is added to the list of candidate labels. Additionally, if L_i^k is non-dominated, then all labels for node i for times greater than T_i are checked to determine if they are dominated by L_i^k. Although this strategy would ensure that only non-dominated labels are extended (i.e. have their forward stars explored), it would have a poor running time, as labels may be rechecked several times without any reduction in the number of labels. The running time of this strategy would also suffer greatly from networks with large time windows and a

large range of arc travel times, since these factors could increase the number of labels that would be rechecked for dominance several times.

Strategy 4: Check Upon Examination. This is the strategy outlined in the pseudocode and algorithm description of Section 3.4. This strategy should fare very well in practice, as only non-dominated labels have their forward stars explored. The only disadvantage to this approach is that many unnecessary labels may be created, since dominance of a label is checked only upon a label's examination, and not upon its insertion.

Strategy 5: Check Upon Insertion and Examination. This is the hybrid strategy of 2 and 4, whereby labels are checked upon insertion and upon examination. Although theoretically better than strategies 2 and 4, since few dominated labels will ever be inserted, and no dominated labels will ever have their forward stars explored, this strategy should perform slightly worse than Strategies 2 and 4 in practice, since all non-dominated labels will have their dominance checked twice without any gain for the algorithm.

Strategy 6: Check Always. This is an extreme approach, whereby any time a new label is inserted or examined, all labels are rechecked to see if more dominated ones can be found. Other variations on this are such that this operation could be performed after k labels have been treated, for a user-defined value of k. This strategy will not fare well in

practice, since it will suffer greatly from the problem of rechecking the dominance of the same label many times.

3.6 Running Time Analysis of the SPWC-Static Algorithm

We describe the running time of the SPWC-Static Algorithm of Figure 3.1 in terms of n, m, and the following variables:

$$D = \sum_{i \in N} (u_i - l_i + 1) \text{ = the maximum number of possible labels}$$

$$d = \max_{i \in N} (u_i - l_i + 1) \text{ = the size of the largest time window}$$

In Pallottino and Scutella [18], it is indicated that their Chrono-SPT algorithm solves the related SPPTW in $O(min\{nD, md\})$. Desrochers and Soumis [7] make an identical claim for their Generalized Permanent Labeling Algorithm. However, both of these algorithms use a buckets structure, and visit each time-bucket in chronological order. As such, there is an additional factor of $O(T^*)$ in the running time of these algorithms, where $T^* = (\max_{\forall i \in N}\{u_i\} - \min_{\forall i \in N}\{l_i\})$, implying that the algorithms run in $O(T^* + min\{nD, md\})$. The factor of T^* could become significant if, for example, the time windows for each node are disjoint and far apart from each other in time, or simply if the number of labels created is relatively few but T^* is very large. For instance, such a situation could arise in a very sparse network that has very large time windows.

To ensure that T^* does not dramatically increase the practical running time of our SPWC-Static Algorithm, one could implement a version of the algorithm which employs a heap to maintain the minimum non-empty time bucket. This would lead to a running time of $O((min\{nD, md\} \log (n))$.

Since we have no a priori knowledge as to how many labels will be present in the time-buckets structure, we will use a time-buckets implementation of the SPWC-Static Algorithm. Thus, our algorithm will have a running time of $O(T^* + min\{nD, md\})$, assuming a reasonable dominance strategy is selected from subsection 3.5.2

3.7 Labels in Dynamic Networks and the SPWC-Dynamic Algorithm

In the case of the shortest path problem with time windows and linear waiting costs in a network with dynamic arc travel times, the dominance lemmas of Section 3.3 no longer apply, even if arc travel times obey the FIFO condition and arc travel costs are static. (The FIFO condition is defined in Section 4.2.) For example, consider the simple network depicted in Figure 3.3 below.

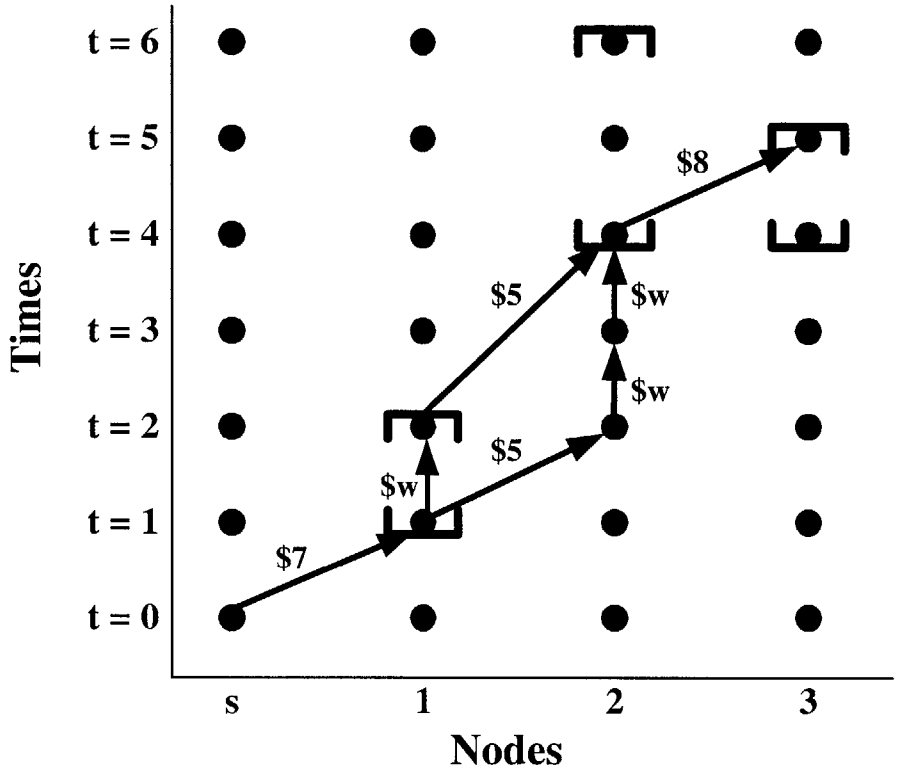


Figure 3.3 The time-space network corresponding to a topological network with 4 nodes. For each arc in the time-space expansion, the cost associated with traversing that arc is printed next to it. The lower and upper bounds of the time windows are depicted by the brackets associated with each node.

The arc *(1,2)* in Figure 3.3 has a dynamic travel time: $d_{12}(t=1) = 1$, and $d_{12}(t=2) = 2$. In this network, the minimum cost path that arrives at node *3* has value $20 + w$ for any non-negative value of w, and it is achieved by taking a schedule that involves waiting within the time window of node 1. According to the dominance lemmas of Section 3.3, such a path could be discarded, since the label for node 1 at time $t = 2$ would be considered dominated by the label for node 1 at time $t = 1$. From this example, we see that the criteria of dominance defined by Lemma 3.2 cannot be employed for the case of dynamic

travel times, since we cannot discard labels that originate from "waiting arcs" if the arc travel times are dynamic. Since waiting labels must be explored, every node in the time-space network that is reachable along some feasible path from the source will have an associated label that must be examined by our algorithm, and the forward star of this label must be explored.

The SPWC-Dynamic Algorithm is thus similar to the SPWC-Static Algorithm, with the exception that waiting nodes must be considered, and that no dominance of labels needs to be checked.

The pseudocode for the SPWC-Dynamic Algorithm is given below:

```
Step 1: Initialize
//P_i is the set of permanent labels for node i
//B_t is a time-bucket corresponding to nodes with
//minimum arrival time t

T = max{ u_i : i ∈ N }
P_i = ∅ ∀i ∈ N
C_i = ∞, ∀i ∈ N, ∀(T_i, C_i) : T_i ∈ [l_i, u_i]
B_0 = { L_source = (0,0) }
B_t = ∅ ∀t, 1 ≤ t ≤ T
t = 0

Step 2: Find the next label to be treated
if B_t != ∅ then
   select a label (T_i, C_i) ∈ B_t
else if B_t' = ∅ for all t' > t, then stop
else let t = min{ t' > t : B_t' != ∅ }

Step 3: Treat Label (T_i, C_i) (note T_i = t)
B_t = B_t \ { (T_i, C_i) }
P_i = P_i ⋃ {(T_i, C_i) }

if (T_i + 1) ≤ u_i
   B_t+1 = B_t+1 ⋃ { (T_i + 1, C_i + w) }

for all j ∈ A(i) do
   if T_i + d_ij ≤ u_j then
      T_j = max(l_j, T_i + d_ij)
      C_j = C_i + c_ij + w(T_j - T_i - d_ij)
      if B_Tj contains a label L' = (T_j, C_j')
            C_j' = min{ C_j', C_j }
      else B_Tj = B_Tj ⋃ { (T_j, C_j) }


Step 4: Compute Minimum Costs
For each node i, find the minimum cost label in P_i
```

Figure 3.4 The SPWC-Dynamic Algorithm solves the one-to-all minimum cost problem in a dynamic network with soft time windows and linear waiting costs at the nodes.

The SPWC-Dynamic Algorithm maintains a number of labels that is no more than the number maintained in the worst case of the SPWC-Static Algorithm (the case in which no domination of labels occurs). As such, the data structures presented in Section 3.5 that are used to implement the SPWC-Static Algorithm are sufficient to implement the SPWC-Dynamic Algorithm as well.

The running time of the SPWC-Dynamic Algorithm is $O(T^* + min\{nD, md\})$. Although this running time bound is the same as the running time bound for the SPWC-Static Algorithm, in practice we expect the dynamic algorithm to perform significantly worse that the static algorithm. This is because, in the dynamic case, all waiting arcs are explored, and no labels are discarded by a domination criteria.

3.8 Computational Results

Implementations of the algorithms in this chapter were written in the C++ programming language based on the pseudocode and implementation details presented in Sections 3.4 – 3.7. The tests were performed on a Dell Pentium III 933 megahertz computer with 256 megabytes of RAM.

3.8.1 Objectives

The objectives of the computational study were the following:

1. Analyze the strategies presented in sub-section 3.5.2 to determine which ones lead to efficient running times.
2. Analyze the variation of the running time of the static and dynamic algorithms as a function of following parameters: (a) size of networks with constant density; (b) number of arcs; (c) number of nodes; (d) size of the time windows.

3.8.2 Experiments

The network $G = (N,A)$ was pseudo-randomly generated, as were the link travel times. To compute the time windows, a breadth-first search of the network was conducted, and a distance label d_i was assigned to each node i, corresponding to the travel time along some feasible path from the source to node i. For a time window width of h, node i was assigned the time window $[ceiling(d_i-h), floor(d_i-h)]$. If $ceiling(d_i-h) < 0$, then the time window for node i was adjusted to be $[0, h]$. In this way, the time windows were created such that every node would be reachable within its time window along some feasible path from the source.

All running times are in seconds, and they represent the average running time over 10 trials of each algorithm. For each of the 10 trials, a source node was randomly selected from the set of all nodes in the network. Unless otherwise specified, time windows were of width 10, arc travel times ranged from 1 to 3, arc costs ranged from -5 to 5, and the waiting cost w was 2.

In the graphs below, arc travel time ranges are denoted as [minimum arc travel time, maximum arc travel time], arc travel cost ranges are denoted as [minimum arc travel cost, maximum arc travel cost], waiting costs are denoted by w, and time window widths are denoted by tww.

3.8.3 Results

Figure 3.5 shows the variations of the running times of the SPWC-Static Algorithm implemented with Strategies 1, 2, 4 and 5 as a function of network size for networks with n nodes and $3n$ arcs. It also shows the running times of the SPWC-Dynamic Algorithm as a function of these network parameters. (Such sparse networks are common in network models of traffic flows on road networks.) As suggested by the theory, Strategy 1 (never checking) and the SPWC-dynamic algorithm exhibit linear behavior since the running times of these algorithms depends solely on the number of arcs explored. The other strategies also increase in running time proportionately to the size of the networks. The rate at which they grow is difficult to predict from the theory, since it depends on the number of non-dominated labels that are added and explored. Numerical results of Figure 3.5 indicate that the fastest implementation for sparse networks is Strategy 4, followed by 5, 2, and then 1.

Figure 3.6 shows the variations of the running times of the SPWC-Dynamic Algorithm and the SPWC-Static Algorithm implemented with Strategies 1 and 4 (the best and worst strategies from Figure 3.5) as a function of the number of nodes in a network. The

number of arcs was held constant at 3000. For the static implementations, running times slightly decrease with the number of nodes for relatively sparse networks. For dynamic networks, the running times appear to fluctuate arbitrarily as a function of the number of nodes in the network. However, when the network is very sparse, the running times are dominated by the number of time-buckets that must be checked for labels. Thus, for 2950 nodes, the running times for all three algorithms increases dramatically. These running times were not shown so that the overall trends could be illustrated.

Figure 3.7 shows the variations of the running times as a function of the number of arcs. The number of nodes is held constant at 1000. As we would expect, increasing the number of arcs increases the total number of labels created under both algorithms in a linear fashion, thereby increasing the running times linearly. For all values of the number of arcs, Strategy 4 runs faster that Strategy 1, implying that checking the dominance of labels saves enough time to make the dominance checking worthwhile.

Figures 3.8 and 3.9 show the variations of the running times as a function of the size of the time windows. The number of nodes is fixed at 1000 and the number of arcs is fixed at 3000. In the case of Strategy 1, and in the case of the SPWC-Dynamic Algorithm, the running time is linear since the number of reachable nodes in the time-space network (and thus the number of feasible labels) grows linearly with the size of the time windows for both of these implementations (over networks of constant n and m). However for large values of the time window width, Strategy 4 exhibits a nearly constant running time

that is much smaller than Strategy 1's running time or the running time of the dynamic algorithm. In this case, the domination of labels plays a particularly important role. Since the time windows are large, the algorithms that examine every reachable node-time pair in the windows suffer. However, Strategy 1 explores only from non-dominated labels within the time windows, and thus it cuts off many potential paths. For window widths greater than 100, the additional number of node-time pairs in the time-space network appears to be inconsequential, because most of these node-time pairs will never be reached, since they are only reachable along paths that contain a dominated label.

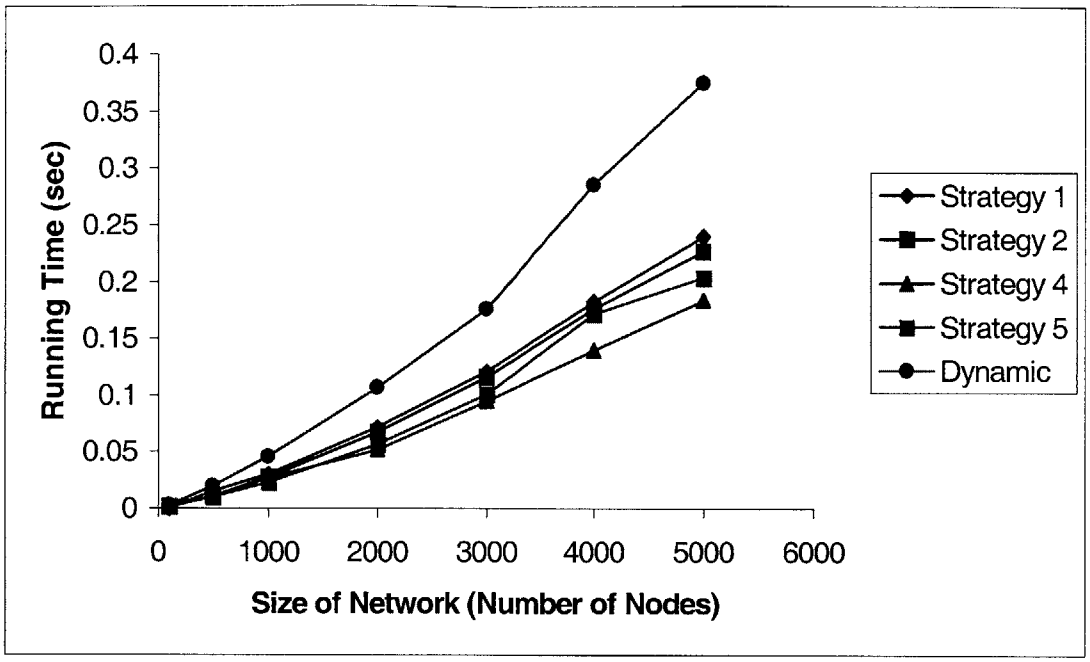


Figure 3.5 Running times of the SPWC-Dynamic Algorithm and several implementations of the SPWC-Static Algorithm as a function of network size. The number of arcs is constant at 3n. $d_{ij} \in [1,3], c_{ij} \in [-5,5], tww=10, w=2$.

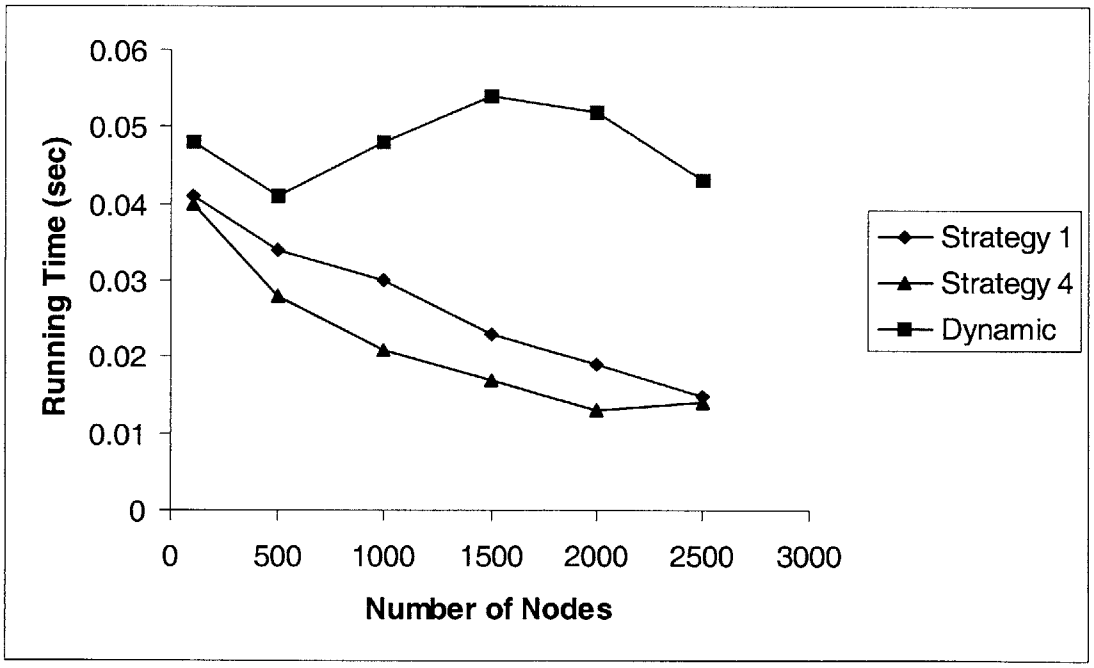


Figure 3.6 Running times of the SPWC-Dynamic Algorithm and of two implementations of the SPWC-Static Algorithm as a function of the number of nodes in the network. The number of arcs is 3000. $d_{ij} \in [1,3], c_{ij} \in [-5,5], tww=10, w=2$.

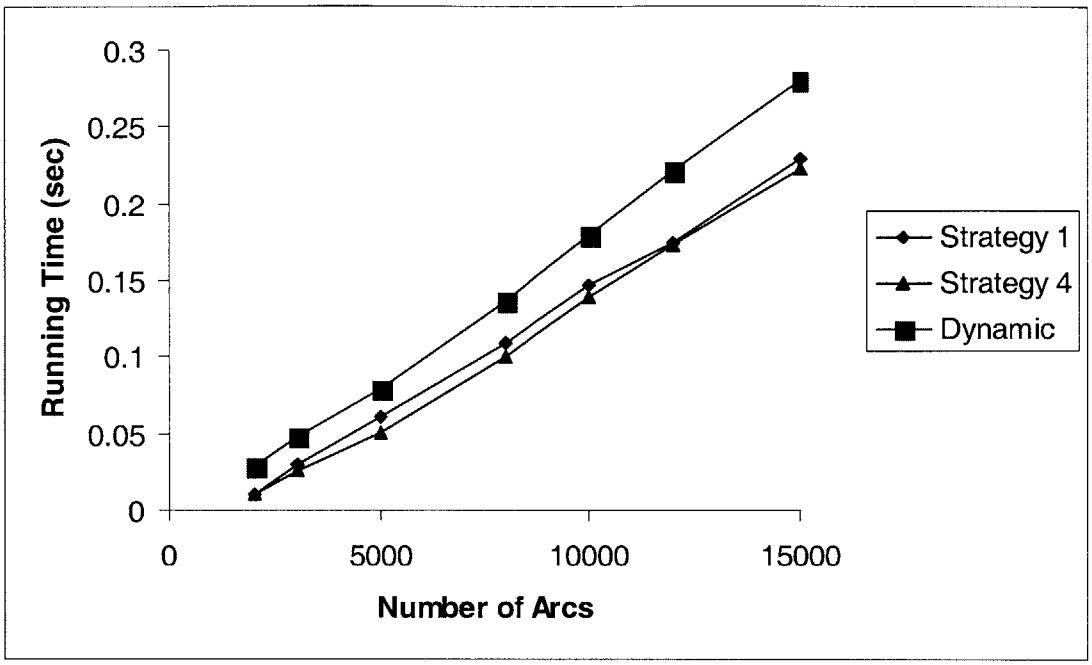


Figure 3.7 Running times of the SPWC-Dynamic Algorithm and two implementations of the SPWC-Static Algorithm as a function of the number of arcs in the network. The number of nodes is 1000. $d_{ij} \in [1,3], c_{ij} \in [-5,5], tww=10, w=2$.

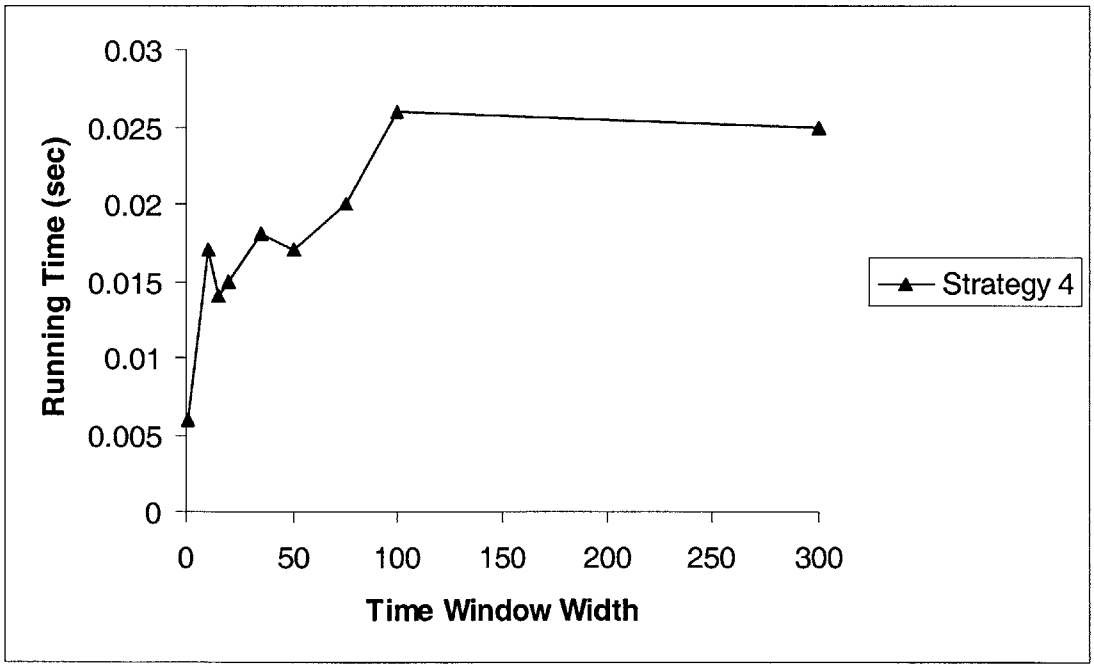


Figure 3.8 Running times of one implementation of the SPWC static algorithm as a function of the width of the time windows of the nodes in the network. The number of nodes is 1000 and the number of arcs is 3000. $d_{ij} \in [1,3], c_{ij} \in [-5,5], w=2$.

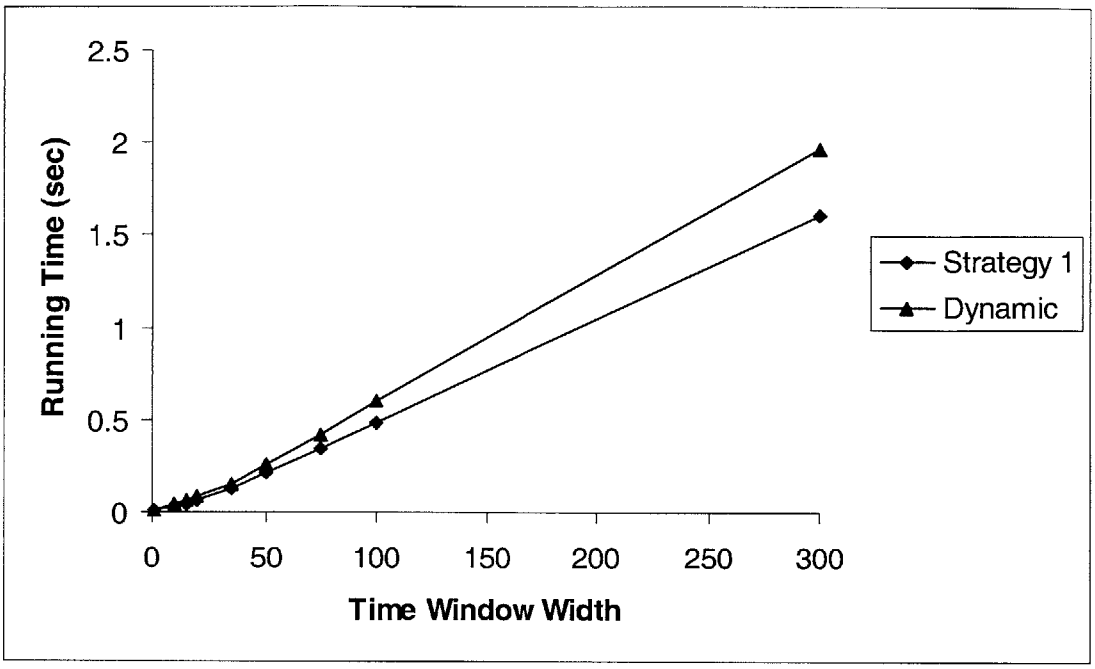


Figure 3.9 Running times of the SPWC-Dynamic Algorithm and one implementation of the SPWC-Static Algorithm as a function of the width of the time windows of the nodes in the network. The number of nodes is 1000 and the number of arcs is 3000. $d_{ij} \in [1,3], c_{ij} \in [-5,5], w=2$.

Chapter 4

Minimum-Time Path Reoptimization Algorithms

In this chapter we discuss the problem of reoptimization for the one-to-all shortest path problem in dynamic networks. We examine the problem in the context of both FIFO and non-FIFO networks. We develop a generic solution algorithm, and we describe several implementations of this generic algorithm. Each implementation uses shortest path information obtained during previous iterations of the algorithm whenever such information is available.

We begin by studying the reoptimization problem in a FIFO network for earlier departure times in Sections 4.2-4.5. We then investigate the reoptimization problem in a non-FIFO network in Section 4.6. In Section 4.7, we study the symmetric reoptimization problem, where we are given a shortest path solution from a source node s for a given departure time k, and we would like to reoptimize this solution for a departure time k' from the source, such that $k' > k$. In Section 4.8, we provide computational results for the algorithms developed in this chapter..

4.1 Problem Background and Introduction

The topic of reoptimization of network algorithms has been studied extensively recently. To date, there have been two primary areas of research in shortest path reoptimization. In the first case, the origin node changes from iteration to iteration. In the second case, the origin node remains the same, but the cost (or travel time) along exactly one arc in the network changes between iterations.

The first efficient reoptimization strategy for a change of origin node reoptimization problem was developed in Gallo [16]. Gallo's algorithm utilizes the fact that the subtree rooted at the new source node is still optimal, and it then uses reduced costs relative to the previous shortest path tree in order to compute a new shortest path tree. Later improvements were made to this procedure by Gallo and Pallottino [15] and Florian et al [12]. Fundamentally different approaches, including auction algorithms and hanging tree algorithms, have been recently proposed as new avenues of research for the change of origin in the shortest path reoptimization problem [19].

The second type of reoptimization problem (a change of arc cost or arc travel time) was first considered in Murchland [17] and Dionne [11]. Their algorithms proved to be too memory-intensive, however, and a new approach that uses Dijkstra's algorithm [10] to allow for a more memory-efficient solution was proposed in Fujishige [13]. Gallo [16] proposed an efficient reoptimization algorithm that was based on the fact that the new arc cost is always either higher or lower than the old one [20].

In this chapter, we will consider a new, third type of reoptimization problem, in which the origin node remains the same from iteration to iteration, but the departure from the origin is permitted at a time earlier or later in the $(k+1)^{st}$ iteration of the problem than in the k^{th}. Furthermore, whereas all previous reoptimization algorithms have been developed for static networks, this type of reoptimization problem is designed for networks with dynamic arc travel time data. If arc travel times are dynamic, this could have the effect of changing every travel time in the network, as opposed to changing just a single travel time as in previously studied reoptimization problems. If we view this problem in the time-space network, we note that the new solution may contain several subtrees that were a part of the previous solution, and we wish to utilize these subtrees to avoid recomputing shortest paths to the nodes in these subtrees.

Throughout this chapter, we assume that we begin with a solution to the one-to-all shortest paths problem for a given source node s and a given departure time k from the source. We refer to this shortest path tree solution as *SP(k)*, where *SP(k)* is the topological shortest path tree for a given departure time k from the source. SP(k) contains the paths in the network of minimum arrival time to every node, as well as the corresponding arrival times of these paths at each node.

The goal of this chapter is then to develop reoptimization techniques to solve any of the following variants of the shortest path problem in FIFO networks: (1) compute the

minimum arrival time paths at all nodes for a particular departure time k' from the source such that $k' < k$; (2) compute the minimum arrival time paths at all nodes for all departure times from time $k-1$ down to 0; (3) compute the minimum arrival time paths at all nodes over every departure time in a given interval (or set of intervals) of departure times $[k'', k']$ such that $k'' < k' < k$; (4) compute the minimum arrival time paths regardless of departure time for some/all values of k', such that $k' < k$; and (5) compute the shortest travel time paths regardless of departure time for some/all values of k', such that $k' < k$. Although we will speak of the reoptimization problem in the general sense (problem type 1), any of the above variants are solvable by slight adjustments to the generic algorithm we will present to solve the general reoptimization problem of type 1. (In Appendix A, we state how to solve each of the variants described above.)

4.2 Properties of FIFO Networks

Oftentimes, arc travel time functions will behave such that commodities must exit an arc in the order in which they entered. We refer to this condition as the FIFO (first in first out) condition. The FIFO condition, also known as the non-overtaking condition in traffic theory [1], may be written mathematically in a variety of ways. One definition is that the FIFO condition is valid if and only if:

$$t + d_{ij}(t) \leq t' + d_{ij}(t') \qquad \forall \{t, t': t \leq t'\}$$

Whenever the FIFO condition (also referred to as the FIFO property) holds for every arc in a network, we say that the network is FIFO.

The following lemmas and proofs are helpful to develop insight into the reoptimization problem for a FIFO network. They are also useful in the development of the efficient reoptimization algorithms described in this chapter. If desired, the reader may skip this section without loss of continuity, and refer back to it as needed.

Lemmas 4.1 through 4.3 are borrowed from Chabini and Yadappanavar [3].

Lemma 4.1: *If $f(t)$ and $g(t)$ are two non-decreasing functions, then $h(t) = f(g(t))$ is also non-decreasing.*

Proof: Since both $f(t)$ and $g(t)$ are non-decreasing functions, for $t \leq t'$, $g(t) \leq g(t')$ and for $y \leq y'$, $f(y) \leq f(y')$. Let $y = g(t)$ and $y' = g(t')$. Then, from the above, $t \leq t'$ implies that $y \leq y'$, and thus $h(t) = f(g(t)) = f(y) \leq f(y') = f(g(t')) = h(t')$. Therefore, $h(t) = f(g(t))$ is a non-decreasing function. ■

Lemma 4.2: *The composition of a finite number of non-decreasing functions is a non-decreasing function.*

Proof: By induction, using Lemma 4.1. ■

Lemma 4.3: *For any path through a FIFO network, the arrival time at the end of the path as a function of the departure time at the start of the path is non-decreasing.*

Proof: The arrival time function of a path is the composition of the arrival time functions of the arcs along that path. Every arc arrival time function in a FIFO network is non-decreasing by Equation 2.1. Thus, from Lemma 4.2, we have that the arrival time function along any path in a FIFO network is a non-decreasing function of departure time. ■

Lemma 4.4: *For any node j in a FIFO network, the minimum arrival time at node j can be found along a path that arrives at every intermediate node i in that path at its minimum arrival time value* a_i.

Proof: Arrival time functions are non-decreasing functions of departure times for any path in a FIFO network by Lemma 4.3. Therefore, if the minimum arrival time path P to node j arrives at an intermediate node i at a time $t > a_i$, then the path consisting of the optimal path to node i, concatenated with the arcs belonging to path P from i to j, will arrive at node j a time no greater than a_j. ■

Lemma 4.5: *If f(t) and g(t) are two non-decreasing functions over a given discrete interval [a,b], then h(t) = min{f(t), g(t)} is a non-decreasing discrete function over the same interval.*

Proof: If $f(a) = g(a)$, then let $a = a'$ be the first time instant $a' \in (a, b]$ such that $f(a) < g(a)$. If no such a' exists, then $f(t)$ is equivalent to $g(t)$ on $[a,b]$ and the lemma is proven. Otherwise, we assume without loss of generality that $f(a) \leq g(a)$. We define a breakpoint $t' \in [a,b]$ such that for $t \in [a,t'-1]$, $f(t) < g(t)$, but for $t = t', f(t) > g(t)$. If no such t' exists, then $h(t) = f(t)$ and the lemma is proven. Otherwise, $h(t) = f(t)$ for $t \in [a, t'-1]$, *and* $h(t) = g(t)$ for $t = t'$. The function $h(t)$ is thus non-decreasing over $[a, t']$ since $h(t) = f(t)$ *for* $t \in [a,t'-1]$, and $h(t') = g(t') > f(t')$. Repeating this procedure for every breakpoint $t' \in [a,b]$ yields the desired result. ■

Lemma 4.6: *A discrete function equal to the minimum of a finite number of non-decreasing discrete functions is itself a non-decreasing discrete function.*

Proof: By induction, using Lemma 4.5. ■

Lemma 4.7: *Let SP(k) represent the shortest path tree in a network corresponding to departure time of k at the source. In a FIFO network, the minimum arrival times SP(k) are upper bounds on SP(k-c) for every all integral values of c in the interval [1, k].*

Proof: We provide two distinct, yet related proofs of this result, as each proof offers a different insight into the problem. For our first proof, let us denote the arrival time at a node i along a path p which departs the source at time k as $a_i(p, k)$. Additionally, let

$P(s,i,k)$ denote the set of all paths from the source node s to node i for a departure time of k from the source. Let p^* be a path to node i such that $p^* \in SP(k)$. By Lemma 4.3, $a_i(p^*, k) \geq a_i(p^*, k\text{-}c)$. Since p^* is feasible for departure time $k\text{-}c$ from the source, but not necessarily optimal, we have: $a_i(p^*, k\text{-}c) \geq \min_{p \in P(s,i,k-c)} \{a_i(p, k-c)\}$. Thus, $a_i(p^*, k) \geq \min_{p \in P(s,i,k-c)} \{a_i(p, k-c)\}$, and for any node i, the minimum arrival time at node i when departing the source at time k is greater than or equal to the minimum arrival time at node i when departing the source at time $k\text{-}c$.

For our second proof, we may note that the function $a_i(t)$, representing the minimum arrival time at a given node i, is a non-decreasing function of the departure time t, by Lemmas 4.3 and 4.6. Therefore, $a_i(k) \geq a_i(k\text{-}c)\ \forall\ i$, and $SP(k)$ is an upper bound on $SP(k\text{-}c)$. ■

Lemma 4.8: *In a FIFO network, the minimum arrival times SP(k) are lower bounds on SP(k+c) for all integral values of c > 0. Furthermore, the arrival times obtained by using the paths in SP(k), but departing at time k+c from the source, are upper bounds on SP(k+c).*

Proof: No proof of this lemma is required. This lemma is a restatement of Lemma 4.7, illustrating the symmetry of the reoptimization problem. It is presented in this form to maintain the convention that the shortest path solution is always know for the departure

time k from the source, and it is this shortest path tree, *SP(k)*, that we wish to reoptimize for a different (in this case, a later) departure time. ■

Lemma 4.9: *Let path p in a FIFO network correspond to a departure from the source at time k and an arrival at a node i at time* t_i*, such that* $(i, t_i) \in SP(k)$*. Let p* be a path which departs from the source at time k-c, such that* $k\text{-}c \leq k$*, and arrives at node i at time* t_i *as well. Then the minimum arrival time at any node j, such that* (i, t_i) *as an ancestor of j in SP(k), will not decrease as a result of using path p* instead of path p to arrive at node j.*

Proof: Consider a particular node j. We assume that node i is on some minimum arrival time path from (s, k) to j. Then, the arrival time at node j along this minimum arrival time path is equal to the arrival time at j when departing node i at time t_i, by Lemma 4.4. Since path p^* departs node i at time t_i, the minimum arrival time to node j as given by *SP(k)* will not be decreased by taking the path p^*. ■

4.3 Description of the Reoptimization Algorithm in FIFO Networks for Earlier Departure Times

We maintain the convention of Section 4.1, whereby a solution to a minimum arrival time problem is known for the departure time k from the source, and we wish to reoptimize this problem for a different departure time from the source. In this section, we develop

the Reoptimization Algorithm in FIFO networks (RA-FIFO) for earlier departure times. Employing the *SP(k)* notation to denote an earlier departure time from the source, we refer to this algorithm as RA-FIFO for *SP(k-c)*.

We begin by stating in detail the algorithm for reoptimizing for time k-c, assuming *SP(k)* is known. We then suggest a theoretical improvement to the algorithm, and finally we provide the pseudocode for the generic one-to-all RA-FIFO for *SP(k-c)*. In later sections, we analyze the RA-Non-FIFO for *SP(k-c)*, and the RA-FIFO and RA-Non-FIFO for later departure times.

4.3.1 The Special Case of Static Travel Times

If arc travel times are static, then *SP(k-c)* will have the exact same topological structure as *SP(k)* , with the minimum arrival time at every node reduced by exactly c units of time. (If arc travel times are independent of time, then the solution for departure time k is an optimal solution for all departure times.)

In the case of dynamic travel times, however, the shortest paths for departure times k-c may differ substantially from the shortest paths for departure time k. Fortunately, Lemma 4.7 provides bounds on *SP(k-c)* in terms of *SP(k)*, such that *SP(k)* may help to efficiently determine if an optimal solution has been reached for a departure time k-c. The details of exactly how these bounds are used is explained in sub-section 4.3.2. In subsection 4.3.3 we discuss a theoretical improvement to achieve even tighter upper bounds.

4.3.2 Reusing Optimal Paths to Prune the Search Tree

We assume that we have the solution *SP(k)*. Note that *SP(k)* holds the predecessor tree corresponding to the shortest paths from *s* to all other nodes along paths that depart *s* at time *k*, as well as the arrival times at the destination nodes of these minimum time paths. We assume that the lengths of these paths are stored in the form of node-time distance labels, such that *(i, t)* corresponds to the arrival time label of a minimum arrival time path that arrives at node *i* at time t.

To reoptimize for *SP(k-c)*, we begin by setting the minimum arrival time at each node equal to the minimum arrival time at that node as given by *SP(k)*. We maintain a list of candidate labels, which initially includes solely the node-time pair *(s, k-c)*. The list of candidate labels holds all labels that need to be examined by the algorithm. Next, we find the candidate label of minimum arrival time; in this case the source label is the only such label. We remove the minimum arrival time candidate label from our list of candidate labels, and we update the minimum arrival times of the neighbors in its forward star.

For any node in the forward star that gets updated to a smaller minimum arrival time label, we insert the corresponding node-time pair into the list of candidate labels. Any node in the forward star that is reached at a time that is later than its minimum arrival time label need not be added to the list of candidate labels, by Lemma 4.7. Any node in

the forward star that is reached exactly at the arrival time corresponding to *SP(k)* need not be added to the set of candidate labels by Lemma 4.9.

The main loop of the algorithm thus consists of retrieving the candidate label which has the current minimum arrival time among all candidate labels, exploring its forward star according to the above procedure, and repeating this process until there are no more candidate labels.

4.3.3 Reusing Optimal Paths to Obtain Better Upper Bounds

In theory, the procedure described above can be further enhanced by noting more restrictive conditions under which a label should be added to the list of candidate labels. Since a shortest path tree is already computed for departure time k, when reoptimizing *SP(k)* for departure time k-c, we can compute even tighter upper bounds (that is, upper bounds with smaller values) on the minimum arrival time at each node for the departure time of k-c from the source. The new tighter bounds consist of the arrival time at each node i that is achieved by departing from the source node s at time k-c along the set of arcs specified in *SP(k)* to reach node i. These labels are upper bounds on the minimum arrival time at each node i since they are obtained by traversing a feasible path from the source to node i. By the proof of Lemma 4.7, they are no larger than the minimum arrival times specified by *SP(k)*. Although the minimum arrival time labels for *SP(k)* serve as good upper bounds for *SP(k-c)* if c is small, it may be beneficial to compute the new, tighter upper bounds if c is large. In general, if the shortest path tree *SP(k)* does not

differ by much from *SP(k-c)*, these new bounds may be optimal for many nodes, thus reducing the number of times each node has its minimum arrival time updated. In the special case where c is large, *SP(k)* may be a poor upper bound on SP(k-c), and thus finding a tighter upper bound may be advantageous.

Additionally, if the range of possible arc travel time values is large, then using the tighter upper bounds may be advantageous, as many minimum arrival time updates for nodes will be avoided. However, over all sample networks tested, the time spent to update the upper bounds to these potentially tighter values was greater than the savings gained by using the tighter upper bounds. These results are provided in Section 4.8.

To reduce the time spent on updating the nodes to their tighter bounds, one could implement a procedure where only the nodes that were examined in the previous iteration have their upper bounds updated to the tighter values. This procedure can be implemented with very little additional computation time. However, it does not update enough of the labels to their smaller values during any single iteration, and the running times achieved are nearly identical to not simply using the relaxed upper bounds described in subsection 4.3.2. For this reason, the running times for this implementation are omitted.

4.3.4 Pseudocode for the Generic RA-FIFO for *SP(k-c)*

The following pseudocode summarizes the Generic RA-FIFO for *SP(k-c)*:

```
Step 1: Initialize
//The method ShortestPaths(n,m,k) sets pred(i) and
//mintime(i) ∀ i

SP(k) = ShortestPaths(n,m,k)
SP(k-c) = SP(k)

Step 2: Reoptimize for time k-c < k
SP(k-c) = SP(k-c) \ { source }
Candidates = { (source, k-c) }

Step 3: Main Loop
while Candidates != ∅ do
   (i, t) = arg_mintime( Candidates )
   SP(k-c) = (SP(k-c) \ {i, mintime(i)}) ∪ {(i, t)}
   For all successor j of node i do
      a_j = t + d_ij(t)
      if a_j < mintime(j)
            mintime(j) = a_j
            Candidates = Candidates ∪ { (j, a_j) }
```

Figure 4.1 The generic RA-FIFO for solving the one-to-all shortest path problem in FIFO networks for a departure time from the source at time k-c < k. The method *arg_mintime* returns the node-time pair corresponding to the candidate label of minimum arrival time.

4.4 Implementation of The Reoptimization Algorithm in FIFO Networks for Earlier Departure Times

In the FIFO case, it is sufficient to store the node-time pair (i, t) (along with the predecessor of node i on the path which arrived to node i at time t) for just the minimum arrival time corresponding to node i. To see that we need to store only the path information for the shortest path to node i and not as well other paths which go through node i at times $t^* > t$, we must demonstrate that we are not discarding any information

which may be useful in computing the shortest paths to other nodes j in the current reoptimization phase, say $SP(k')$, and we must ensure that we are not discarding any information that will be useful in computing shortest paths in later reoptimizations, $SP(k'')$ for some $k'' < k' < k$.

For the current reoptimization phase $SP(k')$, we need only to store the minimum arrival time labels for each node since Lemma 4.4 implies that all shortest paths need only be composed of such labels. Furthermore, by Lemma 4.7, maintaining only the minimum time labels when reoptimizing for $SP(k)$ is all that will be necessary for the reoptimization of $SP(k\text{-}c)$, since these labels are upper bounds for $SP(k\text{-}c)$.

Thus, it is sufficient to maintain only the minimum arrival time label for each node in the network. In algorithms which parallel Dial's and Dijkstra's algorithm for the standard shortest path problem, we can now describe two separate implementations of the generic algorithm, based on how we locate the next minimum time label that needs to be updated. The first implementation is based on a time-bucket data structure, and the second utilizes a binary heap.

4.4.1 Time-Buckets Implementation Details and Running Time Analysis

To facilitate a chronological scan of the labels, we may store the labels in "time-buckets." We maintain a time-bucket for each time from ($k^* = k$ + *the maximum shortest path from*

s to all other nodes when departing s at time k) down to 0. We describe the details of the time-buckets and heap implementations in the following subsections.

4.4.1.1 Time-Buckets Implementation Details

The goal of our time-buckets implementation is to provide an implementation that is efficient both in terms of memory usage and computation time required. To achieve this, we will maintain labels in time-buckets such that all operations on these labels can take place in *O(1)* time.

Since we do not know the value of the maximum arrival time along all minimum travel time paths before these paths are computed, we may upper bound the value of k^* to ensure that enough time-buckets are created during the initialization phase of the algorithm. Note that we cannot use a circular queue to store the labels, since such a data structure would overwrite information from previous shortest path solutions, which our algorithm utilizes to reoptimize those solutions for earlier departure times.

Time-bucket t holds a doubly-linked list of labels, where a label is a node-time pair (i, t) corresponding to a path that arrives at node i at time t. The label also holds the predecessor of node i on this path, and a flag such that candidate labels are marked "dirty." That is, a label (i, t) is dirty if we need to update the neighbors in the forward star of (i, t). If a label is not in the set of candidate labels, then it is denoted as "clean." (Note that if candidate labels are maintained in a separate data structure, then the

clean/dirty flag becomes unnecessary. It is described here in order to illustrate a memory-efficient implementation of the algorithm.)

To keep track of the time-buckets, we maintain one bucket-pointer for each time t. Each bucket-pointer points either to the head node of a doubly-linked list of labels corresponding to a particular minimum arrival time, or it points to null (if there are no labels for that time). In this way, we need only to maintain an array of bucket-pointers, plus, for each node i, 2 additional pointers for the doubly-linked list, for a total memory size on of $O(n+k^*)$. We must also maintain the predecessor and a clean/dirty boolean variable for each node i. Both of these pieces of information can be stored in arrays of size n indexed by node.

We now show that we can use these data structures to perform label updates in $O(1)$ time. Examining label (i, t) consists of removing it from its current time bucket, inserting it into a new time-bucket, changing its flag to dirty, and possibly updating its predecessor information. (It is possible that node i will be updated to a smaller minimum arrival time value, but maintain the same predecessor node in the new shortest path tree, if the minimum arrival time at that predecessor has changed.)

To remove the label (i, t), we look up the node after node i in its linked list, and the node before node i in the linked list to which it belongs. We then set the pointers accordingly to remove node i from the linked list. Each of these operations require $O(1)$ time.

To insert node i into a new bucket-list, we look up the bucket-pointer for the time t. We insert node i in front of the head node in this time-bucket, and we update the bucket-pointer to make node i the new first label for the time-bucket t. Setting predecessor and clean/dirty information require a single array access each. Thus, inserting a label requires $O(1)$ time. Inserting dirty labels at the front of the list results in a valuable time savings, since once a clean label is found in the current time bucket, the examination of the remainder of labels in that time-bucket can be halted.

The following is pseudocode for the RA-FIFO for *SP(k-c)* using time-buckets:

```
Step 1: Initialize
//The method ShortestPaths(n,m,k)places node i in
//time-bucket B(mintime(i)) ∀ i

For t = 0 to k*
   B(t) = Ø
SP(k) = Run ShortestPaths(n,m,k)

For i = 0 to n-1
   flag(i) = clean
dirtycount = 0

Step 2: Reoptimize for time k-c < k
mintime(source) = k-c
setflag(source) = dirty
t = k-c

Step 3: Main Loop
while (dirtycount > 0) do
   For all i ∈ B(t) such that flag(i) is dirty
      for all j ∈ A(i) do
            a_j = t + d_ij(t)
            if (a_j < mintime(j))
                  mintime(j) = a_j
                  setflag(j) = dirty
      setflag(i) = clean
   t = Time of next non-empty time-bucket
```

Figure 4.2 The RA-FIFO for *SP(k-c)* of Figure 4.1 implemented using time-buckets. The method *setflag* sets the clean/dirty flag of a node, and updates the variable dirtycount as necessary.

4.4.1.2 Time-Buckets Running Time Analysis

Providing accurate average running time bounds is difficult because of the randomness associated with the FIFO travel times. However, we can easily bound the worst-case running time of the RA-FIFO for *SP(k-c)*.

Since reoptimizing assumes that we begin with one solution, we exclude the time for Step 1 in the pseudocode of Figure 4.2 when discussing the running time of the algorithm. To analyze the running time, we assume we are given the solution for *SP(k)*, and we wish to reoptimize for one reoptimization phase (reoptimizing for one new departure time *k*-c from the source). For one reoptimization phase, in the worst case, every minimum time label will be smaller than its previous value, and no information from the previous computation will be useful at all. In this case, the algorithm degenerates to a dynamic variant of Dial's algorithm, implemented using a basic buckets data structure (i.e. implemented without using a circular queue data structure). For a network with n nodes, m arcs, and a maximum simple path length of $\mathcal{T}_{k-c}$ for a departure time of k-c from the source, the running time is $O(m + \mathcal{T}_{k-c})$. Observe that $\mathcal{T}_{k-c}$ can be no greater than n-1 times the maximum arc travel time in the network. To reoptimize over all $k' < k$, the running time is $O(km + k\mathcal{T}_k)$.

However, we will show in the computational results subsection that the average running time will be much better than the worst-case running time. The factor of $O(m)$ in the running time should be much smaller on the average case because the results of the previous computations should allow the reoptimization to be completed with the inspection of fewer than m arcs and with potentially even fewer decreases of labels into different time-buckets, especially if reoptimizations are performed for k-1, then k-2, etc.

4.4.2 Heap Implementation Details and Running Time Analysis

A second method to facilitate locating candidate labels is the use of a binary heap. In the worst case, the running time obtained by using a heap can be greater than the worst-case running time for the time-buckets implementation. However, for most practical problem instances, using a heap can result in significant speedups.

4.4.2.1 Heap Implementation Details

At most n labels are stored in the heap at any one time, corresponding to the current minimum arrival time labels for each node in the network. Insertion and the updating of minimum arrival time labels can be performed in $O(log(n))$ time with a heap data structure. Identification of the minimum element in the heap takes $O(1)$ time, and removal of this element takes $O(log(n))$. Predecessor labels can be maintained using a simple array, as in the time-buckets implementation, and thus updating a predecessor label can be performed in $O(1)$ time. To make heap operations run as efficiently as possible, labels are inserted into the heap only when necessary, so that heap operations whose runtime depend on the number of elements in the heap will operate efficiently. The pseudocode presented in Figure 4.3 demonstrates this idea.

The following is pseudocode for the RA-FIFO for *SP(k-c)* using a binary heap:

```
Step 1: Initialize
//The method ShortestPaths(n,m,k) sets mintime(i) ∀ i
Run ShortestPaths(n,m,k)

Step 2: Reoptimize for time k-c < k
mintime(source) = k-c
Insert( source, k-c )

Step 3: Main Loop
while heap is not empty do
   (i, t) = minimum-time label in heap
   for all j ∈ A(i) do
      a_j = t + d_ij(t)
      if (a_j < mintime(j))
            if ( j is not in the heap )
                  Insert( j, a_j )
            else
                  DecreaseKey( j, a_j )
```

Figure 4.3 The RA-FIFO for *SP(k-c)* of Figure 4.1 implemented using a binary heap.

4.4.2.2 Heap Running Time Analysis

Similarly to the time-buckets implementation, providing accurate average running time bounds is difficult because of the randomness associated with the FIFO travel times. However, we may easily bound the worst-case running time. We assume that are given a shortest path solution, and thus we exclude the time for Step 1 in the pseudocode above when discussing the running time of the binary heap implementation.

For one reoptimization phase, in the worst case, every minimum time label will be smaller than its previous value, and no information from the previous computation will be useful. In this case, the algorithm degenerates to a dynamic extension of Dijkstra's

shortest path algorithm implemented with a binary heap. For a network with n nodes and m arcs, the running time is $O(m + mlog(n))$.

However, the average running time of the heap implementation will be better than the worst-case bound, since both the first, $O(m)$, and second factor, $O(mlog(n))$, will be smaller than their worst-case bounds for practical problem instances. We expect the first factor to be much smaller in the average case because of the reuse of the results from previous computations, which will result in fewer than m label examinations. We expect the second factor to be smaller than its worst-case bound since the number of updates to the heap should be fewer than m. Note that m is the maximum number of updates that can be performed on the heap, and even in a standard shortest path algorithm, fewer than m updates will actually be performed. In our reoptimization algorithm, the upper bounds provided by the previous computation will limit the number of heap updates to even fewer. Additionally, the number of elements in the heap at any given time should be fewer than n for practical network instances, implying that any single heap operation can be performed in faster than $log(n)$ time.

4.5 The Reoptimization Algorithm in non-FIFO Networks for Earlier Departure Times

For the non-FIFO reoptimization problem for earlier departure times, we assume we are given the minimum arrival time paths from the source node to every other node in the

network for a given departure time k from the source. For a given departure time $k\text{-}c < k$, we would like to solve the minimum arrival time problem regardless of departure time from the source. That is, we would like to find the minimum arrival time at all nodes in the network if we permit departure from the source at either time k or time $k\text{-}c$. Note that in the case of FIFO networks, this problem is trivial, as it is always given by *SP(k-c)*. However, for non-FIFO networks, there is no guarantee that departing from the source node at earlier time will lead to earlier arrival times at any of the nodes.

This problem may be extended to finding the minimum arrival time regardless of departure time over three or more departure times k', k'', k''', etc., such that $k''' < k'' < k' < k$. Similarly, it may be solved for an interval of departure times $[k'', k']$ such that $k'' < k' < k$ by solving the problem for each departure time in the desired range. For clarity, we will discuss this problem in the most basic context of finding the minimum arrival times regardless of departure time, for the given departure time k and a single, earlier departure time, $k\text{-}c$. Furthermore, we will refer to the algorithm used to reoptimizing a solution for time k for earlier departure times in non-FIFO networks regardless of departure time as the RA-Non-FIFO for *SP(k-c)*.

4.5.1 Description of the RA-Non-FIFO for Earlier Departure Times

To efficiently reoptimize an all-to-one minimum arrival time solution regardless of departure time in non-FIFO networks, we will use one of the basic principles of the RA-FIFO for *SP(k-c)*. Although Lemmas 4.4 and 4.7 no longer apply in the case of non-

FIFO networks, we can still use the same basic reoptimization procedure as outlined by the Generic RA-FIFO for *SP(k-c)*. The change that we will make is that many labels must now be saved for a given node, because minimum arrival time paths in non-FIFO networks are not necessarily composed of the minimum arrival time labels for each node on that path. Additionally, since *SP(k)* no longer represents upper bounds on *SP(k-c)*, any label created during the shortest path algorithm for departure time k might be useful for *SP(k-c)*, not just the minimum arrival time labels.

The following is pseudocode for the Generic RA-Non-FIFO for *SP(k-c)* regardless of departure time:

```
Step 1: Initialize
SP(k) = ShortestPaths(n,m,k)

Step 2: Reoptimize for time k-c < k
Candidates = { (source,k-c) }

Step 3: Main Loop
(i, t) = arg_mintime( Candidates )
while (i, t) != Ø and
(i, t) < argmax{mintime(v) ∀v ∈ N}
do
   for all j ∈ A(i) do
      a_j = t + d_ij(t)
      if label (j, a_j) does not exist
            Candidates = Candidates U { (j, a_j) }

Step 4: Compute Minimums
For each node i, compute the minimum arrival-time
   label.
```

Figure 4.4 The Generic RA-Non-FIFO for *SP(k-c)* for solving the one-to-all shortest path problem in non-FIFO networks regardless of the departure time from the source. The method *arg_mintime* returns the node-time pair corresponding to the candidate label of minimum arrival time.

4.5.2 Implementation of the RA-Non-FIFO for Earlier Departure Times

Just as the RA-FIFO implementation for earlier departure times has two different implementations based on time-buckets or a heap data structure, so does the non-FIFO version. With the exception of the details outlined below, these implementations translate directly from the pseudocode in Figure 4.4 and either Figure 4.2 (time-buckets) or Figure 4.3 (heap). Thus, we omit the pseudocode for these specific implementations.

In the non-FIFO solution algorithm, there may be many labels in different time-buckets that correspond to the same node. Thus, if we denote the travel time of the longest minimum arrival time path in the network corresponding to a departure time of k from the source node as $\mathcal{T}_k$, then it may be necessary to store up to $(k + \mathcal{T}_k)$ time-buckets, each of which may hold up to n labels. In the worst case, all n nodes are dirty in each time-bucket, and need to have their forward stars explored. This implies that m arcs will be checked in one phase of the reoptimization algorithm (reoptimizing for one other departure time, k-c, from the source), thus implying a running time of $O(m\mathcal{T}_k)$. (To solve the reoptimization problem for all times from k-1 down to zero, the total running time would be $O(m(k+\mathcal{T}_k))$, since at most every arc in the time-space network will be explored.)

For the heap-based implementation, the heap will contain all dirty labels. In the worst case, for a departure time k-c from the source, the heap may contain $O(n\mathcal{F}_{k-c})$ labels. This implies a running time of $O(m\mathcal{F}_{k-c}\ log(n\mathcal{F}_{k-c}))$ for the reoptimization problem for one earlier departure time, time k-c, from the source. (To reoptimize for all departure times from k-1 down to zero using the heap-based implementation would incur a running time of $O(m(k+\mathcal{F}_k\ log\ (n\mathcal{F}'))$ for the entire algorithm, where $\mathcal{F}' = \max_{t\in[0,k]} \mathcal{F}_k$.)

The results presented in Section 4.8 for the RA-Non-FIFO for $SP(k$-$c)$ were obtained using a simplified buckets data structure, for ease of implementation. This implementation stores each time-bucket as an array of size n (we refer to these arrays as time-arrays). Implementation of the algorithm using this data structure leads to a worst-case running time that is equal to the worst-case running time of the more efficient time-buckets implementation described above, but it causes poorer average case performance, since, for each time-array, the entire array will have to be scanned for dirty labels.

4.6 Reoptimization for Later Departure Times

Finally, we address the symmetric problem of reoptimizing for a later departure time in both FIFO and non-FIFO networks. We will first investigate the FIFO case, and then we will turn to the non-FIFO case.

4.6.1 The Reoptimization Algorithm in FIFO Networks for *SP(k+c)*

In the FIFO case, Lemma 4.8 provides us with upper bounds on *SP(k+c)*, based on using the paths of *SP(k)*, but departing at time $k+c$ instead. However, computing these new paths can be time-consuming if the shortest path tree *SP(k)* is maintained by solely storing the predecessor of node i for all nodes i, and not storing the set of children of i, for all nodes i.

The role of reducing the number of nodes that are searched is played by the lower bounds (the values of *SP(k)*) when we are reoptimizing for *SP(k+c)*. Unfortunately, these lower bounds cannot effectively reduce the number of such labels that will be searched in the *SP(k+c)* reoptimization for the following reason. Although the arrival times of *SP(k)* do provide lower bounds for the arrival times of *SP(k+c)*, these lower bounds may not be feasible arrival time values for any given node on a path that departs the source at time $k+c$. Therefore, we cannot begin our reoptimization algorithm for *SP(k+c)* by assuming that these lower bounds are optimal and working from there, as we did in the reoptimization algorithms for *SP(k-c)*. This implies that if a path is found from departure time $k+c$ from the source to a node i at the time *SP(k)*, we must still explore the forward star of node i. Since the nodes in forward star of i are not initialized with optimal labels (since these labels may be infeasible), the nodes in the forward star of i must be updated.

Furthermore, it would be incorrect to simply update all of the nodes in the forward star of node i with their optimal lower bounds and stop searching from all of these nodes (as we

do in the *SP(k-c)* case). Even once their labels are updated to their minimum values, we still need to explore from all of the nodes in the subtree of node i. This is because the nodes in the solution to *SP(k+c)* may not have been in the subtree of node i for *SP(k)*.

Finally, one may wish to use the paths given by SP(k), but depart along these paths at time $k+c$ to obtain feasible upper bounds on the minimum arrival times at all nodes for a departure time of $k+c$ from the source. However, this method is an analog to the approach of finding tighter upper bounds described in subsection 4.3.3, and, the extra computation time spent to update every node with these feasible upper bounds is not worthwhile. As the computational results in section 4.8 show, the running time of the RA-FIFO for *SP(k+c)* suffers greatly from the additional work that must be performed, and as such, it is faster to re-run a shortest paths algorithm for time $k+c$ from scratch than to "reoptimize" the *SP(k)* solution.

4.6.2 The Reoptimization Algorithm in non-FIFO Networks for *SP(k+c)*

The RA-Non-FIFO for *SP(k+c)* is nearly identical to the symmetric non-FIFO algorithm for *SP(k-c)*. Since the *SP(k-c)* algorithm for non-FIFO networks did not make use of any of the lemmas of Section 4.2, it is unaffected by the problems of efficient FIFO reoptimization detailed in the previous subsection. It is straightforward to reoptimize the *SP(k)* solution for the minimum arrival times at all nodes regardless of departure time from the source for a later departure time $k+c$, for the following reason.

In the pseudocode of Figure 4.4 for the Reoptimize-Non-FIFO *SP(k-c)* problem, we see that the reoptimization phase terminates once the time of the smallest candidate label is as big as the maximum value of all of the minimum arrival time labels, because at that point no more updates could possibly be made (since d_{ij} is positive for all arcs in the network). In solving for *SP(k+c)*, this condition is still valid, but we have an additional stopping criteria if we are reoptimizing for a set of departure times *[k+1, k+c]*. Let $t \in [k+1, k+c]$ be the departure time from the source in the current phase of the reoptimization algorithm over the set of departure times *[k+1, k+c]*. Then, the additional constraint is that as soon as the maximum value of all minimum arrival time labels is smaller than t, no more reoptimizations need be performed, since the minimum arrival time labels will not decrease for any nodes in the network. Although this termination constraint implies that the RA-Non-FIFO for later departure times will have a very efficient running time in practice, this algorithm was not implemented at the time of the writing of this thesis, and thus no numerical results are provided.

4.7 Computational Results

All codes tested in this section were written in the C++ programming language, based on the pseudocode and implementation details in Sections 4.3 – 4.7. The tests were performed on a Dell Pentium III 933 megahertz computer with 256 megabytes of RAM.

4.7.1 Objectives

The objectives of the computational study were the following:

1. Compare the variation of the running times of the time-buckets and heap implementations of the RA-FIFO for *SP(k-c)* as a function of the following parameters: (a) the size of networks with constant density; (b) the number of arcs; (c) the number of nodes; (d) the number of reoptimizations performed
2. Analyze the running times of the non-FIFO reoptimization algorithm for *SP(k-c)* as a function of parameters (a) - (d) above.
3. Compare the running times achieved by using the reoptimization algorithm instead of repeating a standard shortest path algorithm for each of the following implementations: RA-FIFO for *SP(k-c)* using time-buckets, RA-FIFO for *SP(k-c)* using a heap with relaxed upper bounds, RA-FIFO for *SP(k-c)* with a heap using tighter upper bounds, RA-Non-FIFO for *SP(k-c)* using time-buckets, and RA-FIFO for *SP(k+c)* using time-buckets.
4. Compare the running time of the RA-FIFO for SP(*k+c)* with the corresponding standard shortest path algorithms that starts from scratch at each iteration.

4.7.2 Experiments

The network $G = (N,A)$ was pseudo-randomly generated. The arc travel times were pseudo-randomly generated as integers in the range from 1 to 3 (denoted as $d_{ij} \in [1,3]$ in

Figures 4.5 – 4.21). Dynamic travel times were generated such that arc travel time data was dynamic for all departure times from all nodes in the network. Unless otherwise specified, a solution for *SP(k=100)* was assumed to be known, and this solution was reoptimized for all k from 99 down to 0. (Thereby solving the reoptimization problems for $c = 1$.) Savings ratios were computed by running the standard shortest path version of the algorithm over the same 100 departure times from the source. All running times are in seconds, and they represent the average running time over 5 trials of each algorithm, for the 100 iterations (or, "reoptimization phases"). For each of the 5 trials of the algorithm, a node was chosen randomly to be the source node.

4.7.3 Results

Figure 4.5 shows the variation in running time of the RA-FIFO for *SP(k-c)* as a function of network size. Running times are provided for the time-buckets implementation, the heap implementation using relaxed upper bounds, and the heap implementation using tighter upper bounds. Running times scale approximately linearly for all three implementations, and in virtually all cases, the fastest implementation is the heap with relaxed upper bounds. The linear scale is consistent with the running time of the algorithm, which has a linear bound (as a function of m) for the time-buckets case and a bound of $O(m\log(n))$ for the heap implementation, assuming all nodes are in the heap at all times. Since the heap is not always in this state for duration of the algorithm over the networks in this figure, it makes sense that the $O(\log(n))$ factor due to the heap should be negligible.

Figure 4.6 shows the savings ratio obtained by dividing the time of running a standard shortest paths algorithm for each departure time by the time of 100 reoptimizations. Over networks of increasing size, the ratios fluctuate but appear to tend to approximately 3 for the time-buckets implementation and the heap with relaxed bounds. The heap with tighter bounds attains a savings ratio just slightly above 1, since it suffers from the computation time to compute the tighter bounds. (We note that, although the heap implementation with tighter bounds does not achieve a good savings ratio, its run time performance is still better than the bucket implementation.)

Figures 4.7 and 4.8 respectively show the running times and savings ratios of the three *SP(k-c)* implementations for FIFO networks as a function of the number of nodes in the network over a constant number of arcs. The graphs have two factors at work – the number of nodes in the graph, which will cause linear increases in the running time, and the "height" of the network. (By height of the network, we mean the maximum number of arcs along any shortest path in the network.) As the network becomes less dense, the height of the network increases, but the running times of the heap algorithm with relaxed time bounds remains essentially constant, and the running time of the time-buckets implementation steadily decreases. These trends can be explained by the following analysis. Anytime a node maintains its minimum arrival time from the previous reoptimization phase, the entire subtree rooted at that node will not need to be explored. In sparse networks of large height, the number of nodes in a subtree may be a significant

portion of the total number of nodes, since there will be only one path to many of the nodes in the network. This is reflected in the graph of savings ratios, whereby the heap with relaxed bounds and the time-buckets implementations both obtain very large savings over the traditional corresponding shortest path problems which start each iteration from scratch. The heap implementation with tighter upper bounds suffers from the extra work required to update the nodes to their lower costs. As expected, in networks with larger numbers of nodes, more computational time is required to compute the tighter lower bounds, since these bounds must be computed for every node in the network.

Figures 4.9 and 4.10 respectively show the running times and savings ratios of the three *SP(k-c)* implementations for FIFO networks as a function of the number of arcs in the network over a constant number of nodes. For a large number of arcs (corresponding to a very dense graph with a small height) the savings ratios are all just slightly above 1, because the subtrees discussed in the analysis of Figures 4.7 and 4.8 contain very few nodes. The running times of all three implementations exhibit linear behavior as a function of the number of arcs in the network, as the running time analysis of the implementations suggest for increasing values of m.

Figures 4.11 and 4.12 respectively show the running times and savings ratios of the three *SP(k-c)* implementations for FIFO networks as a function of the number of reoptimizations performed for a network of constant size. The savings ratios are essentially unaffected by this parameter, and the running times increase linearly for all

three implementations. These results confirm that the running time of the implementations is independent of the time at which the reoptimizations take place, as the theory indicates.

Figures 4.13 and 4.14 respectively show the running times and savings ratio of the bucket implementation of the non-FIFO algorithms for *SP(k-c)* with increasing size of the network (constant density). The running time of the non-FIFO algorithm tends approximately to *O(n*log(height of network))* due to the initialization phase of the algorithm, which depends on the length of the longest path in the network (because the number of labels that must be initialized is directly proportional to this length). The non-FIFO algorithm achieves approximately a 7-fold savings over a standard shortest path algorithm.

Figure 4.15 and 4.16 respectively show the running times and savings ratio of the bucket implementation of non-FIFO algorithms for *SP(k-c)* while increasing the number of nodes, over a constant number of arcs (10,000). As the height of the network increases, the running time of the algorithm increases as well, due to two factors. Firstly, the number of nodes which must be examined increases, and secondly, the number of time-buckets which must be explored by the algorithm increases with increasing network height. We observe that the savings ratio also increases as a function of the number of nodes, due to the computationally intensive initialization procedure of the standard non-FIFO shortest path algorithm.

Figure 4.17 and 4.18 respectively show the running times and savings ratio of the bucket implementation of non-FIFO algorithms for *SP(k-c)* as a function of the number of arcs in the network, while holding the number of nodes constant at 100. The analysis of these trends is similar that of Figures 4.15 and 4.16. For a very dense network, the reoptimization algorithm needs to do very little computational work to find the labels of minimum arrival time, since the network has a very low height. Furthermore, the savings ratio achieved by the non-FIFO reoptimization algorithm is very close to 1 for dense networks, since the standard non-FIFO shortest path algorithm is efficient for such networks.

Figure 4.19 and 4.20 respectively show the running times and savings ratios of the bucket implementation of non-FIFO algorithms for *SP(k-c)* as a function of number of reoptimization phases that are performed. Figure 4.19 shows a linear trends, as the number of reoptimizations performed does not appear to affect the running time of the algorithm per reoptimization phase. The savings ratio increases dramatically with the number of phases due to the computationally intensive initialization procedure of the standard shortest path algorithm for non-FIFO networks.

Figure 4.21 shows the running times of a heap-based implementation of SP(k+c) in a FIFO network, and the running times of computing a standard shortest path over the same number of reoptimization phases. As discussed in Subsection 4.6.1, the standard shortest

path algorithm which starts from scratch for each new departure time runs faster than the reoptimization approach due to the high overhead and small savings of the reoptimization algorithm for later departure times.

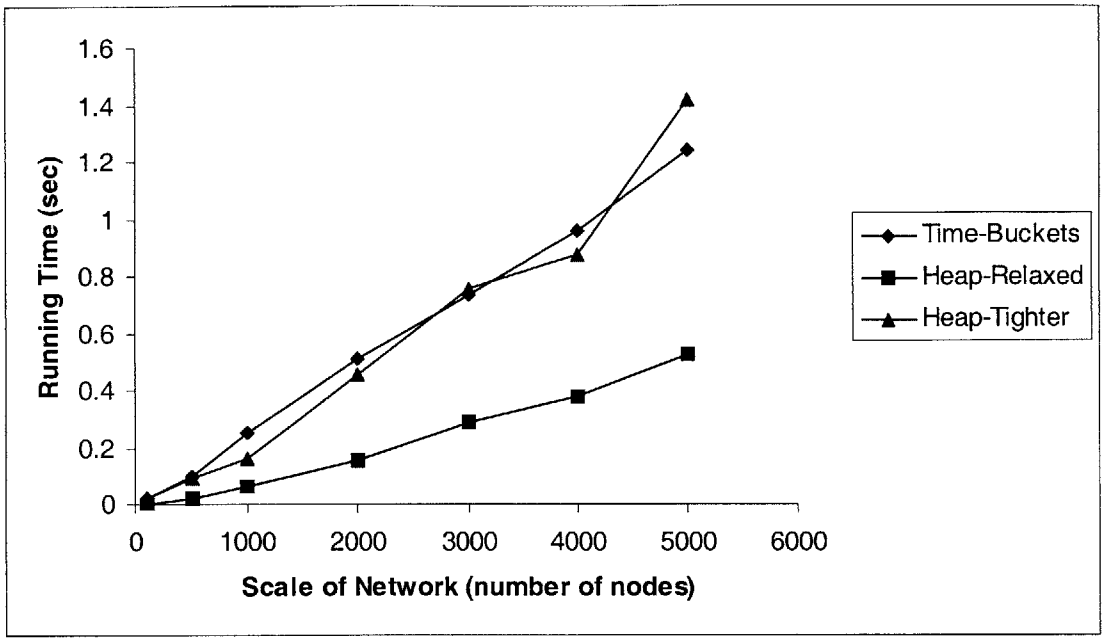


Figure 4.5 Running times of the time-buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of network size. The number of arcs is three times the number of nodes. $d_{ij} \in [1,3]$.

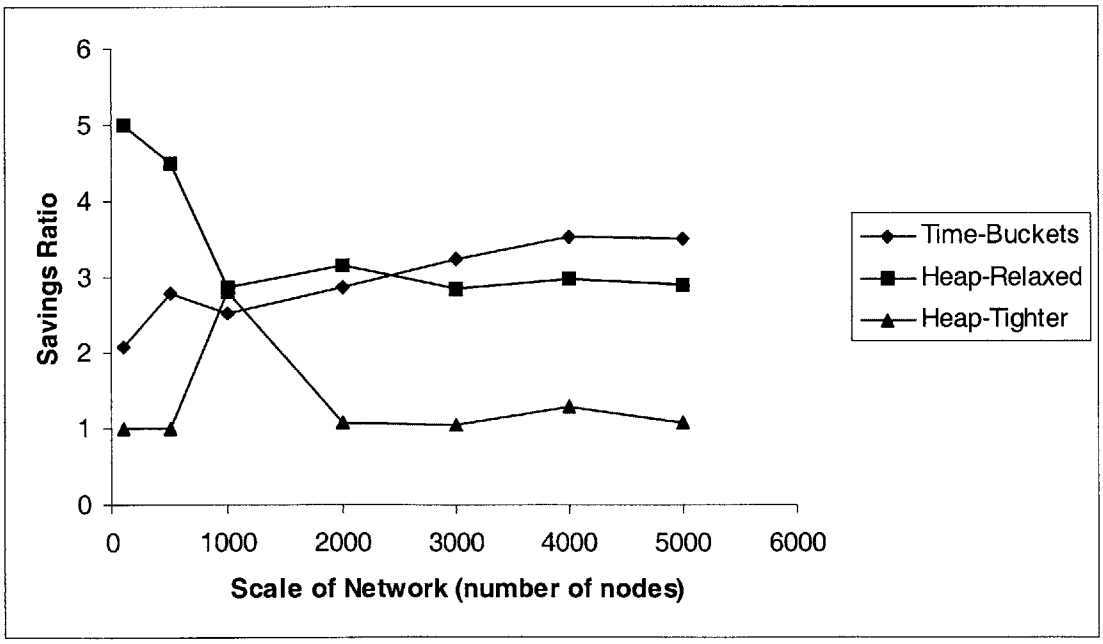


Figure 4.6 Savings ratios for the time-buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of network size. The number of arcs is three times the number of nodes. $d_{ij} \in [1,3]$.

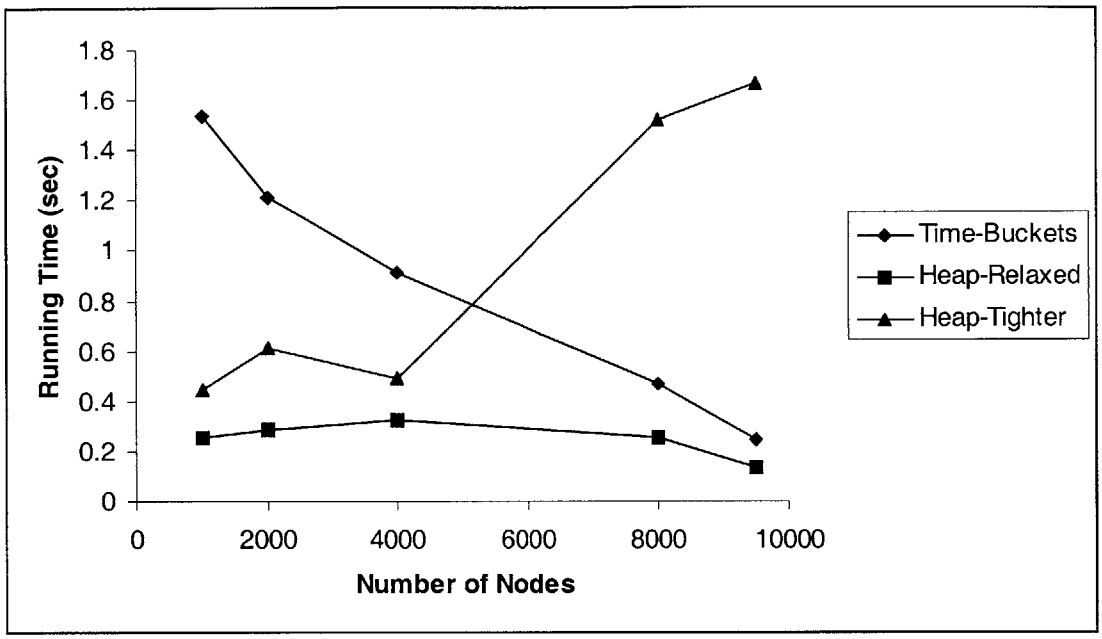


Figure 4.7 Running times of the time-buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of the number of nodes in the network. The number of arcs is 10000. $d_{ij} \in [1,3]$.

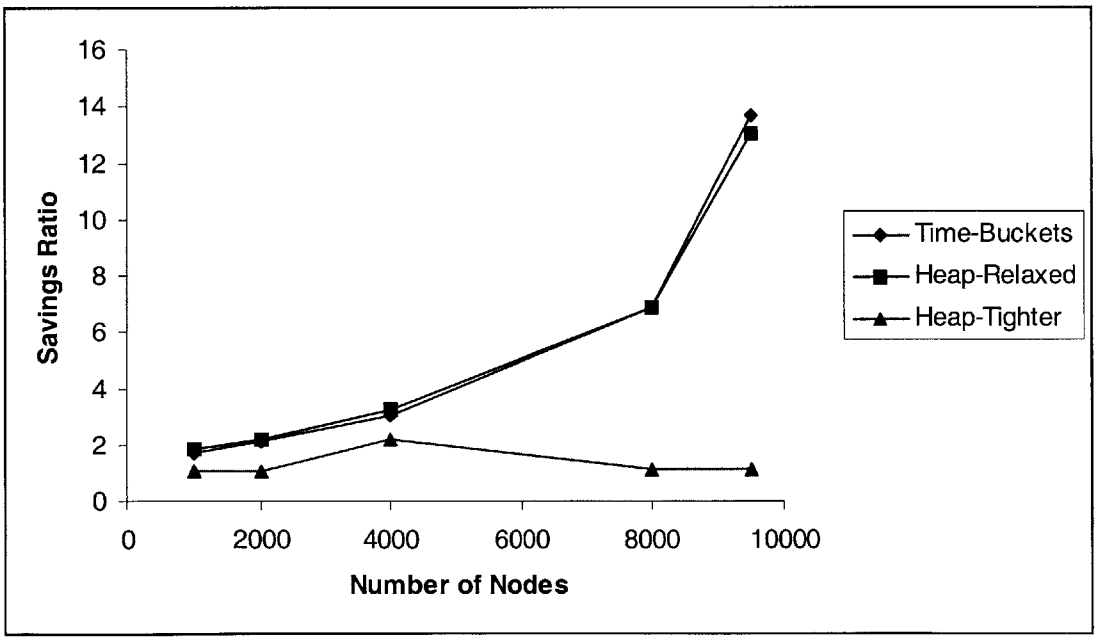


Figure 4.8 Savings ratio of the time-buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of the number of nodes in the network. The number of arcs is 10000. $d_{ij} \in [1,3]$.

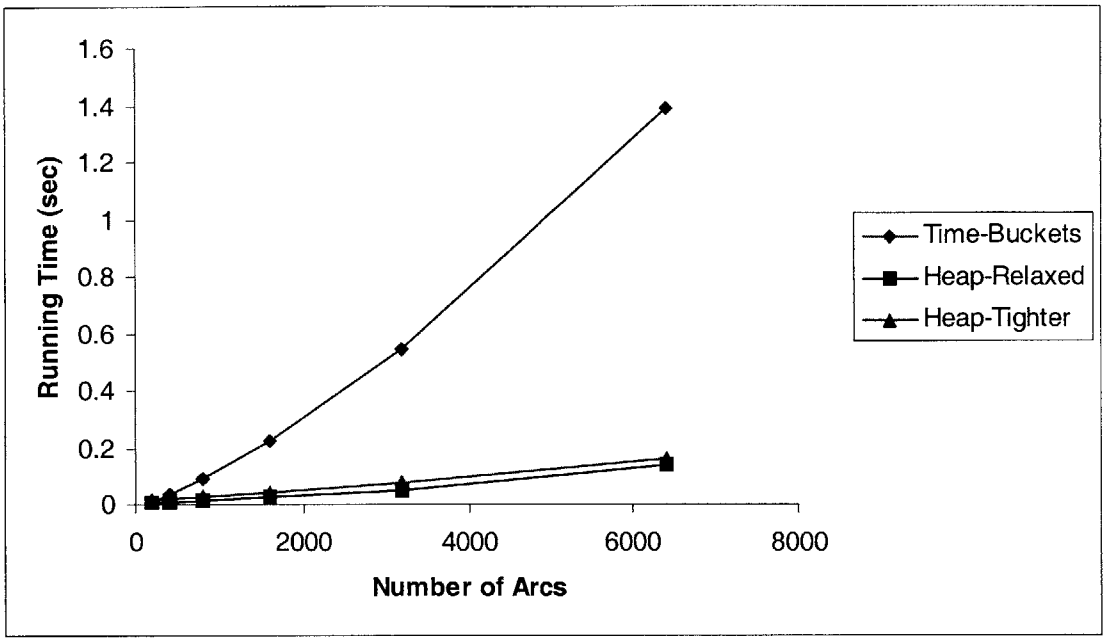


Figure 4.9 Running times of the time-buckets buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of the number of arcs in the network. The number of nodes is 100. $d_{ij} \in [1,3]$.

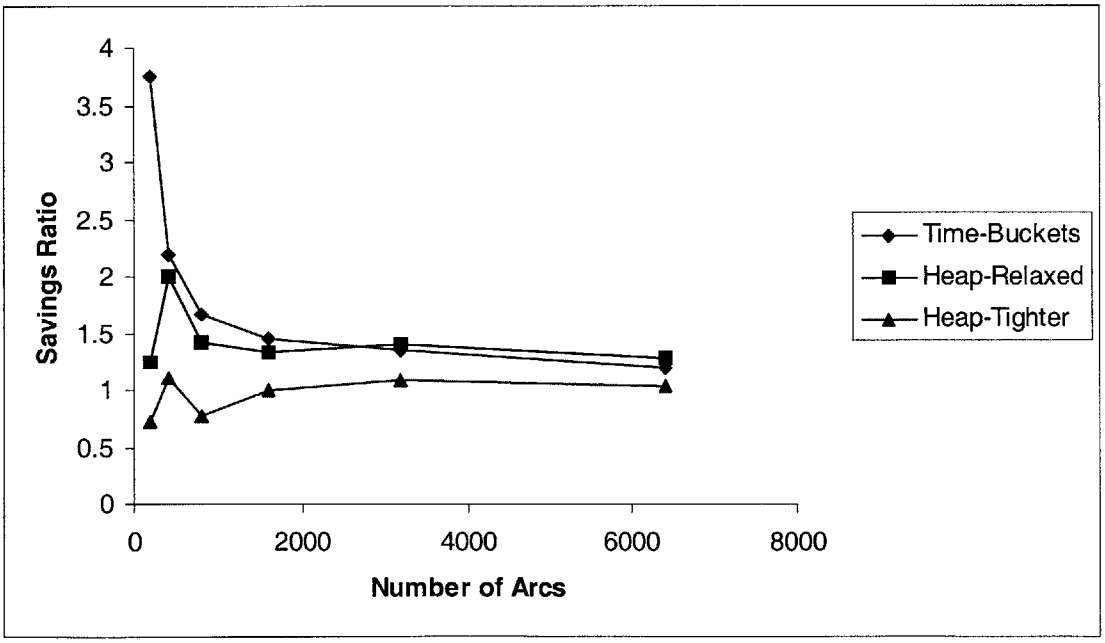


Figure 4.10 Savings ratio of the time-buckets buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of the number of arcs in the network. The number of nodes is 100. $d_{ij} \in [1,3]$.

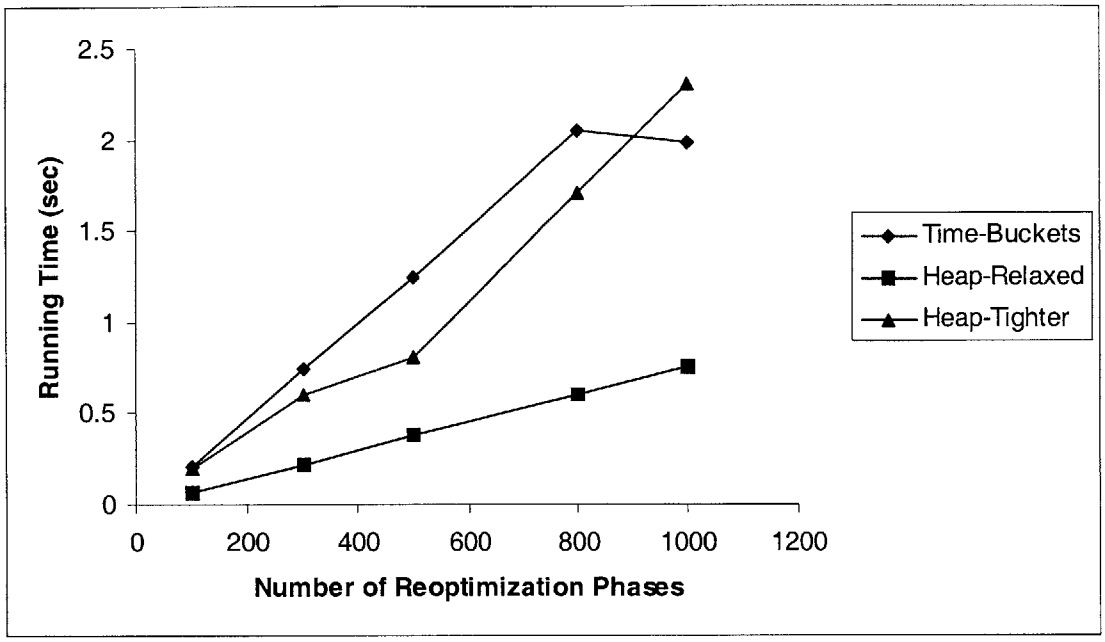


Figure 4.11 Running times of the time-buckets buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of the number of reoptimization phases performed. The number of nodes is 1000 and the number of arcs is 3000. $d_{ij} \in [1,3]$.

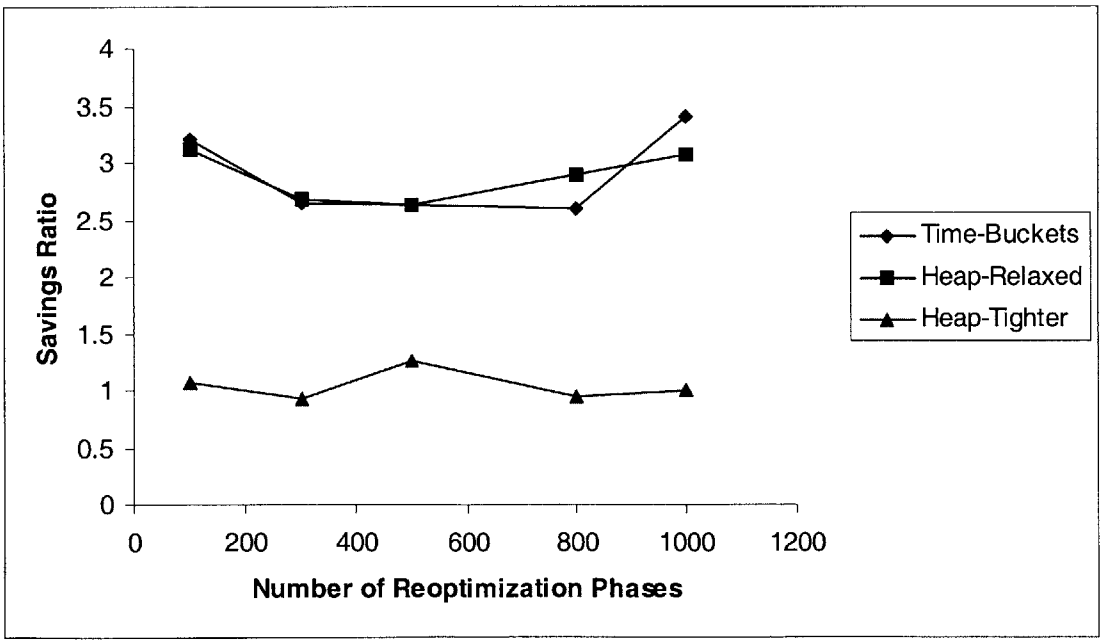


Figure 4.12 Savings ratio of the time-buckets buckets implementation, and the heap implementation with both relaxed and with tight upper bounds as a function of the number of reoptimization phases performed. The number of nodes is 1000 and the number of arcs is 3000. $d_{ij} \in [1,3]$.

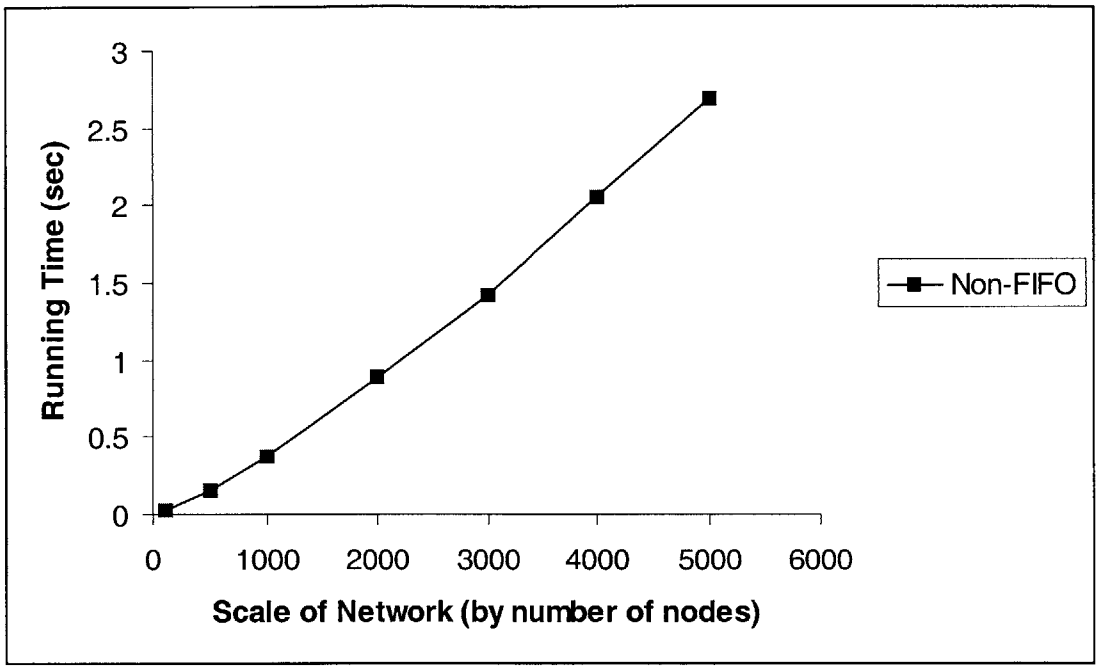


Figure 4.13 Running times of the time-buckets implementation of the non-FIFO reoptimization algorithm as a function of network size. The number of arcs is three times the number of nodes. $d_{ij} \in [1,3]$.

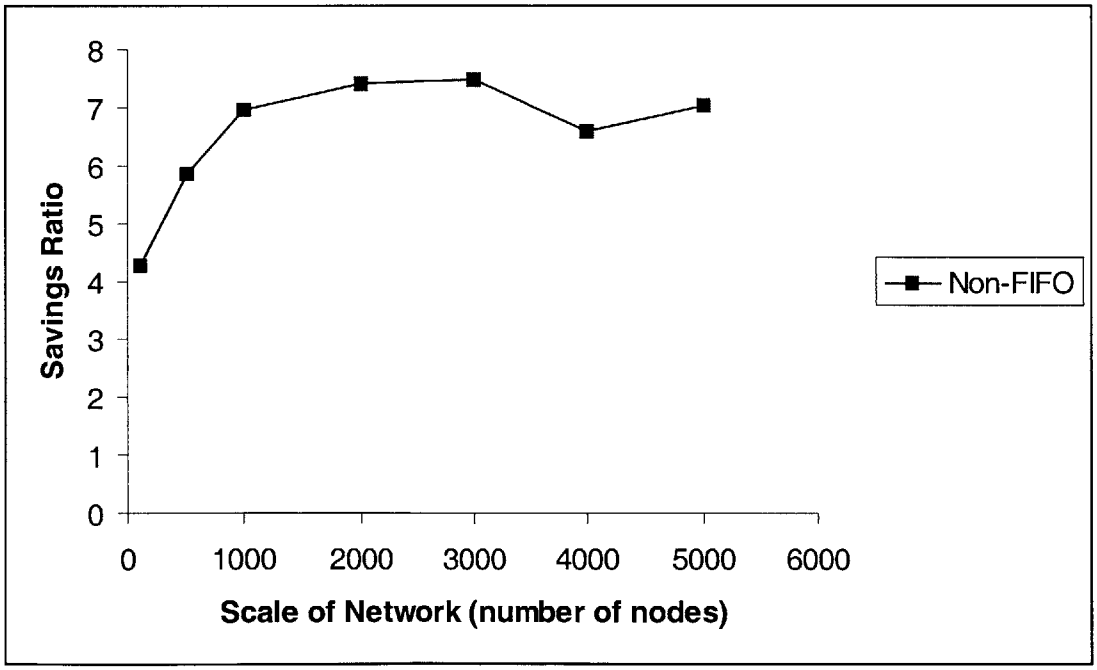


Figure 4.14 Savings ratio of the time-buckets implementation of the non-FIFO reoptimization algorithm, as a function of network size. The number of arcs is three times the number of nodes. $d_{ij} \in [1,3]$.

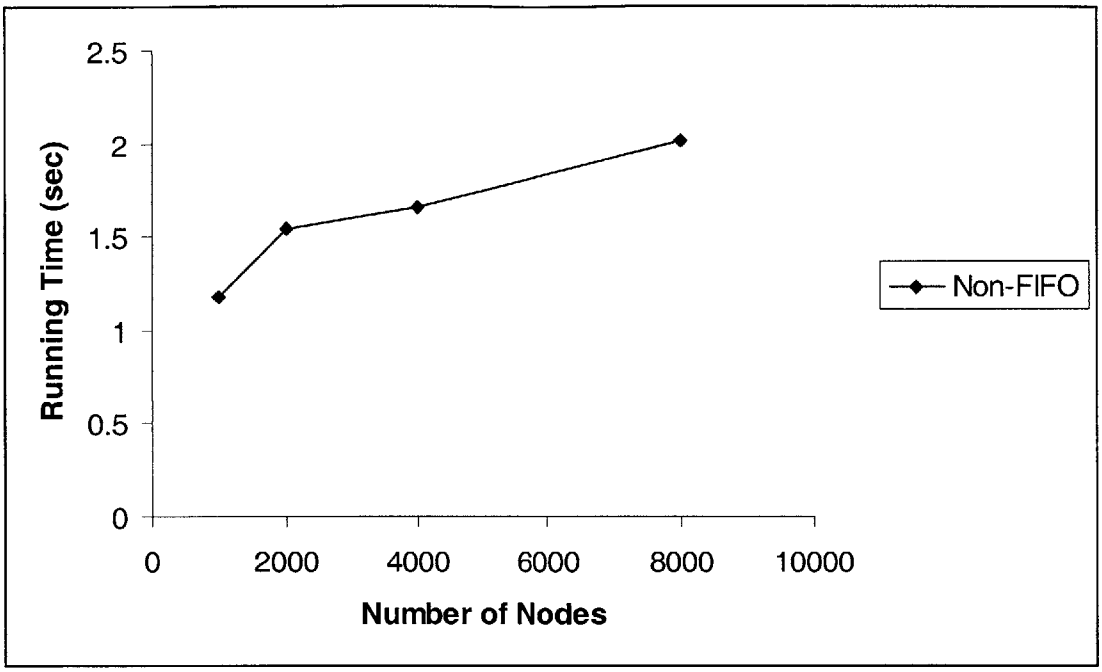


Figure 4.15 Running times of the time-buckets implementation of the non-FIFO reoptimization algorithm as a function of the number of nodes in the network. The number of arcs is 10000. $d_{ij} \in [1,3]$.

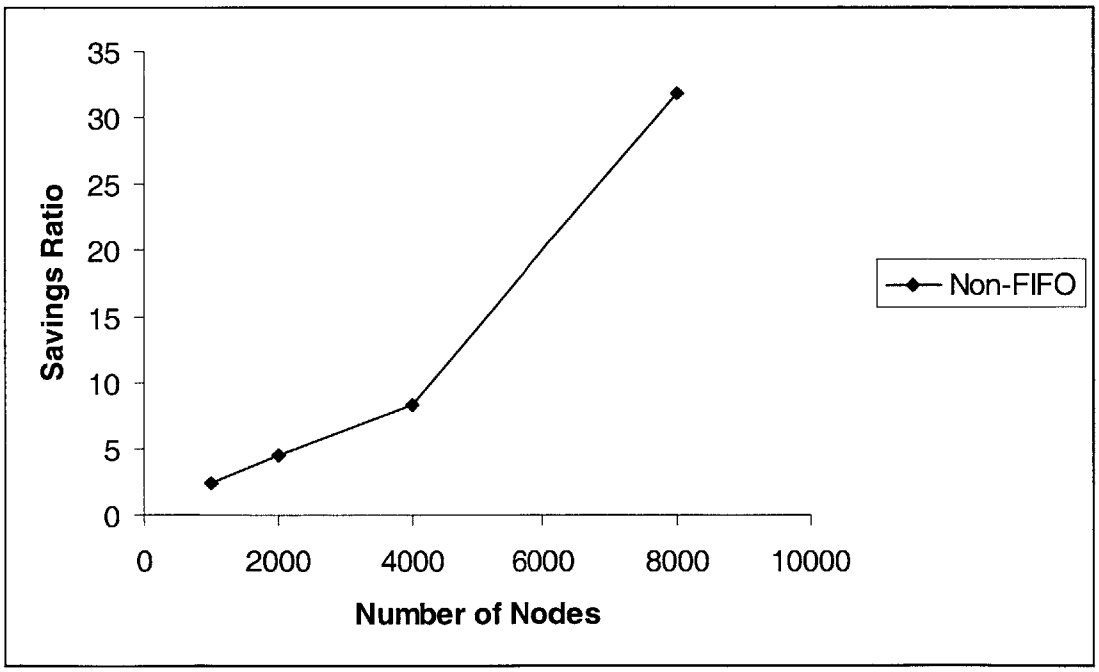


Figure 4.16 Savings ratio of the time-buckets implementations of the non-FIFO reoptimization algorithm as a function of the number of nodes in the network. The number of arcs is 10000. $d_{ij} \in [1,3]$.

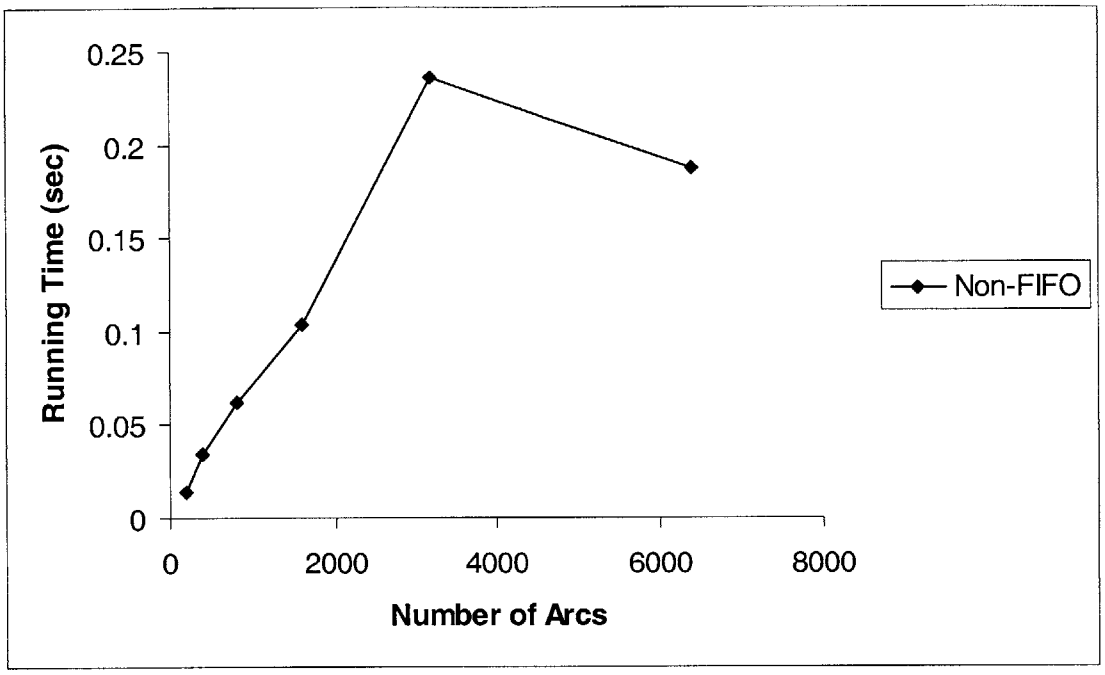


Figure 4.17 Running times of the time-buckets implementation of the non-FIFO reoptimization algorithm as a function of the number of arcs in the network. The number of nodes is 100. $d_{ij} \in [1,3]$.

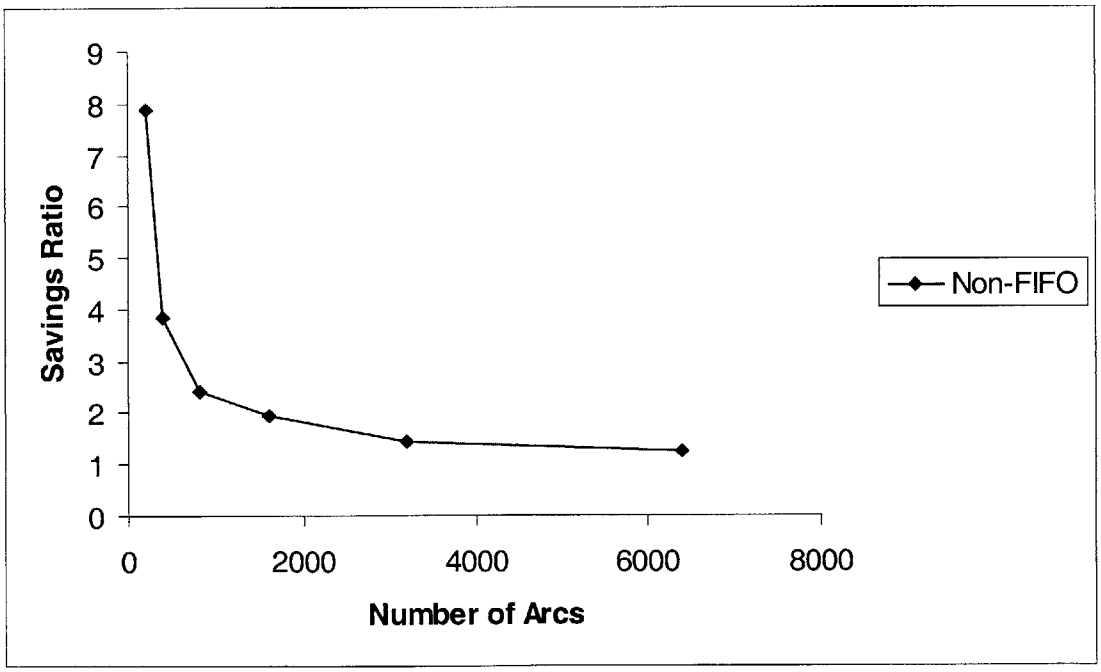


Figure 4.18 Savings ratio of the time-buckets implementations of the non-FIFO reoptimization algorithm as a function of the number of nodes in the network. The number of arcs is 10000. $d_{ij} \in [1,3]$.

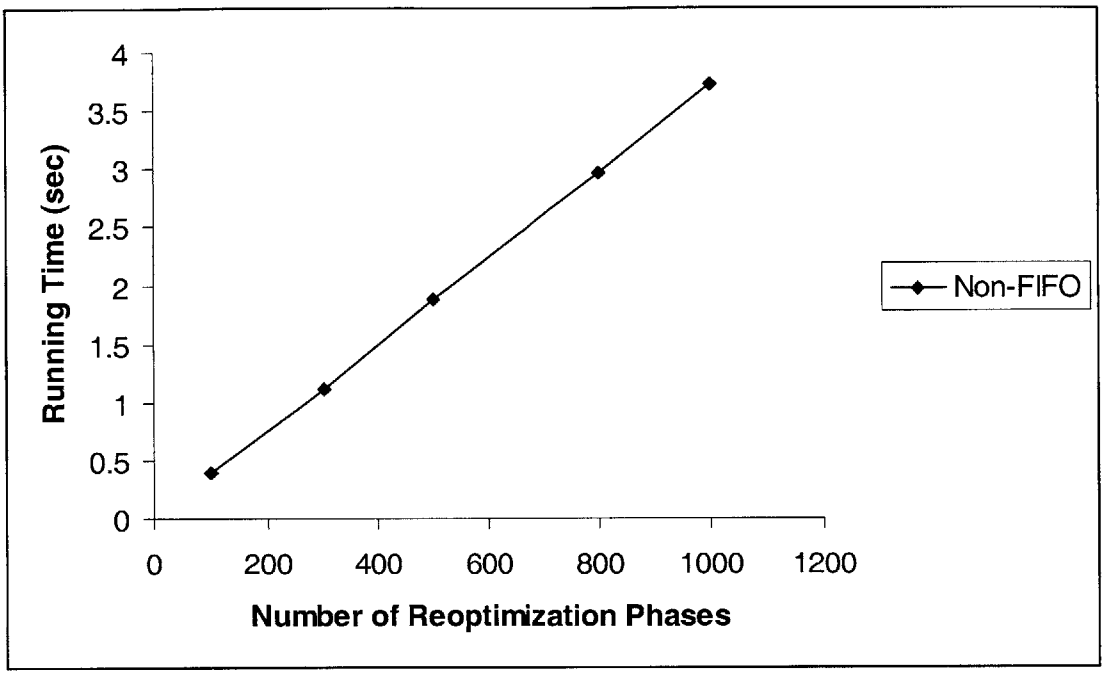


Figure 4.19 Running times of the time-buckets implementation of the non-FIFO reoptimization algorithm as a function of the number of reoptimization phases performed. The number of nodes is 1000 and the number of arcs is 3000. $d_{ij} \in [1,3]$.

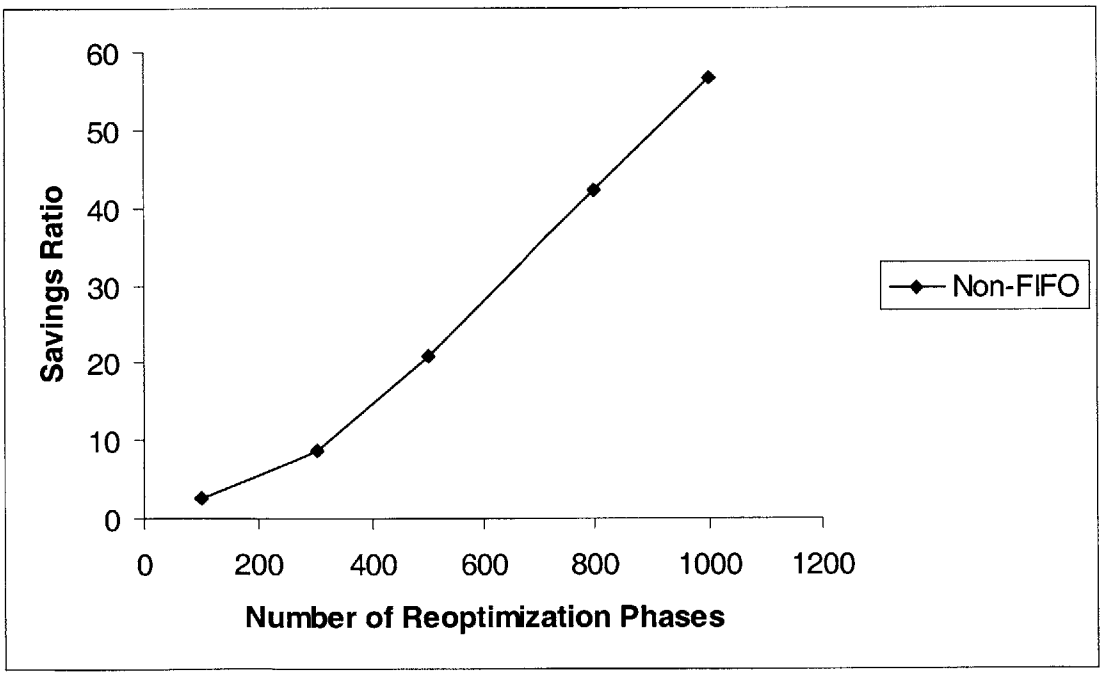


Figure 4.20 Savings ratio for the time-buckets implementation of the non-FIFO reoptimization algorithm as a function of the number of reoptimization phases performed. The number of nodes is 1000 and the number of arcs is 3000. $d_{ij} \in [1,3]$.

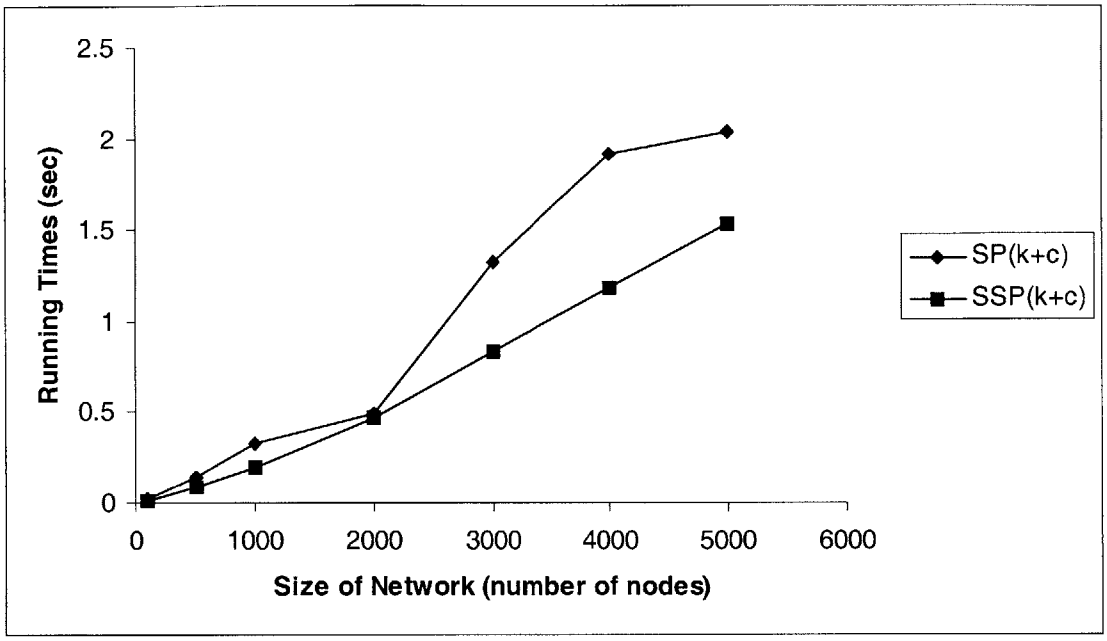


Figure 4.21 Running times of a heap-based implementation of the Reoptimization Algorithm for SP(k+c) and running times of a standard shortest path algorithm over the same interval. The number of nodes is 100 and the number of arcs is 300. $d_{ij} \in [1,3]$.

Chapter 5

Conclusion

In this chapter, we summarize the results of Chapters 4 and 5, and we state what we have contributed to the field of network optimization, specifically in the context of transportation systems. We also identify future research areas that arise from the developments in this thesis.

5.1 Summary of Contributions

We have examined in detail the shortest path problem in networks with time windows and linear waiting costs, and the shortest path reoptimization problem in time-dependent networks. In the following subsections, we summarize the research conducted and the contributions made for each of these problem-types.

5.1.1 The Shortest Path Problem With Time Windows and Linear Waiting Costs

We studied the SPWC from a point of view of the labels in the time-space network in order to understand the dominance of labels in both the static and dynamic case. We developed a generic algorithm to solve this problem for both network types, with at least

six possible implementation strategies for the case of static networks. We analyzed these strategies theoretically, and we implemented the most promising ones in C++, along with an implementation of the algorithm for dynamic networks. We obtained running time results for these implementations based on randomly generated networks of substantial size.

5.1.2 The Shortest Paths Reoptimization Problem

We formulated a new reoptimization problem that does not seem to have been discussed in the literature. We examined the reoptimization problem in FIFO and non-FIFO networks, for earlier and later departure times from the source. We discussed two implementations of the basic reoptimization algorithm, one using heaps and another using time-buckets. We implemented several versions of the algorithm to handle different reoptimization problems, and we showed how these algorithms may be extended to address other problem variants. On large, randomly generated networks, we achieved a savings ratio of roughly 3 on average by using the reoptimization techniques discussed in this thesis to find shortest paths over multiple starting times, as opposed to computing the shortest paths by a repetitive application of the standard algorithm for each starting time.

5.2 Future Research Directions

There is much research to be done in both the field of time windows and the field of reoptimization. In the following two sub-sections, we suggest some of these possible directions.

5.2.1 Time Window Algorithms

The algorithms in this thesis the network data is discrete and integral. However, there has been recent work in the field of continuous-time network algorithms. In light of this, one might consider a continuous-time implementation of the SPWC Algorithm. We believe that such an implementation would be similar to its discrete-time counterpart presented in Chapter 3. There may, however, be implementation subtleties unbeknownst to us, and there may exist diverse strategies for checking dominance in the continuous-time context.

Under the current SPWC Algorithm, in the case of dynamic arc travel times, every reachable node in the time-space network is examined. This suggests two directions of future research. The first direction is to note that since every label reachable along some path from the source will be explored, there seems to be no reason to limit network data as we have done in this thesis. For example, the waiting cost w could be a function of each node and time, $w(i, t)$. Similarly, arc costs need not be time-independent. Although theory suggests that such changes would not alter the running time of these algorithms in

the worst case, experimental analysis based on practical implementations would have to be constructed to test these extensions.

The second direction of research related to the SPWC Algorithm with dynamic arc travel times is to investigate the conditions and assumptions under which one can improve the running time of this algorithm. That is, do there exist some conditions on the arc costs, waiting costs, arc travel times, and time windows under which it would be possible to significantly reduce the running time of the SPWC Algorithm for dynamic networks so that it achieves the same average-case running time as the algorithm for the static case? Discovering such conditions, if they exist, could potentially enable faster implementations of the algorithm for dynamic networks, which serve as better models for practical network flow applications than their static counterparts.

5.2.2 Reoptimization Algorithms

Reoptimization for different departure times is a new field of study, and as such, there are many areas of research that warrant further study. We shall suggest a few of these areas in this subsection.

Firstly, as the reoptimization problem for different departure times is relatively new, the algorithms developed in this thesis may not have optimal running times. It is likely that as this field matures, other types of algorithms for the reoptimization of a shortest path tree for various departure times will be developed. Particularly in the case of *SP(k+c)*,

there is a need to find better algorithms, as the one developed in Chapter 4 of this thesis was shown to have a worse running time in practice than re-running a standard shortest path algorithm for each desired departure time. It might be possible to apply the techniques used in algorithm IOT-C in [3] to the reoptimization problem in order to develop more efficient solution algorithms.

Secondly, we may wish to extend the idea of reoptimization for varying departure times past the one-to-all shortest path problem as follows. We can continue the study of the reoptimization of shortest paths by investigating the all-to-one minimum arrival time problem for varying departure times. We can study the reoptimization of minimum cost paths by investigating the one-to-all and all-to-one minimum cost path problem for varying departure times. Finally we can study the reoptimization of other network flow problems, for example, by investigating the maximum-flow problem for different departure times.

Finally, just as in the case of the SPWC Algorithm, there exists the opportunity to extend these algorithms to the continuous-time domain. Algorithms IOT-C and DOT-C in [4] are very closely related to the reoptimization problem for different departure times, and they could potentially be modified to solve reoptimization problems in networks with continuous-time data.

Appendix A

In this section, we show how to implement the reoptimization variants for FIFO networks listed in Section 4.1.

Variant 1: Compute *SP(k')* for one value of *k'*. This is the "standard" variant. The pseudocode is provided in Figure 4.1, where we set $k\text{-}c = k'$.

Variant 2: Compute the minimum arrival time paths for all departure times from *k-1* down to *0*. Run Variant 1 one time to find *SP(k-1)*. Then run steps 2-3 of Variant 1 for all times from *k-2* down to *0*.

Variant 3: Compute the minimum arrival time paths for a given interval of departure times. For each (non-overlapping) interval *[k', k'']*, such that $k'' < k' < k$, run Variant 1 one time to find *SP(k')*. Then run steps 2-3 of Variant 1 for all times from *k'-1* down to *k''*.

Variant 4: Compute the minimum arrival time paths regardless of departure time for some/all $k' < k$. By Lemma 4.6, the minimum arrival time function is non-decreasing as a function of departure time from the source. Therefore, the minimum arrival time for a range of departure times can be found simply by computing Variant 1 for the smallest value of k' in the desired range of departure times.

Variant 5: Compute the shortest travel time paths regardless of departure time for some/all $k' < k$. Run step 1 of the pseudocode below once, to initialize the data for departure time *k*. For each desired departure time $k' < k$, run steps 2 – 4 of the pseudocode below.

Step 1: Initialize

```
//ShortestPaths(n,m,k) sets pred(i) and mintime(i) ∀ i
//The method arg_mintime returns the node-time pair
//corresponding to the candidate label of minimum
//arrival time.

SP(k) = ShortestPaths(n,m,k)
SP(k-c) = SP(k)
Mintraveltime(i) = mintime(i) - k ∀ i
```

Step 2: Reoptimize for time k-c < k

```
SP(k-c) = SP(k-c) \ { source }
Candidates = { (source, k-c) }
```

Step 3: Main Loop

```
while Candidates != Ø do
   (i, t) = arg_mintime( Candidates )
   SP(k-c) = (SP(k-c) \ {i, mintime(i)}) ⋃ {(i, t)}
   for all successor j of node i do
      a_j = t + d_ij(t)
      if a_j < mintime(j)
            mintime(j) = a_j
            Candidates = Candidates ⋃ { (j, a_j) }
```

Step 4: Compute Minimum Travel Times

```
for all nodes i do
   if mintime(i) - (k-c) < mintraveltime(i)
      mintraveltime(i) = mintime(i) - (k-c)
```

Appendix B

Some technical terms that are used throughout the literature have slightly different definitions, depending on the context in which they are used. Thus, we include a glossary of terms that are used in this thesis and in related literature, such that there is no confusion due to multiple definitions.

For the Time Windows Algorithms of Chapter 3:

Active – Labels that should be explored by the algorithm are known as active labels.

Candidate Label – An active label.

Clean – A label that is not active is said to be clean.

Dirty – A label that is active is said to be dirty.

Examine – A label in the time-space network is said to be examined by SPWC algorithm when that label is arrived at during a chronological scan of all of the labels. At this point, it is determined if the label should be explored, and if it should be placed in the set of permanent labels or discarded.

Explore – When a label $L_i = (T_i, C_i)$ is explored, the node-time pairs in the forward star of node i at time T_i are updated.

Label – a node-time pair for a node i in the time-space network with a cost C_i corresponding to the cost of some feasible path that arrives at node i time T_i. A label for node i is denoted as (T_i, C_i).

Treat – A label is treated when it is examined.

Update – A label is updated when a path of lower cost is found to it. The cost associated with the label is changed to the new, lower cost, and predecessor information is changed as well.

For the Reoptimization Algorithms of Chapter 4 (only definitions that differ in the case of the reoptimization algorithms are presented here):

Label – a node-time pair for a node i in the time-space network corresponding to some feasible path that arrives at node i at time t. A label for node i is denoted as (i, t).

Examine – Synonymous with the definition of explore given above.

Treat – Synonymous with the definition of explore given above.

References

[1] I. Chabini (1998). Discrete dynamic shortest path problems in transportation applications: complexity and algorithms with optimal run time. *Transportation Research Record*, 1645, pp. 170-175.

[2] I. Chabini and B. Dean (1998). Shortest path problems in discrete-time dynamic networks: complexity, algorithms and implementations. Internal Report.

[3] I. Chabini and V. Yadappanavar (2000). Advances in discrete-time dynamic data representation with communication and computation transportation applications. Internal Report.

[4] B. Dean (1999). Continuous-time dynamic shortest path algorithms. Master of Engineering Thesis, Massachusetts Institute of Technology (Supervisor: Professor Ismail Chabini).

[5] G. Desaulniers and D. Villeneuve (2000). The shortest path problem with time windows and linear waiting costs. *Transportation Science*, pp. 312-319.

[6] M. Desrochers (1986). "La fabrication d'horaires de travail pour les conducteurs d'autobus par une méthode de génération de colonnes." Doctoral Dissertation, Université de Montréal, Montreal.

[7] M. Desrochers and F. Soumis (1988). A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR* 26 (3), pp. 191-212.

[8] J. Desrosiers, P. Pelletier, and F. Soumis (1983). "Plus court chemin avec contraintes d'horaires." *R.A.I.R.O* 17 (4), pp. 357-377.

[9] R. Dial (1969). Algorithm 360: shortest path forest with topological ordering. *Communications of the ACM* 12, pp. 632-633.

[10] E. W. Dijkstra (1959). A note on two problems in connexion with graphs. *Numerishe Matematik* 1, pp. 269-271.

[11] R. Dionne (1978). "Étude et Extension d'un Algorithme de Murchland." *INFOR* 16, pp. 132-146.

[12] M. Florian, S. Nguyen, and S. Pallottino (1981). A dual simplex algorithm for finding all shortest paths. *Networks* 11, pp. 367-378.

[13] S. Fujishige (1981). A note on the problem of updating shortest paths. *Networks* 11, pp. 317-319.

[14] G. Gallo (1980). Reoptimization procedures in shortest path problems. *Rivista di Matematica per le Scienze Economiche e Sociali* 3, pp. 3-13.

[15] G. Gallo and S. Pallottino (1982). A new algorithm to find the shortest paths between all pairs of nodes. *Discrete Applied Mathematics* 4, pp. 23-35.

[16] I. Ioachim, S. Gélinas, J. Desrosiers, and F. Soumis. A dynamic programming algorithm for the shortest path problem with time windows and linear node costs. *Networks* 31, pp. 193-204.

[17] J. D. Murchland (1970). A fixed matrix method for all shortest distances in a directed graph and for the inverse problem. Doctoral Dissertation, University of Karlsruhe, Germany.

[18] S. Nguyen, S. Pallottino, and M. G. Scutella (1999). A new dual algorithm for shortest path reoptimization. Technical Report: TR-99-14, Università di Pisa.

[19] S. Pallottino and M. Scutella (1997). Shortest path algorithms in transportation models: classical and innovative aspects. Technical Report: TR-97-06, Università di Pisa.