On computing the Pareto optimal solution set in a large scale dynamic network

by

Raoul-Sam Daruwala

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Department of Computer Science

New York University

September 2002

Bhubaneswar Mishra

For Sam

Acknowledgments

I'd like to thank my family, friends and colleagues who have helped me along the way.

In particular Gideon Berger for many years of interesting collaboration and adventure, and Bud Mishra for giving me the opportunity to continue working on interesting problems. I'd also like to thank Robert Paige for teaching me to love the art of computing and Richard Cole and the Theory exam for the rigorous training in how to design algorithms.

There have been many far too many other people who have helped me and provided support over the last ten years, I'm sure any list will be incomplete but here goes, thank you to Deepak Goyal, Arash Baratloo, David Tanzer, Jay Sachs, Roy John, Leslie Prichep, Davi Geiger, Ali Argyle, Chris Tignor, Amos Elliston, Milton Cohen, Andy Lottman, and Sudheendra Hebbagilu.

I'd also like to thank some of my closest friends Unmesh, Hemalee, Ajantha, Vince, Pria, Firoz, Gyaan (your turn), the Korbels, Pete, Beth, Noah, Jeff and J-Lila.

Auralice Graft thank you for getting me started and keeping me going.

A very special thank you to Valerie Hallier for typesetting the diagrams in this thesis and to my two guardian angels Anina Karmen and Rosemary Amico.

Abstract

Let $G = (V, E)$ be a graph with time-dependent edges where the cost of a path p through the graph is determined by a vector functions $F(p) = [f_1(p), f_2(p), \ldots, f_n(p)]^T$, where $f_1, f_2, \ldots, f_n$ are independent objective functions. Where $n > 1$ there is no clear idea of what a "best" solution is, instead we turn to the idea of Pareto-optimality to define the efficiency of a path. Given the set of paths P through the network, a path p' is *Pareto-optimal* if for every $p \in P$, $\wedge_{i \in [1,n]}(f_i(p) \geq f_i(p'))$.

The problem of planning itineraries on a transportation system involves computing the set of optimal paths through a time-dependent network where the cost of a path is determined by more than one, possibly non-linear and non-additive, cost function. This thesis introduces an algorithmic toolkit for finding the set of Pareto-optimal paths in time-dependent networks in the presence of multiple objective functions.

Multi-criteria path optimization problems are known to be NP-Hard, however, by exploiting geometric and periodic properties of the dynamic graphs that model transit networks we show that it is possible to compute the Pareto-optimal solutions sets rapidly without using heuristics. We show that we can solve the itinerary problem in the presence of response time constraints for a large scale graph.

Contents

Dedication ii

Acknowledgments iii

Abstract iv

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 A motivating example 2

1.2 Problem Statement and Our Approach 5

1.3 Thesis outline 7

2 Shortest Path Problems in Transportation Models 9

2.1 The Shortest Path Tree Problem 10

2.1.1 Primal algorithms for SPT. 11

2.1.2 SPT_S Algorithms 15

2.1.3 SPT_L Algorithms 16
2.2 Dynamic Shortest Path Problems 26
2.2.1 Space-Time Networks 28
2.2.2 The Minimum Cost Dynamic Path Problem. 32
2.2.3 Train Graphs 36
2.2.4 Timetables and Time Dependent Networks 37
2.2.5 Time Dependent Intermodal Networks 40
2.3 Shortest Path Problems in the Presence of Response Time Constraints 43
2.4 Nonadditive Cost functions in Shortest Path Problems . . . 47
2.4.1 Nonadditive cost functions. 48
2.4.2 The Inherited Constraint Algorithm 49
2.5 Multi-criteria Optimization in Graph Problems 56
2.5.1 Pareto Optimal Solutions 57
2.5.2 The Bicriterion Shortest Path Problem 59

3 The Itinerary Problem **64**
3.1 The Pareto Itinerary Problem 68

4 Computing Pareto Paths **76**
4.1 Building the Dynamic Network from Transit Data 77
4.2 Stop selection and prioritization 85
4.3 A naïve algorithm to compute itineraries 88
4.4 Special cases of the ParetoPath algorithm 93

4.5 Transfer minimization 95
4.5.1 Creating the set of direct edges 96
4.5.2 Using direct edges to minimize transfers 96

5 A Faster way to compute Pareto paths 100

5.1 Similar Edge Sets 102
5.1.1 An equivalence relation for similarity 103
5.1.2 Building similar edge sets 106
5.1.3 Using similarity in the ParetoPath algorithm 108
5.2 The All Times Single Source Pareto Path Problem 111
5.3 Generating itineraries for large server loads 116
5.3.1 The two tier itinerary generation strategy 116
5.3.2 Generating the second tier graph 119
5.3.3 Generating an online response 120
5.3.4 Performance of offline computation 121
5.4 Speeding up online response using Fare Region restriction . . 122
5.5 Other Issues 126
5.5.1 Reducing disk activity 126
5.5.2 Reducing the cost of memory management 128

6 Conclusion 131

6.1 Future directions 131
6.2 Final Remarks 134

Bibliography 138

List of Figures

2.1 The GenericSPT Algorithm. 13

2.2 A simple Update function. 14

2.3 The SPT_L Dissasembling Q.Select function 22

2.4 The SPT_L Dissasembling Q.Enqueue function 22

2.5 SPT_L Topological Ordering Q.Select function 24

2.6 SPT_L Topological Ordering DFS_Visit 25

2.7 SPT_L Topological Ordering Q.Enqueue function 25

2.8 A Dynamic Network 28

2.9 A Space-Time Network 30

2.10 A FIFO graph 31

2.11 Cost Consistency 31

2.12 The non-redundant portion of the space-time network 32

2.13 Chrono-SPT 33

2.14 Chrono-SPT (FIFO and CC) 35

2.15 A train graph. 37

2.16 Train graph with transfer edges. 38

2.17 The MinimumArrivalTime algorithm 40
2.18 A simple network with Nonadditive paths. 48
2.19 Initial Solution for the ICA 53
2.20 Updating the constraint in the ICA 54
2.21 ICA Iterations 55
2.22 ICA Termination 56

3.1 Identical edge sets lead to exponential complexity. 74

4.1 An example of a fare table entry. 80
4.2 Fare Regions 81
4.3 An example of an intersection 82
4.4 Graph reductions for stations with special behavior. 83
4.5 Stop selection problem 87
4.6 Stop selection fix 87
4.7 Duplicate edges cause worst case behavior 91
4.8 Transfer Problem 92
4.9 The MakeDirectEdgeSets algorithm 97
4.10 Augmenting the graph with direct edges. 98
4.11 Extend for similar edge sets. 99

5.1 The MakeSimilarEdgeSets algorithm 106
5.2 The TransportCompare operator 107
5.3 Reduced multi-graph for similar sets. 109
5.4 Extend for similar edge sets. 109

5.5 Reducing G to an all times problem. 113
5.6 The AllTimesReduction 113
5.7 Domination in the All Times Pareto Path problem 114
5.8 The AllTimesParetoPath algorithm 115
5.9 Two tier itinerary generation 118
5.10 The guided ParetoPath algorithm 124

List of Tables

1.1 Going from New York to Philadelphia 3

3.1 Traveling from New York City to Philadelphia. 66

5.1 Peformance of the the ParetoPath variants. 127

Chapter 1

Introduction

If the technological advancements that ushered in the "information age" led only to the creation of vast libraries of data and the means to distribute them efficiently our lives would not have changed as dramatically as they have in recent years. Perhaps the most significant advance that we have made in interacting with our new technology has been its impact on the way we make decisions. Our digital assistants have become tools for creating precise, highly refined information on which we base our actions.

Decision making, whether human or automated, is often a matter of answering a question that starts with the words "What is the best...". It is in the evaluation of "what is best" over large data sets that computers have an advantage over us and have made their biggest impact on our lives. Optimization problems, those problems that focus on finding the *best* are among the most studied problems in computer science and, the most studied among the optimization problems are shortest path problems. Since the

end of the 1950s there have been more than two thousand publications on shortest path problems in many application areas and the literature shows many interesting approaches to solving this class of problems. In shortest path problems, the notion of *best* is limited to one criterion. Evaluating what is *best* on multiple criteria simultaneously involves a class of optimization problems which is much larger and harder to solve.

The World Wide Web, arguably the largest collection of information that we have assembled, has many sites that are prime examples of how we use decision making services in our everyday lives. The Google search engine with over a billion pages in its database answers the question, "What are the best web pages if I'm looking for the following topics?" by producing a small subset of the pages on the web that match our search criteria. The MapQuest service is used to search for the best routes from one address to another. A large database of street information forms the knowledge base on which the MapQuest route optimization algorithms operate. Routing problems are in essence shortest path problems. In the MapQuest model, intersections become vertices in a graph, roads become edges, weighted by their lengths.

1.1 A motivating example

We have chosen to create a direction service, not unlike MapQuest's, for transit itinerary planning. Unlike the MapQuest service which is essentially a single criteria optimization, the length of the journey, transit planning in-

Option	Carrier	Route	Time	Fare
1	Amtrak	New York → Philadelphia	1h 50 m	$85
2	Commuter Rail	New York → Trenton → Philadelphia	2 h 12 m	$14
3	Bus	New York → Philadelphia	2h 20 m	$20

Table 1.1: Three options for traveling from New York to Philadelphia by public transport.

volves evaluating the "best" routes over many criteria simultaneously. Consider optimizing when the criteria are the cost of the journey (fare) and the duration of the journey (time) of the journey. The idea of "best", in this problem, is less intuitive than for a single criteria problem. To give an example of what is "best" in a multi-criteria setting consider the following example.

A passenger trying to go from New York to Philadelphia using only public transportation. There are three options available, the first, to take Amtrak, the second to use the commuter rail services that serve Philadelphia and New York, both of which happen to intersect at Trenton, and the third option to take a bus from one city to another. The times of journey and their costs are shown in figure 1.1

Clearly the option 1 is the fastest, and if speed were our single criterion for optimization, option 1 would would be our only choice. Option 2 is the cheapest and is therefore the best solution if spending as little money as possible were the only criterion for deciding we were "What is the *best* way to get from New York to Philadelphia if I want to save as much money as

possible?".

If we ask the question "What is *best* way to get from New York to Philadelphia if I want to get their quickly without spending too much money?" we now have no clear *best* choice. In fact, we have no immediately intuitive meaning for what *best* is when dealing with more than one optimization criterion. We turn to the work of Pareto, the 19th century welfare economist who first introduced the idea of multi-criteria optimization, for a definition. The idea of *best* according to Pareto is, simply put, *not worse than any other*. To understand what this means consider option 3. On both our criteria option 3 is worse than both option 1 and option 2, but both options 1 and 2 are no worse than each other on both criteria *simultaneously*. According to Pareto, both options 1 and 2 are *best* solutions. Calculating the best solution in the multi-criteria problem involves finding the Pareto-optimal set of non-worse solutions. This gives us some idea of why multi-criteria problems are typically harder for computers to solve. Solutions to multi-criteria problems are seldom single values, in fact they are sets of solutions, sets, that can grow very large.

Now that we know what the Pareto set is it becomes clear when we make decisions based on many factors we balance the importance of the various criteria that we are considering. Often we find that there may be more than one solution that is appealing. If in the end we pick only one it is either because we have picked one at random or have then decided that all things being equal we would rather emphasize one of the criteria over the others. If

we decided that cost was more important to us in the above example option 2 would have been the choice. Our goal in the research presented here is to solve the multi-criteria optimized problem and produce the Pareto-optimal set of solutions. We leave in the hands of a user the final decision of which solution is best.

Other itinerary planners have been implemented in the past. Some are currently used on the world wide web. Surprisingly, none of the systems currently available aim to solve the problem in a truly multi-criteria way. Some reduce the independent criteria to a combined function and others use heuristics to get reasonable solutions to the problem.

1.2 Problem Statement and Our Approach

This thesis focuses on the problem of planning itineraries through the public transport systems of the North East Corridor in the United States. The goal is to produce a set of algorithms that can answer the question "What is the best way to go from A to B?" using public transportation.

Transportation has proved to be an interesting area in the study of shortest path problems. Many of the techniques focus on time-dependent versions of the classical shortest path problem. Interestingly, apart from a few early attempts on this variant of the problem in 1966 by Cooke and Halsey [8] and subsequently by Dial [11], there are almost no references in the literature until the 1980s when there was a renewed interest in this variant of the problem. Another interesting aspect of shortest-path problems in transportation

is that the presence of non-additive and non-linear path costs.

The approach we take is to extend the classic Bellman-Ford-Moore shortest path techniques to solve multi-criteria problem on networks with time-dependent edges and non-linear path costs. While the multi-criteria combinatorial optimization problem is known to be NP-Hard we demonstrate that for this problem solutions can be efficiently computed in practice.

To solve the itinerary problem we introduce a domination predicate that allows us to compare solutions and arrive at a set of "best" solutions. A common theme in the algorithms that are presented here is that the most of the variants of the itinerary problem presented are solved by altering the domination predicate and the sets of solutions on which it is applied.

Another theme in this thesis is the use of graph reductions to both reduce the complexity of the algorithms and to allow the same algorithm to solve the different variants of the itinerary problem.

In addition, we look at solving the itinerary problem in the presence of a soft real-time constraint. For the algorithms here to be useful in decision making, a response should be provided in a short amount of time.

We aim to implement the algorithm in as general a fashion as possible. This helps us solve problems in inter-modality, where different parts of the graph can have different behavior. In the transit arena this happens naturally due to the presence of different transit providers and the ways that they levy fares for using their systems.

1.3 Thesis outline

This rest of this thesis is organized as follows.

Chapter 2: This chapter surveys the literature for approaches taken to solving the component parts of the itinerary problem. The topics covered are: the class of shortest path list algorithms focusing on how to improve their performance. Time-dependency, the problem of how to model time-dependence in networks and the new problems and algorithms that time-dependence introduces to the classical shortest path algorithm. Intermodality, the problem of moving between different carriers and how it can increase the complexity of the problem. Non-linear and non-additive cost functions and how they impact the general Bellman-Ford-Moore framework. Real-Time constraints, the problem of adding real time constraints to the algorithm. Finally, we introduce the area of multi-criteria optimization and touch upon a few of the techniques used to solve problems in this area.

Chapter 3: This chapter states the itinerary problem formally.

Chapter 4: This chapter describes how to construct a dynamic graph to model transit networks, introduces the general framework for solving the itinerary problem and starts to discuss the issues involved in using the domination predicate.

Chapter 5: This chapter introduces the idea of performing static analysis on the graph to meet the real time goals of an itinerary planner. We also introduce a new variant of the algorithm to aid in the precomputation necessary. Results on the performance of the variants of the algorithm are

discussed.

Chapter 6 (Conclusion): The techniques introduced in this thesis, their strengths and shortcomings are discussed along with future directions for this work.

Chapter 2

Shortest Path Problems in Transportation Models

This chapter surveys material on combinatorial optimization in order to frame and solve the path optimization problems that are the focus of this thesis. At the core of most path optimization problems in transportation models lies the traditional Bellman-Ford algorithm. The literature on this one topic is substantial and for over forty years different variants of this algorithm have been studied for a myriad of applications. Transportation problems are particularly interesting in this respect because they solve problems on graphs that originate from real networks and in the transportation domain this usually involves focusing on large time dependent graphs that have unusual cost functions. The problem that is central to this thesis is finding optimal paths in a large scale time dependent network where there are multiple cost functions.

This chapter surveys the techniques used in the literature to address the following topics:

- Complexity, variants of the Bellman-Ford scheme are studied, all of which aim to reduce the complexity of the search for optimal solutions in large graphs.
- Time dependence, the idea of time-dependent edges in a graph is introduced along with a few core algorithms that solve time-dependent graph problems.
- Modality, a problem which addresses moving between different transport carriers and the associated costs of doing so.
- Non-additive and non-linear cost functions, here traditional techniques based on the Bellman conditions break down and other ways to solve graph problems where paths have non-linear and non-additive costs are introduced.
- Multicriteria optimization, finding optimal solutions when there is more than one cost function is introduced.

2.1 The Shortest Path Tree Problem

Let $G = (N, A)$ be a simple directed graph where N is a set of nodes of cardinality n and A a set of edges of cardinality m. Let $c : A \to \mathbb{R}$ be a *cost labeling function* that assigns a cost c_{ij} to each $(i, j) \in A$. For a node $i \in N$,

let $FS(i)$, be the *forward star* of node i, i.e. the set of outgoing, adjacent arcs to node i given by, $FS(i) = \{(i,j) \in A\}$, and let $Adj[i]$ be the set of vertices adjacent to i. Also, for a node $i \in N$, let $BS(i)$, the *backward star* of node i be given by $BS(i) = \{(j,i) \in A\}$.

Given a root node $r \in N$, the *Shortest Path Tree* (SPT) problem is to find a directed tree T such that for each $i \in N$ that is connected to r, the only path from r to i in T is one of the shortest paths from r to i in G. Nodes i and j are said to be *connected* if and only if there exists a path from i to j in G. If each $i \in G$ is connected to r then T is also a spanning tree denoted by T^*.

A well known result is that a finite solution exists for the SPT problem if and only if there is no directed cycle of negative cost in G.

2.1.1 Primal algorithms for SPT.

Most of the algorithms proposed to solve the SPT problem follow a primal approach. These algorithms work by growing a directed spanning tree T rooted at a node r by expanding paths from r along the tree. In reality they start with a fictitious minimum cost tree T and iteratively update the tree until a minimum cost path tree T^* is found.

Let $\mathbf{C} = \{C_1, C_2, \ldots C_n\}$ be a set of *cost labels* C_i that represent the *cost* of the path from r along the spanning tree T to each node i. At all times the label C_i provides a upper bound on the cost of the minimum cost path from r to i. Let $\pi[i] : N \rightarrow N$ be the *predecessor function* where $\pi[i]$ is given

by $\pi[i] = \{j|(i,j) \in T\}$. The function π provides an implicit description of T allowing the path from r to i to be reconstructed easily.

During each iteration, in a process known as *scanning*, a node i is selected and the algorithms check that for the entire forward star of i the *Bellman condition*,

$$C_j \leq C_i + c_{ij}, \forall (i,j) \in A$$

holds.

The *reduced cost* of an arc with respect to the label set $\mathbf{C}$ is a function $\bar{c} : A \to \mathbb{R}$ defined by

$$\bar{c}_{(i,j)} = c_{ij} + C_i - C_j$$

There is an equivalence between the non-negativity of the reduced cost function and Bellman's condition holding. Where the reduced cost function is negative $\bar{c}_{(i,j)} < 0$ the label of node i is *improved* $[C_j \leftarrow C_i + c_{ij}]$ and the tree is modified by replacing arc $(\pi[j], j)$ with arc (i,j), $[\pi[j] \leftarrow i]$. If $T_i = (N, A)$ is the subtree of the current tree T rooted in i, and if for all the nodes in N_i their cost labels decreased by $-\bar{c}_{(i,j)}$ then all the arcs in T *verify* the Bellman conditions with respect to the edges processed so far and the minimum cost path from r to i using only the processed edges is given by the cost of the unique path from from r to i on T [9].

To compute the SPT of a graph G the algorithms typically maintain a set of *candidate nodes* Q along with the label set $\mathbf{C}$, which is typically initialized with $C_r = 0$ and $C_i = \infty$, for all $i \neq r$. Initially the set Q contains only the

node r. At each iteration a candidate node i is selected from Q, scanned, and where necessary the labels and predecessors of i are updated. The algorithm terminates when Q is empty and the Bellman conditions hold for all the arcs in T; the node labels now contain the minimum path costs and the predecessors describe a shortest path tree T^*.

```
GenericSPT()
1  InitializeSPT(Q)
2  while Q.NotEmpty()
3  do i ← Q.Select()
4      for j ∈ Adj[i]
5      do if C_i + c_ij < C_j
6             then Update(i, j)
7                  Q.Enqueue(j)
```

Figure 2.1: The GenericSPT Algorithm.

A familiar version of the Update procedure of T is shown below but it is important to note that this is the most naïve version. Update may update labels of nodes other than the current node. In some algorithms the update is propagated over the entire subtree and in others the entire tree substructure may be modified.

More efficient approaches do not propagate label settings on subtrees, but instead maintain a record of the updated labels, using Q, and later check the Bellman conditions on their forward star. At any given time, the node

UPDATE(i, j)

1 $C_j \leftarrow C_i + c_{ij}$

2 $\pi[j] \leftarrow i$

Figure 2.2: A simple UPDATE function.

labels are an upper bound on the path cost from r on T. Since scanning a node is a wasted effort if its label is not exact, most algorithms implement a strategy to reduce the number of wasted scanning operations [21].

The many variations of these primal algorithms differ in the handling of Q and in particular how to implement a selection rule for the candidate nodes in Q and how to update the shortest path tree.

In order to analyze the behavior of the various shortest path algorithms a few definitions are necessary.

- C_i, the *label* of node i, represents an upper bound to the cost of the current path from r to i on T. If C_i is the cost of the path in the current tree T it is an *exact* label.
- C_i^*, the minimum label of all possible C_i, is a *permanent* label.
- The tree T described by the predecessor function π is the *current tree* and when $T = T^*$ it is *optimal*.
- Nodes never inserted into Q are *unreached* and those currently in Q are known as *candidates*.

- Nodes are *scanned* if they are selected from Q and removed from Q. A node remains scanned until it is reinserted in Q.

- *Discarded* nodes are those nodes that are selected and removed from Q but are not eligible for examination.

There is a relationship between the current tree T and the label set $\mathbf{C}$. A node i with an exact label only has arcs with zero reduced cost on the path from r to i on T. If the node has an inexact label, there is scope for improving the label and therefore at least one arc must have a negative reduced cost. No arc with a positive reduced cost belongs to T and if a node has an inexact label then one of its ancestors in T must belong to Q. Scanned nodes with permanent labels will never be reinserted into Q.

2.1.2 SPT_S Algorithms

SPT algorithms that select the minimum distance label in choosing a node to scan are known as *shortest-first search* algorithms, *label-setting*, or SPT Setting (SPT_S) algorithms. Dijkstra [12] proposed the first of these algorithms. In label setting algorithms the node exiting Q is the node with the minimum cost value over all nodes in Q. In the GenericSPT algorithm above the Q.Select simply selects the node with the minimum distance label and the Q.Enqueue operation does nothing, since a node, once it has exited Q, need never reenter Q.

Properties of the graph on which the algorithms can be run and the choice of the data structures that implement Q determine the correctness

and computational complexity of these algorithms. For example, in the case that G contains no arcs with negative costs and the candidate nodes are kept in a simple linked list the computational complexity of the algorithm is $O(n^2)$ time. In the general case, where negative arc weights are allowed the algorithm runs in $O(n2^n)$ time [18]. Implementing the candidate set as a binary heap, the *SPT_S-Heap* algorithm on a graph with with non-negative cost arcs reduces the running time to $O(m \lg n)$ time; this is the most familiar form of Dijkstra's algorithm. The fastest strongly polynomial algorithm in the case of nonnegative arc costs, discovered by Tarjan [34], uses Fibonacci heaps to implement Q and has a running time of $O(m + n \lg n)$.

2.1.3 SPT_L Algorithms

The other major class of algorithms, the *label-correcting* algorithms, are also known as SPT_L, *Shortest Path Tree - List*, algorithms. Typically these algorithms are used when the problems being solved deal with larger classes of cost functions.

SPT_L algorithms allow a node v to enter the set Q multiple times. In general, nodes are inserted and dequeued from the candidate set in $O(1)$ time. At each iteration the top node from Q is removed. The algorithms differ in choosing an appropriate position to insert a node i into Q.

In the Bellman-Ford-Moore method, the simplest label correcting method, the candidate set, a single list, implements a first-in/first-out(FIFO) queue Q. The Q.ENQUEUE and Q.SELECT functions of the GENERICSPT simply

become an insert at the end of a list and remove from the beginning of the list respectively. The first pass of the algorithm consists of scanning the root of the tree r and for each of the subsequent k passes the algorithm scans the nodes added to Q during the $k-1$ pass. The method can be shown to require at most $O(mn)$ iterations, independent of the sign of the arc costs. This is because each node can only be scanned once for each pass and at most n passes are needed, each pass costing $O(m)$ time. In practice the larger number of iterations, when compared to typical label-setting algorithms, is offset by the smaller overhead per iteration due to the trivial node selection strategy; nodes are simply dequeued from the front of a list, an $O(1)$ time operation. Where G is acyclic the number of iterations is exactly n, the same as for Dijkstra's method. Remarkably, after over forty years of study, this algorithm still provides the minimum known complexity for finding shortest paths on graphs with arbitrary cost arcs.

As stated earlier, improvements to the SPT_L class come from minimizing the number of scan operations. Gains can be made by trying to anticipate the updating of inexact labels of the nodes currently in Q before they are actually scanned. Since inexact labels may arise only once a node has been reinserted in Q for the first time, the D'Esopo-Pape (SPT_L-Dequeue)[28] method places a node that enters Q for the first time at the bottom of the queue and a node that reenters Q it is placed back at the top of the queue. In the worst case the number of iterations has been shown to be $\Theta(2^n)$ [19] even when the arc weights are nonnegative. The stack (LIFO) nature of

nodes reentering a queue is the cause of the exponential time complexity. It is possible that a node may be selected and scanned $2^{(n-2)}$ times in the same pass. In practice, however, the D'Esopo-Pape algorithm performs very well, often outperforming Bellman-Ford-Moore on sparse graphs.

In a second variant of this algorithm (SPT_L-2Queue), the queue Q is partitioned into two queues Q_{first} and $Q_{reentering}$. A node exiting the list of nodes to be considered is removed from the top of $Q_{reentering}$. If $Q_{reentering}$ is empty the node is removed from the top of Q_{first}. A node that enters Q for the first time is queued at the bottom of Q_{first}, a node that re-enters Q is placed at the bottom of $Q_{reentering}$. This second variant of the D'Esopo-Pape algorithm has roughly the same performance in practice as the single queue variant but the algorithmic running time can be shown to be $O(n^2m)$ since a node can be selected and scanned no more than n times in a single pass. Both variants can be thought of as implementing Q as two lists connected in series. In the first algorithm, $Q_{reentering}$ is a list with last in/first out (LIFO) behavior and Q_{first} a simple FIFO queue as before. In the second variant, both lists follow a FIFO strategy for enqueuing and dequeuing vertices.

Other algorithms use disjoint queues to enqueue and dequeue nodes with the aim of increasing the probability of selecting nodes with permanent labels. Glover, Glover and Kingman [13] use a thresholding strategy that effectively turns the two disjoint lists into a set of dynamic buckets. A threshold s is chosen, often by experiment, and the Q is implemented as a pair of disjoint queues partitioned as follows, if a node i being queued has

a label $C_i \leq s$ the node is placed in the queue $Q_{\leq s}$, otherwise the node is placed in the queue $Q_{>s}$. Nodes are dequeued from $Q_{\leq s}$ until this queue is empty. When the queue is empty, the threshold value is changed and all the nodes left in $Q_{>s}$ with a threshold lower than s are moved to $Q_{\leq s}$.

When the arcs have nonnegative edge weights the algorithm requires only n^2 iterations with $O(n^2)$ operations. The threshold algorithm performs extremely well on randomly generated problems but its performance is sensitive to the threshold adjustment scheme. For non-negative arcs if,

$$s \leq \min\{C_i | i \in Q\} + \min\{c_{ij} | (i,j) \in A\}$$

i.e. the threshold values are too small, the algorithm reduces to the naive Dijkstra's algorithm. When the threshold values are initially too large, i.e. $s > (n-1)\max\{c_{ij} \in A\}$ the nodes are all inserted directly into $Q_{\leq s}$, and if that queue is implemented as a simple FIFO list the algorithm reduces to the Bellman-Ford-Moore algorithm. Finding a threshold selection policy for a given class of graphs may need a considerable amount of experimentation and one may even be unable to find an effective adjustment scheme although there are schemes for finding a threshold for broad classes of randomly generated problems.

Based on the hypothesis that for many types of problems, the number of iterations of a label correcting method strongly depends on the average rank of the node exiting Q, where nodes are ranked in terms of the size of their labels (small labels correspond to lower ranks), Bersekas proposes a much simpler heuristic for node selection, the *Small Label to Front* (SPT_SLF)

strategy. In this strategy when a node j enters Q its label c_j is compared with the label of the top node i in Q. If $c_j < c_i$, node j is entered at the top of Q, otherwise j is entered at the bottom of Q.

Combining this strategy with the thresholding strategy discussed above yields the *SPT_SLF-threshold* label correcting method for finding shortest paths. Here the queue Q is partitioned into two queues, $Q_{\leq s}$ and $Q_{>s}$, where s is once again a threshold value. When a node j enters the queue for the first time it is placed at the top of $Q_{\leq s}$ if $c_j \leq c_i$ and $c_j > c_i$, where i is the top node of $Q_{\leq s}$. The same policy is used when a node enters $Q_{>s}$ and when a node is transferred from $Q_{>s}$ to $Q_{\leq s}$. When $Q_{\leq s}$ becomes empty the nodes in $Q_{>s}$ are checked sequentially and if a node j satisfies the test for entry into $Q_{\leq s}$ the node is moved to that queue.

In an alternate versions of the SPT_SLF-threshold algorithm when a node j is already in either Q and the label is decreased one can compare the new label c_j with the label c_i of the top node of the queue. If $c_i < c_j$ the node is moved to the front of the queue. This version requires that the queue be implemented as a doubly linked list to keep the queue efficient. While this restriction imposes more overhead, experimental results show that it leads to a further reduction in the number of iterations.

The complexity of the SPT_SLF-threshold algorithm is not well established but some variants can be shown to run in $O(nm)$ time for non-negative arc lengths. Whether the SLF-Threshold algorithms are in general polynomial remains an open question. However, in many classes of graphs, the

SLF-Threshold algorithm has been shown experimentally to be extremely efficient, requiring many fewer iterations than the Bellman-Ford-Moore algorithm and is considerably faster than two list algorithm of D'Esopo-Pape. This improvement has been shown to be due to the high correlation between the number of iterations of a label correcting method and the average rank of the node exiting the queue.

Tarjan [33] proposed a single queue variant of the SPT_L algorithm where only nodes with exact labels are scanned, discarding candidate nodes with inexact labels as soon as they arise. The inexact labels are collected by exploring the subtree T_i after the label C_i has been updated. At each iteration, when i is selected and scanned, if the Bellman condition is violated, i is updated and its descendant labels become inexact. At this point, T_i is traversed and every node except i is discarded and is marked for removal from Q. Node i is then inserted at the tail of a FIFO list, Q, if it is not already in Q. T contains only nodes with exact labels. To collect nodes with inexact labels from T a *subtree disassembly* strategy is used. The tree structure is implemented using the usual predecessor function as well as a *first child* and *adjacent child* function, representing next and previous relationships respectively. This allows the traversal of T_i with a linear complexity but more importantly allows tree modification in constant time. As with the Bellman-Ford-Moore algorithm the disassembling queue variant runs in $O(mn)$ time. At most n passes are necessary and a node is scanned at most once for each pass.

```
Q.SELECT()
1  while Q ≠ ∅ and i = first(Q) is marked discarded
2  do Q←Q−{i}
3      return i
```

Figure 2.3: The SPT_L Dissasembling Q.SELECT function

```
Q.ENQUEUE(i)
1  traverse T_i and mark discarded all nodes ≠ i
2  if i ∉ Q
3     then Q = Q + i
```

Figure 2.4: The SPT_L Dissasembling Q.ENQUEUE function

Goldberg and Radzik [15] suggest a heuristic improvement to the Bellman-Ford-Moore algorithm based on visiting the nodes in a graph in a *topological scan* in order to radically reduce the number of nodes scanned while building the shortest path tree.

An edge e is *admissible* if $\bar{c}_{(e)} < 0$. Extending this definition, $G_{SA} = (N, \bar{A})$, is the partial graph with arcs of non-positive reduced cost, i.e. $\bar{A}$ is the set of all admissible edges. G_{SA} is known as a *strictly admissible graph* with respect to a root node r. G_A is known as an *admissible graph* with respect to r if all the arcs in the graph have a non-negative reduced cost function.

If two nodes i and j are labeled and $\bar{c}_{(i,j)} < 0$ it is better to scan i before j, since when we scan i the label C_j will improve and j will become labeled. The node i is *improvable* by a new node j if in the admissible graph, i is connected to j by a path for which there is at least one arc in G_A. Nodes in the set of labels $\mathbf{C}$ are in a *topological order* if every node always follows all other nodes by which it is improvable. The topological scanning algorithm alters the order in which candidate nodes in Q are processed, placing them in a topological order. It does so by implicitly visiting the graph G_A.

The algorithm topological-scan maintains two queues, $Q_{ordered}$, nodes to be scanned on an iteration, and $Q_{visited}$, a set of visited candidate nodes to build $Q_{ordered}$ from. Each node can only be in one set at a time. Initially $Q_{ordered} = \{\}$ and $Q_{visited} = \{s\}$. Each iteration of the algorithm starts, when $Q_{ordered} = \emptyset$, by populating the queue $Q_{ordered}$, choosing nodes in $Q_{visited}$.

Computing the set of nodes to be scanned from the candidate set is performed as follows. Remove all nodes $i \in Q_{visited}$ with no outgoing arcs with negative reduced cost. Mark all the nodes that are reachable from $Q_{visited}$ in G_A as labeled and place them in $Q_{ordered}$. Topologically sort $Q_{ordered}$ so that for every pair of nodes i and j with an edge (i, j) in G_A, i should precede j forcing i to be scanned before j.

To construct $Q_{ordered}$ from $Q_{visited}$ efficiently, in a single depth first search, both sets are implemented as stacks. Popping nodes from $Q_{visited}$ one by one, if i has already been visited by the current depth first search it is ignored, if i

has not been visited but has no outgoing edges with a negative reduced cost i is marked as scanned, and if i has not been visited and has an outgoing edge that can be traversed with a negative reduced cost the algorithm visits in depth first order all nodes that are *reachable* from i in G_A which have not been previously visited. At the end of the visit of node i it is marked visited and pushed onto $Q_{ordered}$. By taking care to implement the procedure so that nodes are only considered from $Q_{visited}$ at the beginning of each iteration or from $Q_{ordered}$ at the end of each iteration the algorithm makes sure that each node is visited exactly once.

```
Q.SELECT()
1  if Q_ordered ≠ ∅
2     then i ← Q_ordered.POP()
3          return i
4     else while Q_visited ≠ ∅
5          do i ← Q_visited.POP()
6             DFS_VISIT(i)
```

Figure 2.5: SPT_L Topological Ordering Q.SELECT function

In [15, 7] the algorithm is proved to run in worst case $O(mn)$ time. A variant of the algorithm, with the same running time, also due to Goldberg [15] makes sure that labels are immediately updated, during the topological ordering, to enlarge the visited graph. The *DFS* visit is performed on the current strictly admissible graph G_{SA} that is produced by each label update

```
DFS_VISIT(i)
1  for j ∈ Adj[i]
2  do if C_i + c_ij ≤ C_j
3        then //Only DFS visit j if the edge (i, j) is in the admissible graph.
4             DFS_VISIT(j)
5     if i ∉ Q_ordered
6        then Q_ordered.PUSH(i)
```

Figure 2.6: SPT_L Topological Ordering DFS_VISIT function. The DFS_VISIT algorithm implicitly visits the admissible graph G_A by using the Bellman condition as a guard for deepening the search. In practice this is usually implemented using a stack instead of the recursion as shown above.

```
Q.ENQUEUE(i)
1  if i ∉ Q_visited
2     then Q_visited.PUSH(i)
```

Figure 2.7: SPT_L Topological Ordering Q.ENQUEUE function

in **C**. The implementation details can be found in [21]. It is also shown that if G is acyclic, the ordering performed in the first iteration is a topological order for G and so the time complexity for this variant is linear in the number of arcs [34].

Over the years there has been a large amount of experimental testing of algorithms in the classes SPT_L and SPT_S. All the SPT_L algorithms

behave well with sparse graphs, especially when the arc costs have a close relation to the topology of the graph. This often arises in graphs constructed to represent real world networks, where the cost functions represent quantities like distances, costs, and travel-times. The basic Bellman-Ford-Moore algorithm has, on average, the worst running time, not surprisingly, as it performs the largest amount of inexact label scanning. Variants of the basic SPT_L algorithm all try to minimize the number of inexact scans. The SPT_L Threshold and SPT_L Topological Ordering perform well on average on a large classes of graphs. Pallottino *et al.* [21] describes variants of the Tarjan disassembling technique and the Goldberg-Radzik topological reordering techniques, benchmarking the different variants against a large class of graphs.

2.2 Dynamic Shortest Path Problems

A vast majority of the literature on shortest paths is dominated by networks which have a fixed topology and fixed linked costs. Of late, due to interest in graphs that model transportation systems, there has been a renewal of interest in a class of problems known as dynamic shortest path problems. In the literature, when referring to shortest path problems, the term "dynamic", has two meanings. The first, which is not the focus of this thesis, involves the problem of computing the shortest path in a graph where at any given time the edge weights and the topology may change, altering the properties of paths through the graph. The goal of much of the research in

this case is to tackle the problem of computing the effects of incremental changes in the graph on the property being optimized. For example, if after computing the shortest paths from a root node to all other nodes, a single edge changes then, where possible, it is desirable that calculating the change in the spanning tree should on average take substantially less time than the initial computation of the shortest path spanning tree.

This thesis is concerned with the second meaning of "dynamic" in the literature. Here the task at hand is to model the time dependency of edges in the graph and in particular, to study the impact of this dependency on finding optimal paths. Edges in "dynamic" graphs are labeled with time properties that determine "when" an edge can be traversed and "how long" it takes to traverse the edge.

It should be noted that in this second category the behavior of time can be modeled as part of a set of continuous values [25, 26] or in a discrete way [4, 6, 27]. The discussion here considers only the discrete model.

For graphs that model time dependency, under the discrete time model, the algorithms and the analysis of their performance typically center around the use of an expanded static version of the network where the time dependencies of edges and their interactions with the nodes that they connect are represented. It is this underlying space-time network that is central to much of the discussion that follows. While it is not the best approach to explicitly use the expanded space-time networks many of the properties that they expose about dynamic graphs are used to design better algorithms.

2.2.1 Space-Time Networks

Modeling the properties of time dependency in a graph where time is treated discretely needs to address the issues of when an edge can be traversed and, what the cost of traversing a link is at a given time.

More formally, in a dynamic graph $G = (N, A)$, every arc (i, j) has an associated *delay cost* or *travel time* d_{ij}. Thus if t is the *departure* time from a node i along an arc (i, j) then the $t + d_{ij}(t)$ is the *arrival time* at j. In addition the cost function $c_{ij}(t)$ for an arc is the cost associated with traversing (i, j) at time t.

In the discrete model of the dynamic shortest path problem the values that the time variable can take belong to the discrete set $\mathbb{T} = \{t_1, t_1, \ldots t_q\}$ and the delay function $d_{ij} : \mathbb{T} \to \mathbb{T}$, is defined on the set $\mathbb{T}$. To model the behavior of waiting at a node i there is a *waiting cost* function $w_i(t)$ for each node i. For the rest of this discussion all the arcs are assumed to have a non-negative travel time.

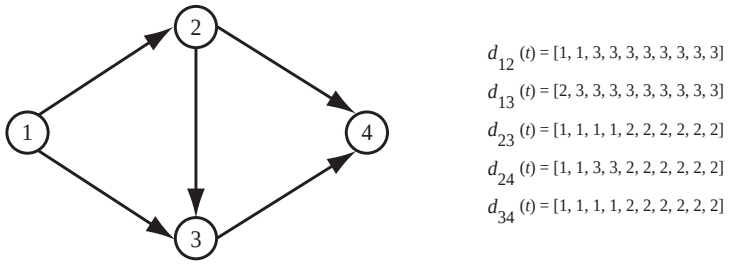


Figure 2.8: An example of a dynamic network

The *Space-Time Network* $R = (V, E)$ of a dynamic graph $G = (N, A)$

under the discrete model is defined by,

$$\begin{aligned} V &= \{i_h | i \in N, h \in [1,q]\} \\ E &= \{(i_h, j_k) | (i,j) \in A, t_h + d_{ij} = t_k, h \in [1,q], k \in [1,q], h < k\} \end{aligned} \tag{2.1}$$

The cost for each arc (i_h, j_k) is given by $c_{ij}(t_h)$. The cost of waiting, introduced by the *wait* function $w_i(t)$ is modeled by adding arcs to E of the form (i_h, i_{h+1}) with $h \in [1,q)$.

R is now a standard acyclic graph with a pseudo-polynomial size with respect to G from which it was constructed. The sizes of the vertex and edge sets are given by $|V| = nq$ and $|E| \leq (m+n)q$ respectivly. Every chronological visit of the nodes in R, the nodes with non-decreasing values of time, provides a topological visit on R.

An arc $(i,j) \in A$ in a dynamic graph is said to be a *FIFO arc* (first-in/first-out) if leaving a node i earlier guarantees that one will arrive no later at j along (i,j).

$$t_h + d_{ij}(t_h) \leq t_k + d_{ij}(t_k) \quad , \quad t_h < t_k \tag{2.2}$$

A dynamic graph is a *FIFO graph* if all its arcs are FIFO arcs. The implication of the FIFO property is that waiting at node i before traversing (i,j) will never cause an earlier arrival at j. For example, if a traveler waits at i for a period of time and then starts traversing an arc, the traveler may spend less time traversing (i,j), but because of the FIFO property, will never arrive at j sooner, had the traveler left i without waiting.

Where waiting at a node is permitted a similar property can be imposed on the arc costs. If departing i earlier along an edge (i,j) does not cost more

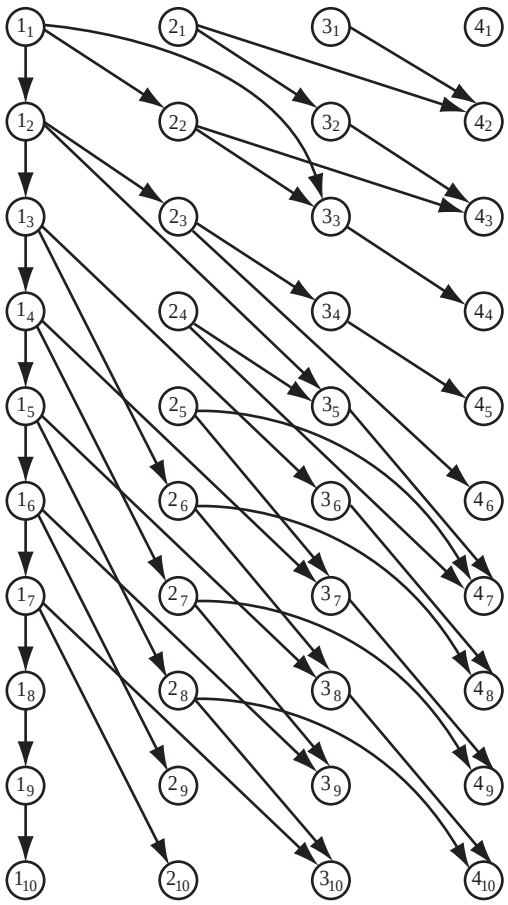


Figure 2.9: An example of a space-time network

than departing later the arc is *cost consistent* (CC).

A graph is a *cost consistent graph* if all its arcs are also cost consistent.

Let $t_u = t_h + d_{ij}(t_h)$ and $t_v = t_k + d_{ij}(t_k)$, for $t_h < t_k$ then,

- (i, j) is a FIFO arc ($t_u \leq t_v$); here it is also CC if for any $t_h < t_k$;

$$c_{ij}(t_h) + \sum_{z=u}^{v-1} w_j(t_z)(t_{z+1} - t_z) \leq c_{ij}(t_k). \tag{2.3}$$

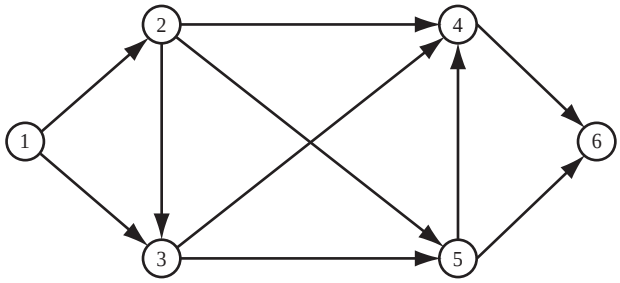


Figure 2.10: If the delays on all the arcs are equal to 1 and the time period $\mathbb{T} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ the graph is FIFO.

- (i, j) is a not a FIFO arc $(t_u > t_v)$. Here it is a CC arc if, for $t_h < t_k$;

$$c_{ij}(t_h) \leq c_{ij}(t_k) + \sum_{z=v}^{u-1} w_j(t_z)(t_{z+1} - t_z). \tag{2.4}$$

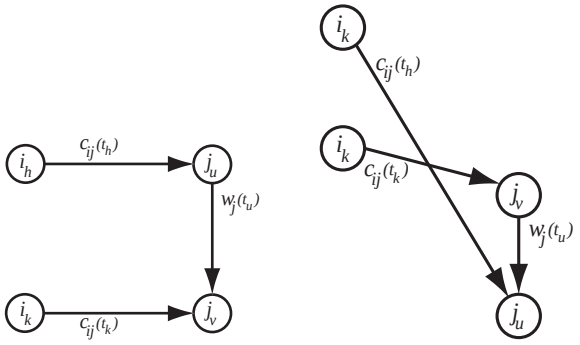


Figure 2.11: CC arc (i, j): (a) (i, j) is FIFO $(u \leq v)$. (b) (i,j) is not FIFO $(u > v)$

As stated earlier, for most problems involving dynamic graphs it is possible to work on the space-time network R implicitly by using a topological visit on R since R is acyclic. In addition, only the non-redundant part of R need be considered.

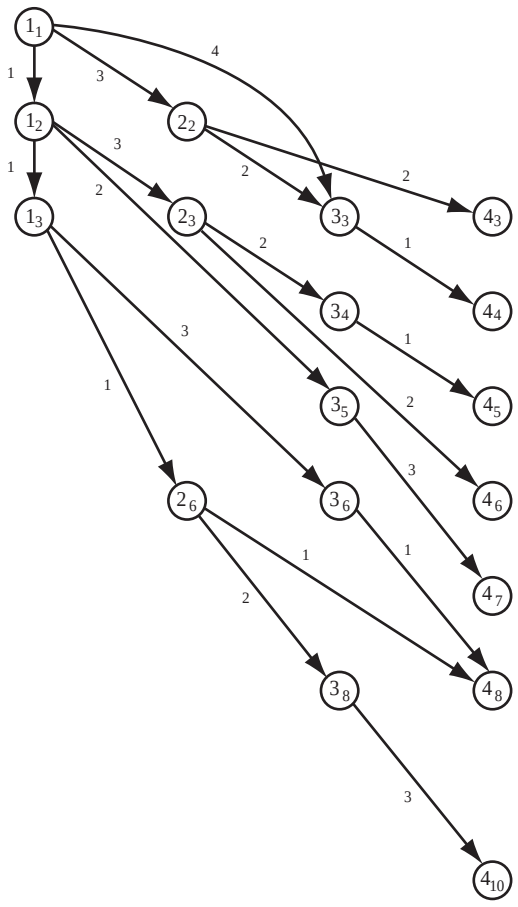


Figure 2.12: The non-redundant portion of the space-time network

2.2.2 The Minimum Cost Dynamic Path Problem.

The *Minimum Cost Dynamic Path Problem* looks for a path from a node r to every other node $i \neq r$, leaving at a time t in a dynamic graph G.

Computing a shortest path tree with topological ordering in R using a bucket list $B = \{B_1, B_2, \ldots, B_q\}$ [11] to efficiently implement the chronological visit of R, where B_h denotes nodes to be visited at time $t_h, h \in [1, q]$,

yields the CHRONO-SPT algorithm. Initially, if departing from a node r at time $t = t_p$ then $B_p = \{r\}$ and the other buckets are empty. The algorithm terminates when all the buckets are empty and all the labels are set to the optimum path from r to every other node i. At each iteration the shortest path tree is expanded using the CHRONO-SPT operation for a node v and a time h.

CHRONO-SPT(i, h)

1 *(typical iteration)*
2 **select** i **from** $B_h; B_h \leftarrow B_h - \{i\}$
3 **for** each edge $(i, j) \in FS[i]$
4 **do** $t_k \leftarrow t_h + d_{ij}$
5 **if** $C_i(t_k) + c_{ij}(t_h) < C_j(t_h)$
6 **then** $C_j(t_k) \leftarrow C_i(t_h) + c_{ij}(t_h)$
7 **if** $j \notin B_k$
8 **then** $B_k \leftarrow B_k \cup \{j\}$
9 **if** $C_i(t_h) + w_i(t_h)(t_{h+1} - t_h) < C_i(t_{h+1})$
10 **then** $C_i(t_{h+1}) \leftarrow C_i(t_h) + w_i(t_h)(t_{h+1} - t_h)$
11 **if** $i \notin B_{h+1}$
12 **then** $B_{h+1} \leftarrow B_{h+1} \cup \{i\}$

Figure 2.13: The code above is for a typical iteration of the CHRONO-SPT. Lines 7-10 in the CHRONO-SPT code deal with the case where waiting at i is allowed.

By considering nodes in R chronologically, i.e. first consider nodes with

time t_1, then t_2 and so on, the shortest path tree with topological ordering in R is generated. This strategy implicitly generates the non-redundant portion of R. It is easy to see that CHRONO-SPT runs in time $\Theta(q + |E^*|)$ where $E^* \subseteq E$ is the non-redundant set of arcs implicitly generated by CHRONO-SPT. Since $|E^*| \leq mq$ in the worst case the algorithm's time complexity is $O(mq)$

If a dynamic graph is both FIFO and cost consistent the local properties can be exploited to solve the SPT problem using stronger assumptions. If G is both cost consistent and FIFO, i.e. leaving along any arc (i, j) at an earlier time causes one to arrive no later than before and at a cost that is no higher than before, then the property of "dominated labels" can be introduced and exploited. Suppose we have visited two different paths from the given origin node r to a node i, and suppose that the two paths arrive at i at time t_h and at time t_k such that $t_h < t_k$ with the costs of the paths being $C_i(t_h)$ and $C_i(t_k)$ respectively, with $C_i(t_h) \leq C_i(t_k)$. In this case there is no need to extend the path that arrives at time t_k as this expansion will not yield a minimum cost path going through node i. The label $C_i(t_k)$ is the *dominated label* for node i, and can be ignored. For a given node i the set of non-dominated labels $\mathbf{C}_i = \{C_i(t_{h_1}), C_i(t_{h_2}), \ldots, C_i(t_{h_z})\}$ then if $t_{h_1} < t_{h_2} \ldots < t_{h_z}$ then it must be the case that $C_i(t_{h_1}) > C_i(t_{h_2}) > \ldots > C_i(t_{h_z})$.

The CHRONO-SPT procedure can be modified to maintain only the set of non-dominated labels. Using the bucket implementation as before dominance need only be checked when a label $C_i(t_h)$ is selected from the current

bucket. It is enough to compare $C_i(t_h)$ to the last selected non-dominated label, *last-label*$_i$.

CHRONO-SPT (FIFO AND CC)(i, h)

```
1  (typical iteration)
2  select i from B_h; B_h ← B_h − {i}
3  if C_i(t_h) < last-label_i
4     then last-label_i ← C_i(t_h)
5          for each  edge (i, j) ∈ FS[i]
6          do t_k ← t_h + d_ij
7             if C_i(t_k) + c_ij(t_h) < C_j(t_h)
8                then C_j(t_k) ← C_i(t_h) + c_ij(t_h)
9                     if j ∉ B_k
10                       then B_k ← B_k ∪ {j}
```

Figure 2.14: If the test on line 2 of CHRONO-SPT (FIFO) fails the label $C_i(t_h)$ is dominated and immediately discarded. In the case that the test succeeds the algorithm proceeds to expand the SPT by traversing outbound edges from i.

Minimum Arrival Time Dynamic Path Problem

Another problem of particular interest is the *minimum time dynamic path problem* for a specific departure time , where after specifying the root node, we wish to find the earliest arrival time to any other node $i \neq r$. In this version of the problem the weights of the arcs in the space-time network

can be ignored and the problem is now reinterpreted as a connectivity problem. In this case, the CHRONO-SPT is reduced to the classic Dial's algorithm [11] with bucket-list implementation. In this case the complexity is $O(m + \min\{q, n \log n\})$ Once again, the algorithm works implicitly on R. The algorithm runs as before ignoring the cost function. Once each node $i \neq r$ has been visited the algorithm terminates. It is trivial to show that if the graph is FIFO then only one non-dominated node is associated with each node.

2.2.3 Train Graphs

Train graphs [24, 14] are a specialization of a space-time network in the context of transportation problems. An *event*, or vertex, in the train graph represents every time a train enters or leaves a station. Two events, v and w are connected by an arc from v to w if v represents the departure of a train from a station and w represents the arrival of the same train at the next station. Successive events at the same station are connected by an edge in the positive time direction, there is also an arc from the last event, at a given station, to the first event. Hence all the events at a station are connected by a simple cycle.

Clearly the train graph addresses the same issues in modeling the time dependent behavior of railway networks as the space time network does. However the explosion in the number of vertices is much smaller, in fact it is now exactly $2m - 1$. The number of edges in the train graph is also exactly

$2m$.

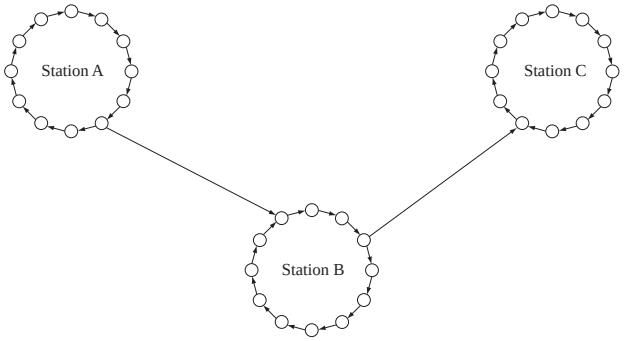


Figure 2.15: A train graph.

With one extra addition the train graph model can also be used to capture the number of train changes. When a train T leave a station a at some event u, and stops next at a station b at event v_1 then continues on to station c with event v_2 where it arrives with event w the arcs (u, v_1) and (v_2, w) are split into chains of two arcs by introducing intermediate *train nodes* T_b and T_c and an arc (T_b, T_c). The arcs (u, v_1), (v_2, w) and, (T_b, T_c) models the act of *entering* a train, *leaving* a train and *staying* on a train respectively. By assigning a weight of 1 to all entering arcs and 0 to all others the process of counting the number of transfers on a path is now computable by a simple additive cost function.

2.2.4 Timetables and Time Dependent Networks

One of the many uses of dynamic shortest path problems is to solve the itinerary problem. Given a graph based on a real transportation system and a departure source and time, find an optimal itinerary. Standard tech-

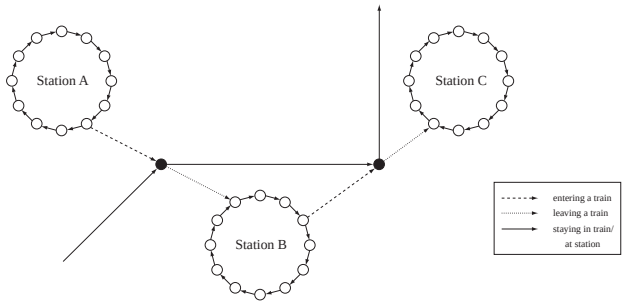


Figure 2.16: Train graph with transfer edges.

niques that use the approaches outlined above are often unacceptably slow for use in their intended application in traveler information systems. Many modern day systems use inexact heuristic solvers that make no guarantees about producing optimal results although they do tend to be fast. In the case where exact results are necessary working with the space-time network model, either implicitly or explicitly, is unacceptably slow.

An approach which moves away from using either space-time networks or train graphs is that of the *time dependent* network [4]. In a time dependent network the properties of an edge are determined by the time at which the link is used. The delay of an edge, an the time at which it is traversable is therefore dependent on the path that is used to reach the edge as well as the departure time.

As in the discrete model of the space-time network above time is modeled as a set of discrete values $\mathbb{T}$, the elements of which are well ordered. In addition, there is a well defined addition function $+ : \mathbb{T} \to \mathbb{T}$.

A *time-table* T is a set of edge connections. Edges are valid for a period

known as the *time-horizon* which is an interval of $\mathbb{T}$. Constructing a space-time network or a train graph from T is a trivial exercise. Events, as before, are arrivals and departures from stations, and the weights on the arcs, for time, are implicitly given by the time difference on the events. Taking train information into account, a proper train graph with transfer information can be easily constructed with arc weights reflecting the cost of a transfer. If the time table needs to model repetition in the time table an edge is introduced from the last event before the end of the period modeled at a node i to the first event at that node.

A *time-dependent network* is a directed graph $G = (V, E)$, where every edge e has a *link-traversal function* $f_e : \mathbb{T} \to \mathbb{T}$ and let $f : \mathbb{T} \to \mathbb{T}$. f has a *non-negative delay* if $f(t) \geq t$ for all t. Also, f is *monotonic* if $t \leq t' \Leftrightarrow f(t) \leq f(t')$.

A *timed path* p in G is a sequence of nodes $(i_1, i_2, \ldots, i_k)$ in the graph and a sequence of times $(t_1, t_2, \ldots, t_k)$, with $t_i \in \mathbb{T}$, where each edge $e = (i_j, i_{j+1})$ is an element of the arc set A and $t_{(j+1)} = f_e(t_j)$. Node i_1 is the departure node and i_k the arrival node. t_k is the arrival time.

This model allows one to use variants of the SPT_S and SPT_L algorithm to efficiently compute the minimum arrival question if there is an efficient way to evaluate the link traversal function and perform comparison on $\mathbb{T}$.

The pseudo-code for MinumumArrivalTime is simplified to demonstrate the efficiency of time dependent networks. As stated earlier, the efficiency of this code depends primarily on the efficiency of implementing

```
MinimumArrivalTime(G = (V, E), f_E : 𝕋 → 𝕋, s, t)
 1  InitializeSPT()
 2  while Q ≠ 0
 3  do (u, t') ← Q.ExtractMin()
 4       Scanned ← Scanned ∪ {u}
 5       for (u, v) ∈ E and u ∉ Scanned
 6       do if C_u > t'
 7            then t'' ← f_(u,v)(C_u)
 8                 Q.Insert(Q, v, t'')
 9                 C_v ← t''
10                 π[v] ← u
```

Figure 2.17: The above algorithm calculates the minimum arrivals to all nodes as it would in a normal SPT problem. The labels C_v are the minimum arrival times taken to reach a vertex v. Given an additional input d for a destination node, the algorithm could be forced to terminate if the search ever tried to expand the spanning tree beyond d. f_E is the link-traversal function for the set of edges E., G the input graph, s the source, and t the time of departure.

the link-traversal function.

2.2.5 Time Dependent Intermodal Networks

Often finding least-time paths on a dynamic network is extended to finding the least-time paths through a series of time-dependent graphs each with

different cost functions. Edges between the sub-graphs represent the time-dependent behavior of transferring, as well as the other costs associated with following an edge from one sub-graph to another. These are known as, *switching costs* . For example, in planning a route from one city to another, a passenger will almost always be forced to consider using many different modes of transportation. Perhaps using public transport to and from major transportation hubs within a city and long distance carriers to move between the cities. From an optimization perspective difficulties arise in computing paths on multimodal networks due to a number of factors. Some of these factors are, the discontinuities of the fixed schedule lines, the fact that modes consist of many transit lines each on its own network and with its own cost functions, the cost of delays when switching between networks and violations of the FIFO rule on links.

A common approach to solving path problems in intermodal networks is to annotate the links with properties. For example one set of attributes might describe the various cost functions, which sub-graph the link belongs to and what type the link might be of. The highly annotated graph is then reduced to a simpler, in terms of composition, but highly expanded graph, much like the reduction of a dynamic graph to the space-time network described above, and then traditional SPT algorithms may be run on the expanded graph. This is seldom practical on real networks however due to the sheer size of the expanded graph in real world networks and the time necessary to compute these networks. Even algorithms that implicitly work on

these graphs are infeasible in practice. In general a $|V|$-node, $|A|$-arc graph with $|M|$-modes and , $|T|$ discrete time intervals will result in a $|V||T||A|$ node network with $|A|(|T|-1)|M|$ arc static network plus $|V|(|M||T|)^2$ to represent switching options at the nodes [36]. A network with 2,000 nodes, 10,000 arcs, 100 modes and fixed schedule lines, and 1,000 time intervals during rush hour could result in a network with $O(10^{13})$ nodes and $O(10^9)$ arcs. Path computation on the expanded network can be carried out, using the traditional SPT methods, or implicitly traversing the expanded graph as in the CHRONO-SPT technique.

Real transit graphs may also have non-FIFO edges. It may be possible to wait at a station for an express, or take a different provider with faster, but perhaps a more expensive service. This means that all of the time replications for a node need to be searched to find the smallest marked time-node that corresponds to the optimal path. The computation effort required drastically increases as the number of time intervals and modes increases. Ziliaskopoulos *et al.* [37] introduce the Time Dependent Intermodal Least-Time Path (TDILTP) algorithm that computes optimum paths without expanding the network. The TDILTP is stated as follows, on the graph G compute the least time paths from every origin node, mode and departure time to a destination node, considering all modes available and the switching costs involved between modes. TDILTP runs in $O(|T|^2|V|^5)$ time. The algorithms computational complexity of this algorithm is independent of the number of modes. The key observation used to reduce the

complexity is that while travel times on different optimum paths may differ many of these paths, for a given origin destination pair are topologically the same. The algorithm takes advantage of this fact by simultaneously updating topologically similar paths.

2.3 Shortest Path Problems in the Presence of Response Time Constraints

The bulk of the literature on speeding up shortest paths utilizes on one of two basic techniques. The first, is reducing the number of computation through guiding the search for shortest paths in order to reduce the amount of scanning that takes place. The disjoint queue methods discussed in section 2.1.1 are good examples of this. The other predominant branch in the literature to speed up the computation of the shortest path tree focuses on choosing an efficient heap structure that implements node selection. The fastest known implementation of Dijkstra's algorithm, using a Fibonacci heap runs in $O(m + n \log n)$ time.

In the presence of these constraints, there are other issues involved in using algorithms, like Dijkstra's algorithm, in practical situations. For example, in any moderately sized graph, a run of a naive Dijkstra may exceed the available space available to the process. There are many situations where space consumption is not an issue but the speed in which a solution is found is of importance. There may be soft real time restrictions, where the av-

erage response time is more important than that maximum response time, which may be the case when designing a centralized query engine to provide routing information to many clients and has to process a large number of queries.

The approach to building *faster* SPT algorithms is usually heuristic. Heuristics often perform well in observing the time constraints but they may not guarantee optimality but instead produce near optimal solutions. Where globally optimal solutions are needed heuristic algorithms are generally avoided. A number of techniques used to improve the response time without losing the optimality of solutions are discussed in [14] and specifically deal with building an advanced traveler information system. Many of these techniques exploit the fact that computing the entire shortest path tree is not necessary to find a route from a source node s to a destination node d. Rather a subset of the shortest path tree that contains the paths is all that needs be generated. Other techniques rely on large scale expensive precomputation of properties on the graph that may be used to speed the search, making it easy to reduce the response time of a query. This level of preprocessing on the graph is acceptable in practice. It may not be possible to store all the precomputed information, as that may take too much space but it may in fact be possible to keep some form of the result to guide future searches.

The first, and most basic of the techniques to speed up computation is *early termination*. Once the shortest path from a s to d has been found

terminate the algorithm.

It may be possible to make safe, but not provable, assumptions about the transit network that yield a considerable speedup. One such assumption might be that the shortest path between two nodes does not deviate too much from the straight line segment from a source node s to a destination node d. In this case only nodes and edges between them that fall within an ellipse with focii at s and d need be considered in the graph, essentially restricting the search horizon [35] of the computation. The ellipse may even change dynamically during computation depending on the intermediate results by alternating and extending the shortest path tree and choosing the ellipse that bounds the supposed area of interest.

Another technique for restricting the explored portion of the transit network relies on the geographical coordinates associated with the stations on a network. In a preprocessing step Dijkstra's algorithm is run from an event to all other stations. For space reasons the actual results are not stored, instead only two values α and β for each edge. There are only $2m$ of these values to store. The values represent angles in the plane. If (v, w) is an edge and s the station associated with event v and s' the station associated with event w then the the values α and β define a circular segment centered at s that contains all the edges that go from s to a s'. Using these precomputed values, if an edge does not lie in the circular segment when computing a path from event v to the station w it may be ignored.

Another interesting approach outlined in [14] is to perform a graph reduc-

tion [32] on the train graph. The technique is common in transport routing applications and is motivated by work on searching massive graphs [1]. Certain stations *transit hubs* are deemed more important that others in transit networks based on their "centrality" in the transport network. Intuitively, these are the stations at which transfers are most likely to occur. Events in the train graph that occur at stations in the set of transit hubs are linked by a directed edge if and only if there is a path in the train graph such that no internal node, non-hub stop, occurs in the path from one hub node to another. In essence, each set of stops on a path between two selected hub events, including the hub events, is a connected component and is replaced in the reduced graph by a directed graph defined on the neighboring selected stations.

The length $l(v, w)$ of an edge (v, w) in the reduced graph is defined as the optimal length path from v to w along a path with no nodes related to transit hubs, in the train graph. Both the reduced graph and the edge weights are constructed in a simple preprocessing step. Each edge in the reduced graph can be annotated with the events (stations and times) along the path it represents in the train graph. This highly compact form can then be used to generate routes from one component to another by performing the search in the more compact reduced graph. Shortest paths in the reduced graph correspond to shortest paths in the train path.

The selection of stations and their events involves some care. There is a trade-off involved. Selecting a small number of stations increases the

size of the connected components and worse still it increases the number of neighboring connected components in the graph. The number of edges is dependent on the number of neighboring components and grows quadratically in the number of stations. The improvement from reducing the number of stations is quickly offset by increasing number of edges. Fortunately, there is a way to curtail the growth in the number of edges between components. If u, v and w are three events for transit hubs and there are edges $(u, v), (v, w), \text{and}, (u, w)$ in the reduced graph satisfying the triangle inequality, $l(u, v) + l(v, w) \leq l(u, w)$ then in constructing the auxiliary graph the edge (u, w) is removed. It is trivial to see that this still preserves the optimality. In the case of transit graphs, the number of edges now grows at a much slower rate than before.

2.4 Nonadditive Cost functions in Shortest Path Problems

Among the differing types of cost functions found in solving with routing problems through networks are non-additive cost functions and nonlinear cost functions. These arise naturally in many situation, routing through transit networks, through telephone networks, flow control in water pipes and in abstract areas such as scheduling activity on networks and evolution in organizational networks. In addition to being non-additive, path costs may also be non-linear. For example in dealing with the notion of 'travel-

time' in a transportation network it may be unreasonable to expect that a traveler has a single fixed value for time. It is far more realistic that a traveler places little value on small savings of time and a far greater value on larger savings of time. At first glance this may not seem to be too large an issue but where the costs are non-linear functions of that nature, it frequently leads to result in the Bellman's conditions for optimality to be violated.

2.4.1 Nonadditive cost functions.

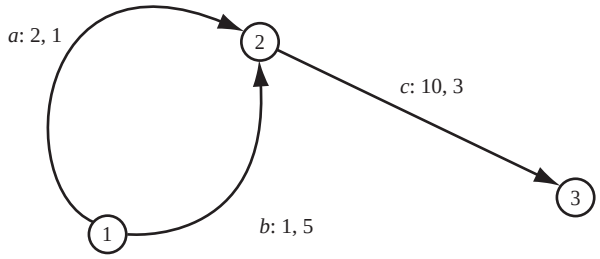


Figure 2.18: A simple network with Nonadditive paths.

In the 3-node 3-link network shown above[30], the first number next to each link represents the time and the second a toll. The aggregate cost on a path is given by the time squared plus tolls. Observe that the cheapest path between nodes 1 and 3 uses arcs b and c. The cost on this path is \$129 vs \$148 on the path that uses arcs a and c. Bellman's Principle, would guide us to using arc b as part of the cheapest path from node 1 to 2 but this is not the case. The cost on arc a is \$5 and the cost of arc b is \$6, the minimum cost from node 1 to 2 is actually arc a.

Because the Bellman's conditions are violated, solving non-linear or non-

additive problems using direct labeling techniques is no longer possible. One obvious way to address this problem is to resort to 'brute force' approaches. A search could find the minimum time path and see if it checks for tolled links. If a tolled link is found, the link can be removed and the algorithm re-run. The process is repeated until all the toll-free minimum paths are removed from consideration or the inclusion and exclusion of all tolled links in the network have been considered. If there are T tolled links in the graph then there are 2^T paths, some of which include tolled links that have already been considered in prior iterations. Other direct approaches can be considered to solve the above problem, many of which are computationally expensive. Another way to tackle the problem is as an integer program using *branch and bound* techniques.

2.4.2 The Inherited Constraint Algorithm

In [30] an inherited constraint algorithm is presented that solves problems in this category. Their method, while still non-polynomial in the worst case manages to find optimal solutions to many real world problems in only a few iterations.

Once again $G = (N, A)$ is a network of arcs but this time with a cost function $c : A^n \rightarrow \mathbb{R}$ which associates a cost with the vector of arc flows

$x \in A^n$. $A = 1, \ldots, n$ is the arc incidence matrix defined as follows:

$$a_{ij} = \begin{cases} 1 & \text{if arc } j \text{ is directed out of node } i \\ -1 & \text{if arc } j \text{ is directed into node } i \\ 0 & \textit{otherwise.} \end{cases} \tag{2.5}$$

Let $c = (c_j : j = 1, \ldots, n)$ be the cost of all arcs in the graph and assume the path cost are additive. Thus the cost of a path is given by

$$C(x) = c^T x \tag{2.6}$$

Thus the minimum cost shortest path problem can be formulated as follows:

$$\begin{array}{ll} min_x & c^T x \\ s.t. & Ax = b \\ & x \in \{0,1\}^n \end{array} \tag{2.7}$$

Assume now that the cost of a path corresponding to x is given by

$$C(x) = v(t^T x) + \tau^T x \tag{2.8}$$

where $t \in \mathbb{R}^n_+$ and $\tau \in \mathbb{R}^n_+$ are the vectors of 'times' and 'tolls'. $v : \mathbb{R}^n_+ \to \mathbb{R}_+$ denotes a cost function for the *value of time*.

$$\begin{array}{ll} \min_x & v(t^T x) + \tau^T x \\ s.t. & Ax = b \\ & x \in \{0,1\}^n \end{array} \tag{2.9}$$

This is known as the *ICA* iterative process.

When $v(x) = t^T x$ this is a simple minimum cost path problem with a composite cost vector $c = t + \tau$

To consider the case where v is non-linear assume that there is a solution to (2.9) above denoted by x^*. x^* is a solution of Eq.(2.9) if and only if it is a solution of

$$\begin{array}{ll} \min_x & v(t^T x) \\ s.t. & Ax = b \\ \tau^T x & = \tau^T x^* \\ & x \in \{0,1\}^n \end{array} \tag{2.10}$$

This implies that x^* is a solution of Eq. (2.9) if an only it is a solution of

$$\begin{array}{ll} \min_x & t^T x \\ s.t. & Ax = b \\ \tau^T x & = \tau^T x^* \\ & x \in \{0,1\}^n \end{array} \tag{2.11}$$

Their is no nice solution to Eq. (2.9), since if there were, Eq. (2.11) would be of no importance. However educated guesses for x^* might allow the design of an algorithm that approaches an optimal solution.

To understand this better let Z denote the value of $\tau^T x^*$, the toll on the

optimal path. The problem then becomes:

$$\begin{aligned} \min_x \quad & t^T x \\ s.t. \quad & Ax = b \\ \tau^T x \quad & = Z \\ & x \in \{0,1\}^n \end{aligned} \tag{2.12}$$

The first guess, say x^0, should be pessimistic to avoid overlooking any potential solutions. This can be accomplished by ignoring the toll constraint and solving for the minimum path time. The solution is illustrated in the figure above. Since x^0 is the minimum time path, there are no feasible solutions to the left of x^0 on the plane. Also, solutions above x^0 on the plan are dominated by x^0 since they have a larger time, or a larger toll both of which imply a larger composite cost. Therefore one only needs to consider solutions below and to the right of x^0.

In order to generate a set of these solutions consider the following problem.

$$\begin{aligned} \min_x \quad & t^T x \\ s.t. \quad & Ax = b \\ \tau^T x \quad & < Z^j \\ & x \in \{0,1\}^n \end{aligned} \tag{2.13}$$

This generates the set of solutions that have tolls less than that on x^0.

Since a strict inequality cannot be represented directly the equations

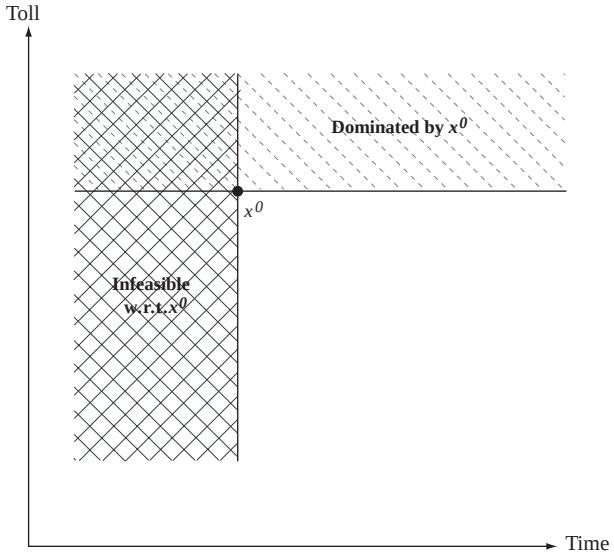


Figure 2.19: The initial solution makes a pessimistic assumption to avoid overlooking any potential solutions.

become:

$$\begin{aligned} \min_x \quad & t^T x \\ s.t. \quad & Ax = b \\ \tau^T x \quad & \leq Z^j - \epsilon \\ & x \in \{0,1\}^n \end{aligned} \tag{2.14}$$

Where ϵ is an arbitrarily small difference in the tolls on the network. This is the CSP_j problem.

The process iterates producing minimum time paths under increasingly strict toll constraints inherited from the previous iteration; each solution dominating other paths in the restricted search space with respect to time. When a path is found, with a toll equal to the minimum toll on the network

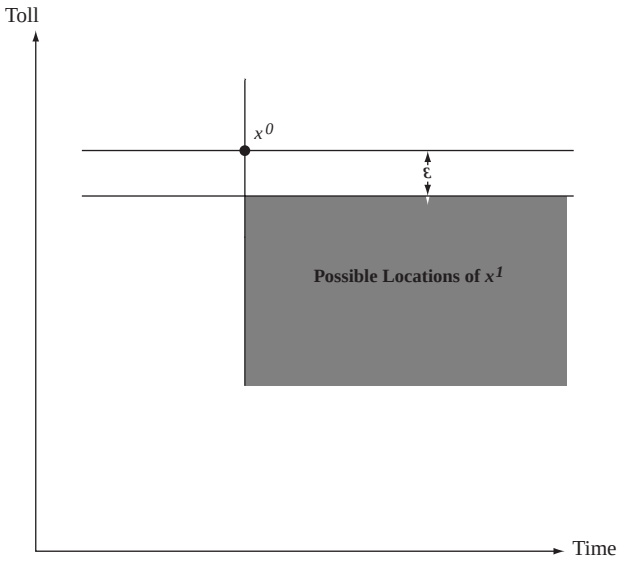


Figure 2.20: ϵ is introduced to rewrite the inequality as a strict inequality.

the algorithm can terminate and evaluate the objective function in (2.9) using each of the candidates to produce an optimal solution.

In general the iteration process described still may take a great many iterations to find the solutions but in many cases the performance can be dramatically improved by recognizing when it is possible to terminate the search early. By constructing a set of time-toll pairs that have the same composite cost as x^j, defined by:

$$I(x^j) = \{(a,b)|v(a) + b = v(t^T x^j) + \tau^T x^j, a \geq t^T x^j, b \geq \tau^T y^0\}, \quad (2.15)$$

where $\tau^T y^0$ is the minimum toll on all paths between the origin and the destination.

Using the fact that minimum toll on all paths may not be zero, y^0 can

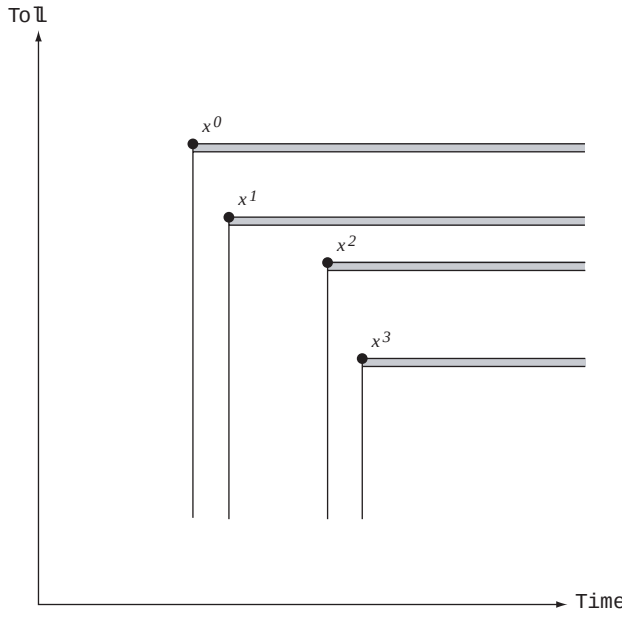


Figure 2.21: ICA Iterations

be found by solving the minimum toll path problem:

$$\begin{aligned} \min_x \quad & \tau^T x \\ s.t. \quad & Ax = b \\ & x \in \{0,1\}^n \end{aligned} \tag{2.16}$$

Let $\bar{t}(x^j)$ denote a point in $I(x^j)$ with the maximum time given by:

$$\bar{t}(x^j) = max\{a|(a,b) \in I(x^j) \text{ for some } b\} \tag{2.17}$$

If a solution x_{j+1} with $t^T x_{j+1} \geq \bar{t}(x_j)$ then the algorithm can terminate since $x_{(j+1)}$ dominates all subsequent candidate solutions with respect to time.

Naturally, it may be that this technique does not reduce the space at all, i.e. when $\bar{t}(x^j) \geq t^T y^0$

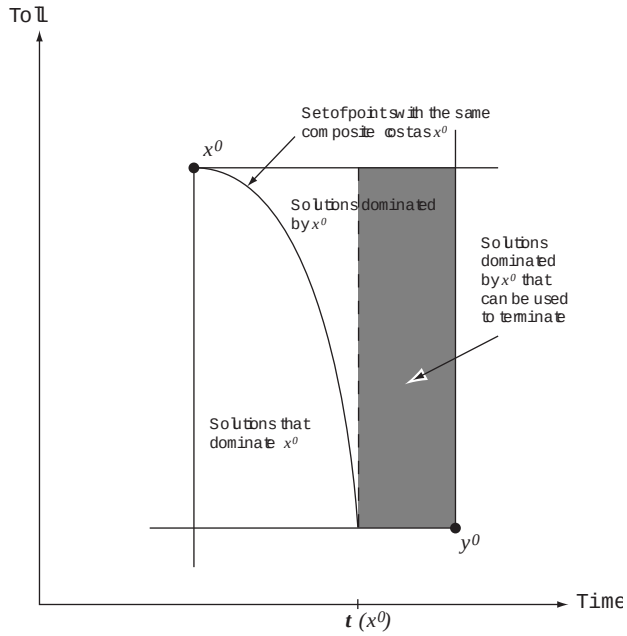


Figure 2.22: Terminating the algorithm.

In this case it may be possible to work from y^0 to reduce the search. To do so a mirror of the CSP_j problem is formulated and used repeatedly to solve a minimum toll problem subject to a time constraint.

At any given iteration the search space may not be reduced, although in practice it often is and the worst case complexity of this inherited constraint algorithm is the same as that of a brute force method, i.e. super-polynomial.

2.5 Multi-criteria Optimization in Graph Problems

The field of multi-criteria optimization has its roots in late nineteenth-century welfare economics in the works of Edgeworth and Pareto. Single

criterion optimization problems, such as the single source shortest path problem discussed above, often yield single solutions. In the SSSP the single solution may be the fastest path, cheapest path, shortest path, etc. where the single solution is the result of optimizing on a single criterion. Unfortunately, when arcs in a graph are labeled with two weighting functions, such as *cost* and *time*, there may not be a single solution that is both the *cheapest* and the *fastest*. In fact, the notion of "optimality" does not apply in the multi-objective domain directly. Solving a multi-objective optimization problem usually involves finding multiple *efficient* solutions, the Pareto optimal set, with with the property that one of the criteria on which a solution was found may be improved but only at the cost of the other criteria.

2.5.1 Pareto Optimal Solutions

Formally, the multi-criteria optimization problem is stated as being the problem of finding a vector of decision variables $\bar{x}^* = [x_1^*, x_2^*, \ldots, x_n^*]^T$ which optimizes the vector function $F(\bar{x})$ the elements of which are the objective functions.

$$F(\bar{x}) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{pmatrix}$$

where $n \geq 2$, over the space S that represents range of the decision

variables. The goal is to find the set of $x_1^*, x_2^*, \ldots, x_n^*$ which yields optimum values over all the objective functions.

A point $\bar{x}^* \in S$ is *Pareto optimal* if for every $\bar{x} \in S$ either

$$\wedge_{i \in I}(f_i(\bar{x}) = f_i(\bar{x}^*),$$

or, for at least one $i \in I$ such that.

$$f_i(\bar{x}) > f_i(\bar{x}^*).$$

That is, $\bar{x}^*$ is Pareto optimal if there exist no other vector $\bar{x}$ that would decrease some criterion without simultaneously increasing the value of another. The set of points in the space S that is Pareto optimal is known as the Pareto optimal set and it is rare that this set contains only a single solution. Solutions s that belong to S are known as *efficient*, *non-inferior* or *non-dominated* solutions.

Locally Pareto optimal points are a weaker set of optima, for which the definition is the same as the one just given, except that the optimality is restricted to a feasible neighborhood of $\bar{x}^*$ That is if $B(\bar{x}^*, \delta)$ denotes a ball of radius δ around the point $\bar{x}^*$ then there is no $x \in S \cap B(\bar{x}^*, \delta)$ such that $f_i(x) \leq f_i(\bar{x}^*)$ with at least one inequality.

The multi-criterion optimization problem, is in general, NP-Hard. It is also worth noting that the solution set to multi-criteria optimization problems can contain exponentially many points. Multi-criteria solution sets can grown exponentially with the number of criteria as well as exponentially with the problem size even when the number of criteria is fixed [29]. It is

this factor that decides the tractability of a problem. Moreover, the number of solutions grows exponentially over all the visited nodes in a graph and this is crucial in deciding whether an algorithm can in fact compute the Pareto optima for a given problem.

In some cases the multi-objective optimization can be solved using *pseudo-polynomial time* algorithms. An algorithm runs in pseudo-polynomial time if it solves any instance in time polynomial in the size of the instance and in the value of the largest integer in the instance description. The existence of a pseudo-polynomial time algorithm is necessary when considering approximation approaches [22]. Interestingly if $P \neq NP$ this condition is not sufficient, in fact, for multi-objective optimization problems a fast approximations scheme exists only if a *value-pseudo polynomial* VPP algorithm exists [29].

2.5.2 The Bicriterion Shortest Path Problem

The bicriterion shortest-path problem, one of the simplest of the multi-criterion optimization problems, is defined as follows. Let $G = (N, A)$ be a simple directed graph where there are two objective functions c_{ij} and w_{ij} associated with each arc $(i, j) \in A$. Call c_{ij} and w_{ij} the *cost* and *weight* of an edge, respectively. Let p be the set of arcs on a path between a given origin and destination and $F(p)$ and $G(p)$ be the cost and weight of the the arcs on the path p.

$$F(p) \quad = \quad f(\{c_{ij} | (i, j) \in p\}) \tag{2.18}$$

$$G(p) \quad = \quad g(\{w_{ij} | (i,j) \in p\}) \tag{2.19}$$

Let P_{ij} is set of all paths from from i to j. A path $p \in P_{ij}$ is a dominated path if there is a path $p' \in P_{ij}$ such that $F(p) \geq F(p')$, $G(p) \geq G(p')$, and at least one inequality holds, otherwise p is non-dominated. The *Bicriterion Shortest Path* (BSP) is to find the set $P*_{ij} \subseteq P_{ij}$ formed by all the non-dominated paths from i to j.

In general the bicriterion shortest path problem is NP-hard [17]. However in certain cases, depending on the functions f and g, the problem can be solved in polynomial time.

In some cases it is even possible to solve the BSP problem with exact methods. In the bicriterion analog to the SSSP, i.e. find all the *optimal* paths from a root node r to all other nodes $i \neq r$ in the graph if for one of the measures, say the path weight, the list of all possible path weights, say $\mathbf{G} = \{G_1, G_2, \ldots, G_k\}$ is known *a priori* and $G_j < G_{j+1}$ for $j = 1, 2, \ldots, k-1$ then with these constraints the problem can be solved using a pseudo-polynomial time algorithm.

Associate k different labels with each node $i \in N$. $C_i(G_j)$ is the current minimum cost of the paths from r to i with weight G_j. If one keeps a set $\mathbf{C}_i$ of the non-dominated labels associated with i then for every pair of non-dominated labels for a given node i, say $C_i(G_j)$ and $C_i(G_k)$

$$G_i < G_k \Leftrightarrow C_i(G_j) > C_i(G_k) \tag{2.20}$$

Based on the set of labels $\mathbf{G}$ and the concept of dominance, there are two different approaches to solving BSP, the first a *multi-labeling* approach,

the second a *topological* approach.

A Multi-Labeling approach. Without loss of generality let the functions F and G in Eq(2.18) above be simple summations of the path costs of the edges and to further simplify the argument here let the arc labels f and g always be non-negative. Given a node, suppose the non-dominated labels associated with this node are listed according to one index, say the 'weight' index, and then the other. Due to the non-dominance condition for entry into this set, these labels are also ordered but in the reverse order.

When a node i is selected and the leaving arc (i, j) is analyzed, each candidate label $C_i(W_h) + c_{ij}$ is compared with the label $C_j(W_h + w_{ij})$. Since w_{ij} is constant for the arc the weight of non-dominated labels for node i is shifted with respect to the weight index and the labels are increased by the same value c_{ij} The candidates still follow the dominance ordering property 2.20 and can be *merged* with C_j producing a new list of non-dominated labels (dominated labels are removed during the merge).

A Topological Approach. This approach is an adaptation of the CHRONO-SPT which maintains a set of non-dominated labels. Here a bucket list $\mathbf{B}$ stores the sets of non-dominated labels, with the h^{th} bucket B_h storing the set of non-dominated labels $C_i(W_h)$, for $i \in N$. At each step, the current label is selected from the lowest non-empty bucket (weights are considered according to their non-decreasing ordering) and is processed in a manner similar to that described in the CHRONO-SPT. The topological approach finds the non-dominated paths according to the non-decreasing ordering of

the path weights.

In transportation algorithms the topological approach is very efficient. For example, given a network with arc costs, the problem is to find the minimum cost paths from a given origin to other nodes subject to the constraint that the number of transfers does not exceed a maximum given number, say k. Here the 'transfer number' is the weight associated with the arcs, transfer arcs have a weight of 1 and non-transfer arcs a weight of 0. The number of transfers, limited to k is used as the index both for the labels and for the buckets. When the labels are selected from the h^{th} bucket $B_h, 0 \leq h \leq k$ all the previous buckets have been emptied while the buckets $B_{h+2}, \ldots, B_k$ are still empty. Then, the wrap-around handling of the bucket list limits their physical number to two. It is therefore possible to implement the bucket list as two lists, Q_{now} and Q_{next} which are B_h and B_{h+1} respectively. As in CHRONO-SPT dominance is checked relative to the last selected non-dominated label relative to node i, the *last-label*$_i$.

A typical iteration selects a node i from Q_{now} with its corresponding label $C_{i-\text{now}} = C_i(W_h)$. If $C_{i-\text{now}} \geq$ *last-label*$_i$, then it is discarded since it is dominated. Otherwise for each arc in the forward star of i that is not a transfer arc the candidate label $C_{i-\text{now}} + c_{ij}$ is compared to $C_{j-\text{now}}$. If $C_{i-\text{now}} < C_{j-\text{now}}$ then $C_{i-\text{now}}$ is decreased and if j is not in the *now* queue it is added to it along with its label. In the case where (i, j) is a transfer arc then $C_{i-\text{now}} + c_{ij}$ is compared to the $C_{j-\text{next}}$ and when necessary inserted into Q_{next}.

When Q_{now} is empty all the minimum cost paths with no more than h transfers have been found and the algorithms swaps Q_{now} and Q_{next} and processes the non-dominated labels relative to the $h+1$ transfer number. When $h = k$ and Q_{now} is empty the algorithm will terminate. The time complexity is $O(mk)$.

Finding Pareto shortest paths is often feasible in practice. Often problems involving graphs with multiple-optimization functions based on real world problems produces a small number of Pareto optima [24], this is in sharp contrast to the worst case. Many real world graphs produce a total number of Pareto optima that is small enough to make the problem efficiently tractable from a practical standpoint.

Chapter 3

The Itinerary Problem

The impetus for this research came from trying to generate intelligent multi-modal itineraries for passengers traveling on fixed route transit systems. While it has been possible to plan itinereraries using airfares, and choose fights using mulitple criteria, i.e. cost and time constraints, this service has not been available for planning trips using public transports. Flight planners, until recently, were run on large mainframes. The Orbitz system was the first to move away from the mainframe model and move to serving customers using clusters of Pentium class computers [10]. Itinerary planners for public transportation deal with sparser but far larger graphs than airline planners and until recenty, prior to the spread of the World Wide Web, had generated no interest or demand. Other intelligent passenger advisory systems (IPA) for the public transport domain, like the one described in this thesis, have been built both in academic settings and by commercial software companies. Of the more interesting IPA systems deployed already are

the TransStar system built by Transcom, and Hafas built by HaCon Ingenieurgesellschaft mbH. The Transcom system, commissioned by the State of California to offer a multi-modal itinerary system for the transit systems in Southern California is now being deployed in a number of cities around the United States. The Transcom system, while being multi-modal optimizes journeys based on time only, searching only for the shortest way to get a passenger from one location to another. HaFas users are Railtrack, a company that runs the railway infrastructure in Great Britain and DeutcheBahn, the German railway provider. Much of the work behind the HaFas servers is documented in [14, 35, 24]. This system currently provides multi-modal itineraries for most European railway networks. While the HaFas system includes price information, it does not currently optimize on the fares of journeys taken. To date, none of the systems that the author is aware of have attempted to solve the Pareto optimum paths problem for large scale dynamic networks using exact methods.

The problem to be solved here is that of providing reasonable itineraries to a user of an IPA system. The goal is to present the user with all "good" solutions to the question, "How do I get from A to B if I leave A at time T?". The assumption made in designing this system is that the Pareto optimal set represents all "good" solutions to a user query. By optimizing simultaneously on the time of journey, the cost of the journey and the directness of a journey (the number of vehicle changes involved in going from A to B) a "good" solution is a Pareto optimal path through the transit systems

Option	Carrier	Route	Time	Fare
1	Amtrak	New York → Philadelphia	1h 50 m	$85
2	Commuter Rail	New York → Trenton → Philadelphia	2 h 12 m	$14
3	Bus	New York → Philadelphia	2h 20 m	$20
4	Limo	New York → Philadelphia	2h 30 m	$195

Table 3.1: The travel time and fare for traveling from New York City to Philadelphia.

from A to B and no non-Pareto optimal path is ever returned to a user of the system. Consider the following example, if a passenger tries to get from New York City to Philadelphia using public transport the choices are currently, take an Amtrak train directly from Penn Station in New York City to 30th Street Station in Philadelphia, take a bus from the Port Authority to Philadelphia or, travel from Penn Station to Trenton on New Jersey Transit, the New Jersey regional commuter line, and then continue on from Trenton to Philadelphia via SEPTA, (South-Eastern Pennsylvania Transit Authority) the Pennsylvania regional commuter line. The costs and travel times are listed in figure [3].

Clearly, a case can be made for using all of the first three solutions, depending on whether cost, time or directness was important to a user. No passenger would opt for the fourth solution, which takes longer than the Amtrak and costs more, if they were to base their decision on the three objective functions, cost, duration of journey and directness.

Unfortunately, Pareto optimal paths are expensive to compute. As stated earlier multi-criteria optimization problems are NP-Hard, in fact even the simplest of the multi-criteria optimization problems, the bicriterion optimization problem is NP-Hard. The approaches taken in solving multi-criteria optimization problems are described in chapter 2. The most favored approach, due to its computational efficiency, is to solve the Pareto shortest paths problem using heuristics but in many cases heuristic solvers are not acceptable as they are not guaranteed to compute the correct solution set. Fortunately, computing Pareto paths using exact methods (that is methods that are guaranteed to generate the set of solutions) is possible in practice [24]. When considering the Pareto optimal path problem, it is possible to construct artificial networks where the number of Pareto paths found to every vertex is exponentially large in both the size of the input graph and the number of criteria being optimized, however real world graphs often have built in properties that cause the number of Pareto paths to be small. For example, the nearly FIFO nature of transit networks implies that if a train runs along a track it seldom, if ever, catches up with the train that left before it. This ensures that when constructing solutions, waiting at a node will seldom generate a new solution that outperforms leaving earlier.

This thesis describes the algorithms and key implementation features that will solve the problem of generating Pareto paths in large scale dynamic networks using the Itinerary Problem as a device to explore the issues involved. Given the problem is NP-Hard but is tractable in practice the design

goals are that the algorithms should be able to present a user with answers in a short period of time on commodity hardware.

In generating itineraries a key issue is that the networks being represented are multi-modal. Previous work [36, 37] in modeling multi-modal networks create an exponential explosion in the size of the graph. The dynamic graph that represents the transit systems of the north-east corridor represents 77 different transit systems from long range passenger trains to local subways and ferries. This is the largest multi-modal transit network modeled. The graph has over 3000 stops and 3.5 million edges connecting them.

The strategy adopted in this work is to create a dynamic multi-graph that models the properties of each transit provider and then adds transfer arcs to connect the components together. Computing an itinerary is now a matter of finding the set of Pareto optimal paths in this multi-graph. The resulting multi-modal graph is roughly the size of the component graphs, with only a handful of edges added to represent the change between modes.

This simple strategy has a number of advantages over most of the other approaches in the literature these will be outlined below as the dynamic multi-graph structure is formalized.

3.1 The Pareto Itinerary Problem

The itinerary problem that this thesis aims to solve is the Pareto itinerary problem.

Let $G = (V, E)$ be a dynamic multi-graph where V is the set of vertices

and E the set of edges representing a single vehicle that goes between two stops stopping at no other stop along the way.

In this problem all time values are discrete, along the lines of [6, 27, 37], and belong to the set $\mathbb{T}$. $\mathbb{T}$, in this problem, is equivalent to the set of positive integers $\mathbb{Z}^+$.

The set of vertices V is partitioned into disjoint subsets called *fare regions* that represent the different fare structures associated with a journey across transit systems. For example the best way to go between two cities using public transport may be to use a mass transit system to get to a major station, an inter-city long distance carrier to the destination city and then another mass transit system to finally reach the destination. Each carrier has its own fare structure and path costs under these fare structures are seldom based on a simple additive cost function. Let FR be the set of fare regions and V_i be the set of vertices in fare region i then,

$$V = \bigcup_{i \in FR} V_i$$

Each edge $e \in E$ describes a segment of a journey between two stops. More formally the edge e can be written as $(u, v, t^{\text{DEP}}, t^{\text{DUR}}, transport)$ where the u and v are the source and destination vertices, t^{DEP} and t^{DUR} represent the time of departure along that edge and the duration of the journey respectively, and finally $transport$ is an identifier for the vehicle that the edge represents.

For an edge $e = (u, v, t^{\text{DEP}}, t^{\text{DUR}}, transport)$ we define the following functions:

- $src(e) = u$
- $dst(e) = v$
- $departure(e) = t^{\text{DEP}}$
- $duration(e) = t^{\text{DUR}}$
- $arrival(e) = t^{\text{DEP}} + t^{\text{DUR}}$
- $transport(e) = transport$

Since the vertex subsets representing the fare regions are disjoint, special edges known as *transfer edges* are added to E to connect the fare regions together. Transfer edges may also be added between two stops in the same fare region. When a transfer edge has a source and destination vertex in different fare regions and the vertices u and v are known as *transfer points*.

Transfer edges have the property that they may be traversed at any time, that is the t^{DEP} for each transfer edge is any value in the set $\mathbb{T}$. The departure time for a transfer edge is represented by the special value $\square$. If u is a transfer point then the set of *transfer times* T is the set of arrival times of non-transfer edges to u. If u and v are transfer points and $(u, v, \square, t^{\text{DUR}}, transport)$ is a transfer edge it is plain to see that a transfer edge could be rewritten as a set of normal edges in the following manner,

$$(u, v, \square, t^{\text{DUR}}, transport) \quad = \quad \bigcup_{t \in T} (u, v, t, t^{\text{DUR}}, transport)$$

Many timetables are *periodic*, that is all information about the transit systems repeat after a set time interval. The *time-horizon* of the timetable

is the value of time at which the timetable repeats itself. For example if the time values in $\mathbb{T}$ are meant to represent minutes, and the timetable is a weekly timetable then the time horizon is 10080, the number of minutes in a week. To allow paths to be computed where the departure times or arrival times are greater than the time-horizon the value of t^{DEP} of an edge e is actually the remainder of the actual time of departure $\tilde{t}^{\text{DEP}}$ modulo the time horizon.

$$t^{\text{DEP}} = \tilde{t}^{\text{DEP}} \text{mod } \textit{time-horizon}$$

Let a path p in the graph G be a sequence of connected edges $\langle e_1, e_2, \ldots, e_n \rangle$ in the G. The *arrival time* of p is given by the arrival time of the last edge in the path. For an edge to extend a path the edge must have a departure time that is at or after the arrival time of the path. To illustrate this point further, given a path $p = \langle e_1, e_2 \rangle$ if $\textit{departure}(e_1) + \textit{duration}(e_1) > \textit{departure}(e_2)$ the departure time of e_2 is $\textit{departure}(e_2) + \textit{time-horizon}$.

For a path $p = \langle e_1, e_2, \ldots, e_n \rangle$ we define the following functions:

- $src(p) = src(e_1)$
- $dst(p) = dst(e_n)$
- $departure(p) = departure(e_1)$
- $arrival(p) = arrival(e_n)$

The cost functions on which a journey is optimized are the duration of journey f^{T}, the monetary cost (fare) for a journey f^{C}, and the directness of the journey f^{XFR}.

Definition 3.1.1 *The* starting time of an itinerary t^{START}, *is the time after which the first edge of any solution found may start.*

Given a path p through the graph, the costs functions are defined as follows.

Definition 3.1.2 *The* duration of the path $f^{\text{T}}(p)$, *is given by the difference between the time of arrival of the path and the starting time of the itinerary.*

$$f^{\text{T}}(p) = arrival(p) - t^{\text{START}}$$

Definition 3.1.3 *The* directness of the path, $f^{\text{XFR}}(p)$ *is a measure of how many different vehicles are used along a path p. This is simply a count of the number of distinct transports that occur when traversing a path using non-transfer edges. Note that transfer edges do not contribute to the value of* $f^{\text{XFR}}(p)$.

Almost all transit providers publish a fare table that determines the cost of every journey on their transit system. Usually, the fare is based on the source station s, the destination station d, and the time at which the journey was taken. This time-dependence is mainly due to peak and off peak fare structures common to many mass transit systems.

Definition 3.1.4 *The* intra-regional cost *of a path p in a fare region i is given by* $f_i^{\text{C}} : V \times V \times \mathbb{T} \to \mathbb{R}$ *and is dependent only of the source and destination in that fare region and the time at which the first edge within the fare region is departed.*

$f_i^{\mathrm{C}}(p)$ is implemented by means of a lookup function.

Let $p = \langle p_1, e_{(1,2)}, p_2, \ldots, e_{(n-1,n)}, p_n \rangle$ where p_i and p_j are path segments through fare region i and j respectively and $e_{(i,j)}$ is a transfer edge from fare region i to fare region j.

Definition 3.1.5 *If a path p travels through multiple fare regions the* inter-regional cost *of the path, $f^{\mathrm{C}}(p)$ is given by the sum of the costs of the paths through each of the fare regions.*

$$f^{\mathrm{C}}(p) = \sum_{i \in I} f_i^{\mathrm{C}}$$

where I is the set of fare regions through which p passes.

Of these cost functions only one, the directness of the journey, f^{XFR} is linear and additive.

Definition 3.1.6 *Let P_v be the set of Pareto optimal paths that reach a vertex v and P be the set of all Pareto optimal paths.*

We can now define the Pareto itinerary problem.

Definition 3.1.7 *Given a multi-modal dynamic network $G = (V, E)$ and set of source vertices $S \subseteq V$, and destination vertices $D \subseteq V$ and a starting time t^{START} for a journey and an vector function $F(p) = [f^{\mathrm{T}}, f^{\mathrm{C}}, f^{\mathrm{XFR}}]^{\mathrm{T}}$ the* Pareto itinerary problem *is to find the set of Pareto optimal paths P from S to D leaving at a time t^{START} in G that is dominating in the vector function $F(p)$.*

The Pareto itinerary problem is NP-Hard, as explained earlier. If the Pareto itinerary problem were to optimize only on two objective functions, $F(p) = [f^{\mathrm{T}}, f^{\mathrm{C}}]^{\mathrm{T}}$ the problem is immediately the bicriterion shortest path problem and remains NP-Hard.

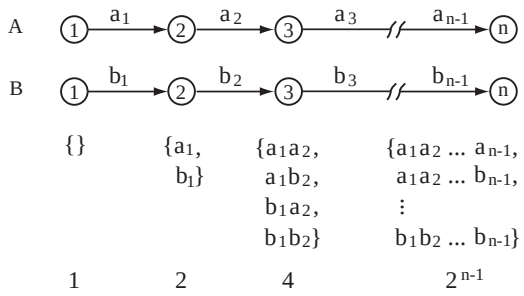


Figure 3.1: Identical edge sets lead to an exponential number of solutions. As the path grows each pair of edges doubles the number of solutions at the previous node.

The complexity of the problem is also easy to illustrate by example. Consider a graph G that has pairs of identical edges between each vertex as illustrated in figure 3.1. Assuming the start vertex is 1 the sets of undominated paths at each vertex is shown below each vertex. The number of non-dominated solutions grows exponentially as the path is extended. In practice however the dynamic network that models transit systems will have properties that cause the number of non-dominated paths at each node to be small.

Now that we have defined the itinerary problem we can turn our attention to finding efficient ways to generating Pareto optimal itineraries. While the

problem remains NP-Hard we can in practice solve it for our real network. In the next chapters we will see how to solve the Pareto problem in the framework of the general Bellman-Ford-Moore algorithm, extending it from a single objective shortest path tree problem to a more general Pareto set problem. The domination of paths is an important factor in keeping the practical running time low. We will see how to expand Pareto optimal paths from a vertex to its neighbors and then apply a domination check, based on $F(p)$ to reduce the number of optimal paths at any given vertex. We will also show how to extend $F(p)$ to solve problems that while not typical optimization problems, are of interest to us.

Chapter 4

Computing Pareto Paths

In this chapter we make a first attempt at solving the itinerary problem by modifying the Bellman-Ford-Moore algorithm to simultaneously optimize solutions based on the objective functions for the duration of a path p, $f^{\mathrm{T}}(p)$ and its cost $f^{\mathrm{C}}(p)$. This basic algorithm is in many ways, analogous to the ChronoSPT algorithm from chapter 2. We then describe how to add the directness measure $f^{\mathrm{XFR}}(p)$ to the vector of objective functions.

In order to construct the ParetoPath algorithm we must first describe the dynamic network on which it is run and how it is constructed from the various sources of transit data. Since the problem is to be solved on large scale networks, once the graph grows large enough, resulting from adding many transit provider, it can no longer be stored in main memory, instead, the graph is stored in a database. In the discussion that follows we shall simply refer to this as "the database". The database that describes the graph may reside in main memory or on disk, depending on whether there

is enough room in main memory. Where it is important, we will discuss the impact of disk performance on the algorithm. To implement the database we use the B-tree data structure from the BerekleyDB system, this system is embeddable in the program so performance of the algorithm can be analyzed without needing to factor in the database management overhead of other more relational systems.

4.1 Building the Dynamic Network from Transit Data

The dynamic multi-graphs used to represent the intra-regional transit networks are built from three different sources of information, transit maps, timetables and fare tables. The information from the timetables is used by the time and transfer cost objective functions, f^{T} and f^{XFR}. The fare tables and the timetables are used to create lookup tables for the time dependent cost function f^{C}.

The vertex set for each intra-regional network is determined by examining the set of transit maps that each provider publishes. Every station on a transit providers map becomes a vertex in the transit network. Every vertex is also geocoded, i.e. there is a longitude and latitude associated with every stop, and this information, while not central to the algorithms presented below, is used to determine the set of source and destination vertices. Geocoding information is also useful for heuristics, like those described in [35], to guide the search through the network.

The edge set in the intra-regional network is determined from the timetable

information published by each provider. As described earlier in chapter 3 every edge represents a segment of a journey on a particular vehicle. Two consecutive timetable entries, say for a train A departing stop u at time t_1 and then departing a stop v at time t_2 represents an edge that leaves a stop u to a stop v, leaving at time t_1 with duration $t_2 - t_1$. Under ideal circumstances this is all that needs to happen to derive an edge set. However, anyone who has ever looked at a timetable will know that there is often a plethora of additional information on every timetable. For example, in some cases a train will stop only to drop off a passenger, whereas in other cases a train will stop only to pick-up passengers. Modifications to the edge set based on constraints imposed by the transit providers on the use of their system is often necessary and part of a post-processing phase. In addition to this additional information, transit providers present their timetable information in different ways. In reality conversion of timetable information into an edge set is largely an *ad hoc* process that cannot as yet be performed automatically.

The fare function f^{C} is based on creating look-up tables from the transit providers fare tables. This is because there are no fixed rules as to how individual providers decide to charge for a journey. In the case of some mass transit systems, like the subways and buses in New York fares are flat fares, in other cities, mass transit systems base their fares on track miles traveled, and in many cases fares are determined on a zone-to-zone basis.

The approach taken here is to model all fares for every provider as a

point-to-point fare function. The stop set is partitioned into disjoint subsets of *fare zones* and the fare zones of two stops is used to determine the fare structure. Where the providers fare structure is based on a zoning system the fare zone of a stop is given by the published information. In the case of flat fare systems like the subway all the stops are placed in a single fare zone and the fare table has a single entry. Lastly, if the fare structure of a transit provider is based on a distance function each stop is assigned to a unique fare zone.

The fare function is also time dependent. Thus the fare table is indexed on the source fare zone, destination fare zone and the time of journey. This is due to *peak* and *off peak* fares on many commuter systems. Peak fare structures are often a result of policy decisions made to reduce the number of riders during rush hours. Often peak fares are asymmetric. Coming into a city center stop during the morning rush may carry an additional surcharge whereas leaving a city center during the the morning rush hours does not carry a surcharge. Leaving a city center during evening rush may also carry a surcharge. The fare tables, when they are constructed, need to reflect this behavior. Consider the following example, on the NYC MTA Metro north service to and from Grand Central Terminus (GCT), a peak fare on Metronorth, according to the MTA website, is defined in the following way, *"weekday trains arriving GCT between 5 AM and 10 AM and departing GCT between 4 PM and 8 PM."* In the case of the morning rush into GCT the fare table, while being indexed on the departure time t^{DEP} needs to reflect

peak behavior for all the trains that arrive at GCT in the peak region. A journey that leaves before 5 AM but arrives at GCT 7 AM will be assigned a peak fare.

Src Zone	Dst Zone	T_START	T_END	Fare
1	2	0	420	$11.50
1	2	420	720	$13.50
1	2	720	1440	$12.50
1	3	0	420	$11.50

Figure 4.1: An example of a few fare table entries. Notice there may be more than one fare for every pair of fare zones vertices, this typically occurs when there are peak and off peak fare structures.

The construction for the graph so far creates a set of disjoint graphs, each representing a single fare region. We connect the intersections together using transfer edges at intersections.

The final stage in constructing the transit network is to add the transfer edge information. Intersections are identified manually and the transfer edges are added to the graph connecting the individual graphs. Transfer edges may also be added between stops in the same fare region. Every member of an intersection is connected directly, by a transfer edge, to every other member.

Many transit system have certain stations at which they limit passengers to either boarding a train only, but not getting off, or disembarking only but will accept no passengers. In these cases it is necessary to examining the

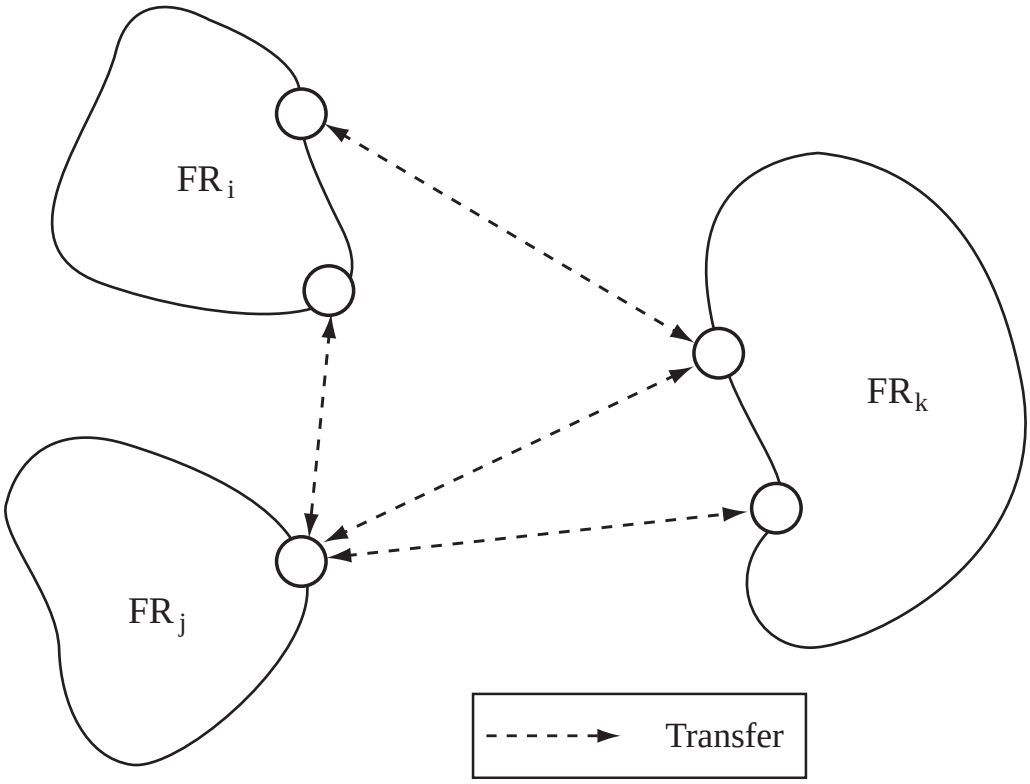


Figure 4.2: The dynamic graph we construct is a collection of individual graphs, each representing a different fare region, connected by transfer edges

transit graph and modify the edge set to reflect the constraints placed upon stops by the providers. The figures 4.4(a)-4.4(c) below show the rewiring operations that need to be performed.

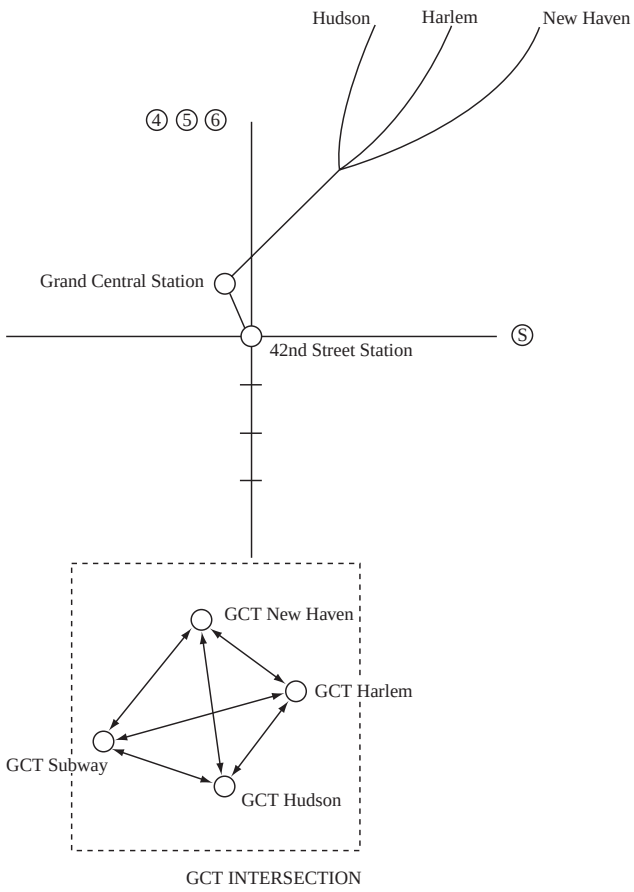
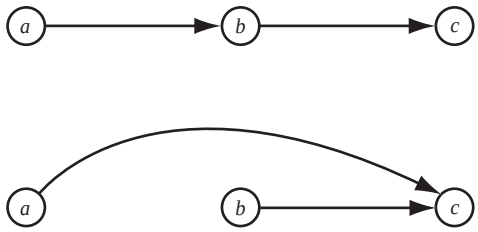
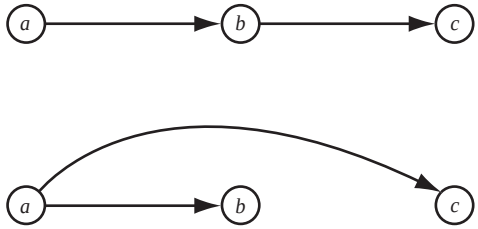


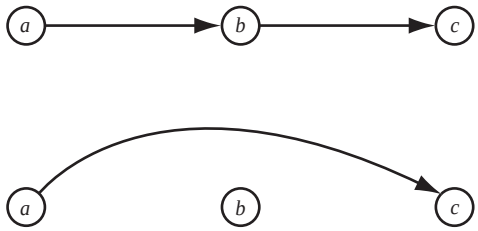
Figure 4.3: An example of how physical stops get represented as intersections. In this example there are four vertices representing Grand Central Station, one for each fare region.



(a) Limiting embarkation



(b) Limiting disembarkation



(c) Skipping stops

Figure 4.4: Graph reductions for stations where passenger may or may not be allowed to embark or disembark.

Figure 4.4(a) shows the case where a passenger may only embark at b but not disembark. The reduction performed is:

EmbarkationOnly(v)

1 **for** $e \in BS(v)$
2 **do for** $e' \in FS(v)$
3 **do** $duration \leftarrow duration(e) + duration(e')$
4 $E \leftarrow E \cup \{(src(e), dst(e'), departure(e), duration, transport(e))\}$
5 $E \leftarrow E \setminus BS(v)$

Figure 4.4(b) shows the case where a passenger may only disembark at b. The reduction here is:

DisembarkationOnly(v)

1 **for** $e \in BS(v)$
2 **do for** $e' \in FS(v)$
3 **do** $duration \leftarrow duration(e) + duration(e')$
4 $E \leftarrow E \cup \{(src(e), dst(e'), departure(e), duration, transport(e))\}$
5 $E \leftarrow E \setminus FS(v)$

Figure 4.4(c) illustrates the case when a stop needs to be skipped.

SkipStop(v)

1 **for** $e \in BS(v)$
2 **do for** $e' \in FS(v)$
3 **do** $duration \leftarrow duration(e) + duration(e')$

$E \leftarrow E \cup \{(src(e), dst(e'), departure(e), duration, transport(e))\}$

$E \leftarrow E \setminus FS(v)$

$E \leftarrow E \setminus BS(v)$

All these reductions can be made time dependent, i.e. a stop may need to be skipped only on the weekends or passengers may not be allowed to get on a train at a station during rush hours.

4.2 Stop selection and prioritization

The itinerary problem requires a set of source and destination stops as its input. Choosing stops properly impacts the quality of the results yielded by an algorithm. A common scenario for a user of an IPA system is a request for directions using street address at the source and destination. Using geocoding software the source and destination addresses are converted into their longitude and latitude co-ordinates. With this geocode information a set of stops is chosen near the start and destination address. There are subtle issues in choosing a stop set that become clear when itineraries are generated.

To generate reasonable itineraries, the starting set often needs to contain more than one vertex to reflect the fact that often there is more than one stop near a starting address. In urban areas, there may be many stops in a given radius while in rural areas there may be a few. Given a starting address it is easy to find the set nearest stops. This can be achieved in a

number of ways, although the fastest and most efficient way is to find nearest neighbors in a Cartesian plane is to use one of the many R-Tree indexing schemes in the literature. [16, 31]. Stop locations are stored in an R-Tree and the tree structure is queried to produce a set of nearby stops. R-Trees can be queried in a number of ways, and in this context the more obvious choices are to find the k-closest stops, a question that is more applicable in rural areas, or to find all the stops in a given radius, a strategy that yields better results in urban areas where the stop density is relatively high.

The other issue with stop selection comes from the presence of intersections. Intersections are typically stops that are so close to one another that starting at one stop in given intersection set makes as much sense to a user as starting from another stop in the same set. If the stop a from fare region FR_a is chosen as the start stop and is a member of an intersection then it is possible that the first stage of an itinerary generated is to transfer to another stop in that intersection, say b from fare region FR_b. To avoid this problem, stops are selected according to the following rule, if a starting stop is a member of an intersection then all the other members of the intersection should be included in the starting set as well.

Another issue involved in choosing stops is caused by solutions from one stop in the source set dominating all other starting conditions. To see how this issue arises consider the following, two stops are chosen as the two closest points to a starting address. One stop, stop A, is much closer to the starting address than the second stop, stop B. If the two stops are on

the path of a single train but B happens to be closer to the destination on the path that the train traverses then, in the set of itineraries generated, all solutions involving stop A will, in most circumstances, be dominated.

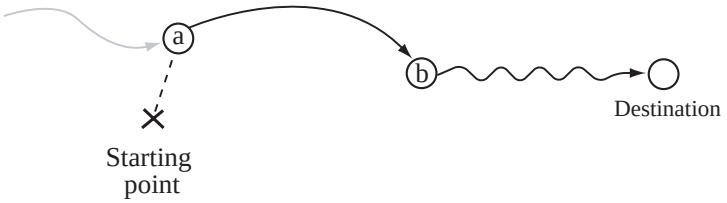


Figure 4.5: Stop selection example

To a user however, getting on at station A is often the most logical option. The strategy taken, to avoid domination of nearby stops, is to associate a priority level with every stop in the starting set and use the priority level in testing for domination when generating the Pareto-paths. This way no Pareto-path that starts with a higher stop priority will ever be dominated by one with a lower priority. We introduce a fourth cost function to determine the paths priority in generating solutions.

Definition 4.2.1 *The priority of a path p,* $f^{\text{PRIORITY}}(p)$*, is given by the priority of the source stop on the path p.*

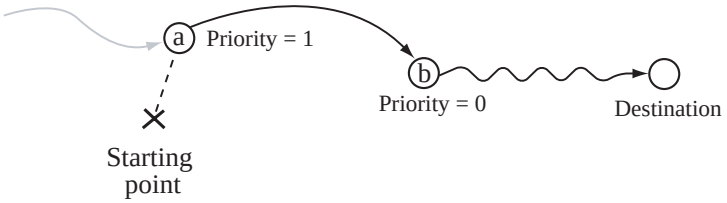


Figure 4.6: Stop selection example

4.3 A naïve algorithm to compute itineraries

This section outlines a naïve algorithm that will compute itineraries based on the cost functions f^{T}, f^{C}, and f^{PRIORITY} in order to demonstrate how the basic algorithm will solve the itinerary problem and that the dynamic graph structure presented here is sufficient to solve the problem directly without relying on a traversal, explicit or implicit, of a space-time network. For reasons that will be explained shortly we do not include f^{XFR} in this version of the algorithm.

Computing the Pareto-optimal paths can no longer be thought of as expanding a shortest path spanning tree as no notion of "shortest" exists. The labeling strategy is still useful however, as in the generic GenericSPT, but each label is now P_v the set of Pareto-optimal solutions at each node. The naïve ParetoPath algorithm is shown in figure 4.3.

The call to Q.InitializeSPT places the elements of the starting set S into Q. This effectively turns the single source problem into a multiple source problem by simulating the first set of iteration from a virtual source vertex connected by arcs, with no cost, on any of the objective functions, and departing at t^{START} to every source in the source set.

As before the algorithm iterates through the queue scanning each vertex as it is selected. Unlike the GenericSPT label correcting algorithm, updating a label set does not involve simply replacing a label if it violates the Bellman conditions, instead, a new labels are added to the set of solutions and then dominated labels are removed. As it is presented on lines

```
ParetoPath(G, S, D, t^START)
 1  Q.InitializeSPT(S)
 2  while Q.NotEmpty
 3  do i ← Q.Select()
 4      for j ∈ Adj[i]
 5      do P_j^* ← Extend(P_i, j)
 6          P_j ← RemoveDominated(P_j, P_j^*)
 7          if ∃p ∈ P_j s.t. p is not marked scanned
 8             then Q.Insert(j)
 9      MarkScanned(P_i)
10      P_D ← RemoveDominated(∪_{k∈D} P_k)
11  return P_D
```

5-6 in figure 4.3 the domination check RemoveDominated can be delayed and performed at once on all the new solutions introduced by extending the paths. RemoveDominated simply performs an $O(k^2)$ computation, where k is the number of Pareto paths at a vertex, comparing each element in the two sets against each other. Domination is checked using a domination predicate $dom(p, p')$, in our case we say a path p is said to dominate p', if the domination condition, given below in equation 4.1, holds.

$$\begin{aligned} dom(p,p') \quad &\Leftrightarrow \quad (f^{\mathrm{C}}(p) < f^{\mathrm{C}}(p')) \\ &\wedge \quad (f^{\mathrm{T}}(p) < f^{\mathrm{T}}(p')) \\ &\wedge \quad (f^{\mathrm{PRIORITY}}(p) > f^{\mathrm{PRIORITY}}(p')) \end{aligned} \tag{4.1}$$

Every path p in a solution set is marked as *scanned* or *unscanned*. Once a node i has been scanned all the solutions for the label set of the node, P_i are marked as scanned. Even though i may be reinserted into the Q there is no need to try to extend these solutions. The procedure EXTEND, shown in figure 4.3 extends the paths from a node i, to a neighboring node j extending only those paths in i that are not marked as scanned. If after the REMOVEDOMINATED call the set P_j^* has added new undominated solutions to P_j then j is reinserted in the queue and will eventually be scanned again. As in the Bellman-Ford-Moore algorithm, the algorithm terminates when there are no more vertices left in Q to be scanned. Since we are looking for the Pareto optimal solution set from S to D we need to return the undominated set of solution that reaches D. Line 10 in figure 4.3 computes this set.

EXTEND(P_i, j)

1 $extended \leftarrow \{\}$**for** each $p \in P_i$ s.t. p is not marked

2 **do for** each $(i, j, t^{\text{DEP}}, t^{\text{DUR}}, transport) \in E$

3 **do** $extended \leftarrow extended \cup$ CONCATENATE$(p, (i, j, t^{\text{DEP}}, t^{\text{DUR}}, transport))$

4 **return** *extended*

The proof of correctness is by a simple induction on the length of the paths formed.

The PARETOPATH algorithm has a complexity $2^{O(n)}$, where n the number of vertices in the dynamic graph G. This is easily shown by considering the

following example. In the case where we have two identical trains, v_1 and v_2 running down a track at the same time with the same cost (a situation that never occurs in practice) the number of solutions at the first stop is 2 as neither solution is dominated by the other. At the next stop down the track there are 4 solutions and so on until the n^{th} stop where there are 2^n solutions. Therefore it is possible to have an exponential number of solutions in n. If there are k identical trains the complexity is $O(k^n)$.

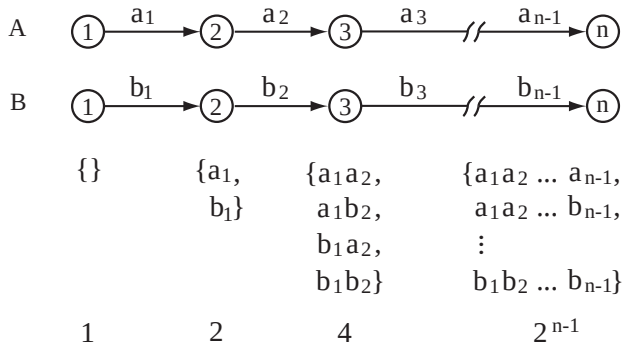


Figure 4.7: Duplicate edges cause worst case behavior

In practice however we never see two trains running down the same track at the same time.

One may be tempted to add f^{XFR} to the domination predicate in order to allow ParetoPath to solve the itinerary problem as shown below.

$$\begin{aligned} dom(p, p') \quad \Leftrightarrow \quad & (f^{\text{C}}(p) < f^{\text{C}}(p')) \\ \wedge \quad & (f^{\text{T}}(p) < f^{\text{T}}(p')) \\ \wedge \quad & (f^{\text{XFR}}(p) < f^{\text{XFR}}(p')) \\ \wedge \quad & (f^{\text{PRIORITY}}(p) > f^{\text{PRIORITY}}(p')) \end{aligned} \tag{4.2}$$

The problem with the domination predicate above can be seen from the example in figure 4.8. In this example, consider two identical train v_1 and v_2 running down the same track at different times. Train v_1 departs from stops a, b, c, and d at times t_1, t_2, t_5, t_7, respectively while train v_2, an express train, departs a, c and e at times t_1, t_4 and t_6. For this example let the cost in terms of f^{C} of getting to the stops be identical and let $t_1 < t_2 \ldots < t_7$.

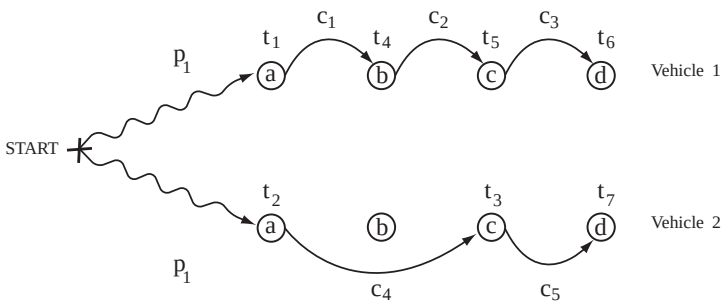


Figure 4.8: The most direct path to stop d will never be found. The *dom* predicate will remove the vehicle 1 solution to c.

The label that corresponds to v_1 reaching c will be dominated by the label for v_2 reaching c. When it comes time to scan the vertex c to expand paths further the path corresponding to train v_1 will no longer be considered for expansion. In order to get to d the path that will be expanded is the path using v_2 and then a transfer to v_1 at c, and while there is nothing wrong with that solution, the solution that represents staying on v_1 to reach d, incurring no transfer costs, is eliminated from the Pareto set thereby preventing the algorithm from discovering the complete set of solutions.

In the next chapter we will show how to reintroduce edges f^{XFR} to the

domination predicate correctly.

4.4 Special cases of the ParetoPath algorithm

If we reduce the set of objective functions to just costs f^{T} and f^{C} the queue may be implemented using any of the schemes discussed in chapter 3. Pallottino et al in [27] suggest that for problems in the transportation domain where networks have non-negative arc costs, are quasi-planar, sparse, and structured the simplest version of the Bellman-Ford-Moore algorithm implementing a FIFO queue often does the best due to the constant time cost queuing and dequeuing vertices. This improved behavior is due to the overhead associated with insert and selection operations in a priority queue. In this case however since we are optimizing on the objective functions f^{T} and f^{C} a shortest search approach can be taken to minimize the amount of scanning if we take note of the fact that as we add edges to a path the cost of the path, with respect to f^{T}, can only increase. Naturally, to be a shortest first algorithm Q must be a priority queue.

If we assume that no optimal journey through the transit graphs takes more than some constant maximal amount of time, T^{MAX} then Q may be implemented as an array of buckets, one bucket for each time interval. Time becomes the measure on which priorities in Q are established and the array implementation avoids the overhead typically associated with inserting and removing elements from a queue. Since every departure and duration on the transit graph takes less than the time-horizon, it is safe to assume that

there will be no itinerary that takes more than twice that amount of time to arrive. In our case the time-horizon is 10080, therefore an array with 20160 buckets is assumed to be adequate for generating itineraries. The priority at which a vertex i is inserted is simply the weight of the earliest arrival times of the paths in P_i.

Implementing a shortest first variant now becomes a matter of maintaining a counter to index the queue. This counter represents the minimum weight of elements in the queue. Each Q.SELECT operation now becomes a matter of incrementing the counter until the first non-empty bucket is reached and then dequeuing the first element (vertex) in that bucket from the queue. As the minimum weight will only increase monotonically, since paths can only grow with respect to f^{T}, once the counter has been incremented no element will ever be enqueued in a bucket with a lower weight. Sweeping through the array ensures an amortized constant time cost for the queue operations.

In another variant of the above algorithms to solve the *minimum arrival time* problem. If we consider the array based queue implementation above but are only interested in the f^{T} as the sole objective function, it is easy to see that PARETOPATH reduces to the simple Dijkstra's algorithm, where the minimum cost element of a queue never decreases, and the running time of this variant is $O(m + n)$.

4.5 Transfer minimization

The last section discussed a new implementation of the ParetoPath algorithm that performs a domination check on three of the four desired domination criteria. This section discusses how to reintroduce the directness criteria, f^{XFR}.

Adding f^{XFR} to do the domination predicate, i.e.

$$\begin{aligned} dom(p, p') \quad \Leftrightarrow \quad & (f^{\text{C}}(p) < f^{\text{C}}(p')) \\ \wedge \quad & (f^{\text{T}}(p) < f^{\text{T}}(p')) \\ \wedge \quad & (f^{\text{XFR}}(p) < f^{\text{XFR}}(p')) \\ \wedge \quad & (f^{\text{PRIORITY}}(p) > f^{\text{PRIORITY}}(p')) \end{aligned} \tag{4.3}$$

implies that we will prune paths that while dominated are still needed to extend the path correctly. Recall the example illustrated in figure 4.8. One approach to fixing this problem would be to mark dominated paths and continue to expand all paths, dominated and undominated at each node. This is an exhaustive search and is impractical, the number of solutions, dominated and non-dominated would grow exponentially with the number of vertices on the path. Instead a new class of edges called direct edges, is used to reintroduce direct paths where they exist.

Definition 4.5.1 *A vertex v is* directly reachable *from u if there is a path p formed by concatenating edges in E that have the same transport.*

Definition 4.5.2 *A* direct path *p is a path that connects two directly reachable vertices to each other.*

Definition 4.5.3 *A* direct edge $e_D(p)$ *is an edge between two directly reachable vertices* u *and* v *constructed to represent the departure time* t^{DEP}*, duration* t^{DUR} *and transport transport of a direct path* p*.*

Let p *is a direct path between two vertices* u *and* v *then the direct edge* e_D *between the vertices is given by* e_D *where*

$$e_D(p) = (u, v, departure(p), duration(p), transport(p))$$

4.5.1 Creating the set of direct edges

The set of direct edges can be precomputed for each vertex in the graph. A naïve way to perform this computation is shown in figure 4.5.1. Each direct edge is simply the transitive closure of the edges in the train starting from every stop that a train visits.

4.5.2 Using direct edges to minimize transfers

Consider the example from figure 4.8 again. By augmenting the graph with direct edges as shown in 4.5.2 we can now see how when calling Extend searching for the direct edge d_1 allows the construction of a more direct Pareto path from a to d.

When extending a path to u by an edge (u, v) the ParetoPath needs to account for the fact that a path that may lead to a more direct solution was removed by the domination predicate 4.3. Paths that are more direct are introduced to the set of paths to be extended from P_u by searching for direct edges, however these paths are not members of the Pareto set. To

```
MakeDirectEdgeSets(G)
 1  E = GetEdges(G)
 2  Transports ← MakeTransportSequences(E)
 3  for each transport_seq ∈ TransportSequences
 4  do last_edge_in_path ← GetLastEdge(t)
 5      l ← length(transport_seq)
 6      for i = 1 to l
 7      do first_edge_in_subpath ← transport_seq[i]
 8          src = source(first_edge_in_subpath)
 9          dep = departure(first_edge_in_subpath)
10          for j = i + 1 to last_edge_in_path
11          do dst = destination(transport_seq[j])
12              dur = arrival(transport_seq[j]) − dep
13              transport = transport(transport)
14              WriteDirectEdgeToDB(src, dst, dep, dur, transport)
15
```

Figure 4.9: The MakeDirectEdgeSets algorithm iterates through each transport sequence creating a new direct edge for every stop it encounters. Note: Line 10 requires that there are no direct edges representing paths that consist of a single edge.

find more direct paths for a path p a search is performed by following the predecessor edges on a path looking for vertices from which a direct edge

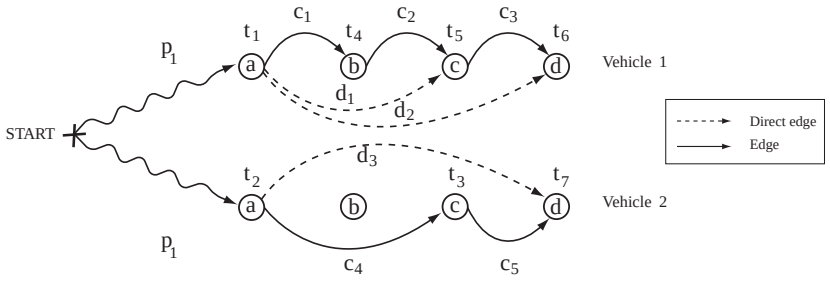


Figure 4.10: Augmenting the graph with direct edges allows a direct path to be found to d. The direct edge reintroduces the vehicle 1 solution at c.

can be used. The search need not continue past a predecessor edge that is a transfer between fare-regions, this is because direct edges are intra-regional in nature. In addition, the search for a more direct path need only be carried out when a transfer from one vehicle to another in the same fare region occurs as this is the only time at which the directness measure f^{XFR} can improve by the introduction of a direct edge.

To take advantage of the direct edges the EXTEND procedure must be modified to introduce these 'more direct', but otherwise dominated, paths to the set of paths being extended for a vertex. The new version of EXTEND is shown in figure 4.5.2. The PARETOPATH algorithm remains unchanged.

The procedures DOESAMOREDIRECTPATHEXIST and CREATEMOREDIRECTPATHS can be implemented fairly inexpensively as follows. As stated earlier the predicate DOESAMOREDIRECTPATHEXIST need only scan backward along a path $p \in P_u$ as far as the first global transfer edge. In addition, by maintaining a pointer at each path to the first transfer along the route, the

```
EXTEND(P_i, j)
1  extended ← {}
2  P ← P_i
3  for each p ∈ P s.t. p is not marked
4  do if MOREDIRECTPATHEXISTS(p)
5        then D ← CREATEMOREDIRECTPATHS(p)
6             P ← P ∪ D
7             for each (i, j, t^DEP, t^DUR, transport) ∈ E
8     do extended ← extended ∪ CONCATENATE(p, (i, j, t^DEP, t^DUR, transport))
9  return extended
```

Figure 4.11: EXTEND for similar edge sets.

search for more direct connections can avoid testing to see if the u is directly reachable from the current vertex. A predicate DIRECTLYREACHABLE is easily implemented by maintaining an adjacency list derived from the direct edges. If DIRECTLYREACHABLE indicates that u is directly reachable from some vertex w then CREATEAMOREDIRECTPATH simply performs a concatenation of the paths in P_w with the first directly reachable edge (w, u).

Chapter 5

A Faster way to compute Pareto paths

The ParetoPath algorithm presented in the previous chapter demonstrates that it is possible to compute Pareto-optimal paths on dynamic networks derived from transportation models. The problem with the algorithm presented however is its long running time in practice. In engineering a system for use as a web service the searches need to be executed rapidly. In addition, the ParetoPath algorithm fails to incorporate the transfer cost objective function f^{XFR}, as one of the criterion on which the Pareto-optimality of a path is evaluated. This chapter introduces new algorithms, variants of the basic ParetoPaths algorithm, to solve the itinerary problem for large scale networks. The algorithms presented in this section describe how to eliminate the redundant searches through the graph, how to incorporate the directness measure, f^{XFR} in the set of objective functions

and how to utilize precomputation on the graph to improve the response time of the algorithm. Finally this section touches upon some of the more important implementation details necessary to implement a fast and scalable system.

Issues of scalability play an important part in the design of an improved ParetoPath algorithm. As the dynamic networks become larger, it becomes harder to store the entire graph in main memory making it necessary to store the the graph structure on disk. As always, when working with large data structures on disk, large performances can be made by minimizing disk access. A typical disk access on modern commodity hardware today costs between 10-12 ms, in comparison, main memory access takes between 2 to 200 clock cycles depending on whether the information being sought is in a cache (L1 or L2), or in main memory. In the discussion below, minimizing disk access quickly becomes an important part of the focus, even if it means performing a comparatively large amount of work in main memory. To accomplish the goal of minimizing disk access, various levels of static analysis on the graph are used to accomplish the performance goals. This approach is common in speeding up shortest path algorithms and good examples of these techniques in practical transportation and direction finding systems can be found in [20, 14, 35].

5.1 Similar Edge Sets

In the basic ParetoPath algorithm most of the computation is spent in scanning a vertex to extend the solution set. This is fairly typical for Bellman-Ford-Moore type algorithms and is the reason why reducing the number of scanning operations is the central focus in improving the performance of SPT_L algorithms in the literature [3, 6, 7, 13, 15, 9, 21, 28]. While the ParetoPath algorithm can certainly benefit from the above strategy the approach taken here is to perform an analysis of the graph ahead of time and use this information to reduce the complexity of the scanning operation. This is because the cost of scanning, in the ParetoPath algorithm is even more severe as each adjacent pair of vertices may have hundreds of edges between them for the intervals that typically represent the periods of time-tables.

When at a vertex v the ParetoPath algorithm tries to extend the path by adding every edge $e \in FS(v)$ and then removing all the dominated solutions from the label at that node. In transportation networks due to the physical and economic constraints, vehicles often traverse the same route many times in a given time period. Recognizing this similarity leads to a variant of the core algorithm that greatly reduces the amount of work done in solving the itinerary problem. This section defines a similarity relation on edges and then explores how to use the relation to implement an improved version of the ParetoPath algorithm.

5.1.1 An equivalence relation for similarity

Consider the following example. Two vehicles v_1 and v_2 travel down a path making the same stops along the way. The vehicles behave identically with respect to the time taken to traverse the path and the cost of traveling along the path. The only way in which they differ is that v_2 leaves at some time after v_1. In expanding a path p from a vertex to an adjacent vertex there may be many possible edges (vehicles traveling between two stops) to choose from, however, if two of these edges have the same impact on the cost of the path but one leaves later and takes the same amount of time to travel between vertices, the path extended by the edge that leaves earlier will be dominant. Recognizing similar edges proves a way to improve the basic ParetoPath algorithm by preventing the call to Expand from creating paths that we know ahead of time will be dominated.

Recall that each non-transfer edge $e \in E$ is given by an tuple of form $(u, v, t^{\text{DEP}}, t^{\text{DUR}}, transport)$.

Definition 5.1.1 *A* transport set *is the set of all edges in E with the same transport label.*

Definition 5.1.2 *A* transport path *is a path formed by concatenating all the edges in a transport set.*

Definition 5.1.3 *If $P = \langle e_1, e_2, \ldots, e_l \rangle$ is a transport path then* Subpath(P)

is the set of all in P, i.e.

$$\text{Subpath}(P) = \bigcup_{i=1}^{l} \bigcup_{j=1}^{l} \langle e_i, \ldots, e_j \rangle$$

Definition 5.1.4 *Two transport paths P_1 and P_2 are* similar *iff*

1. *The two paths are* time equivalent *i.e.*

$$\begin{aligned} &\forall p_1 \in \text{Subpath}(P_1) \\ &\quad \exists p_2 \in \text{Subpath}(P_2) \text{ s.t.} \\ &\quad f^{\text{T}}(p_1) - departure(p_1) = f^{\text{T}}(p_2) - departure(p_2) \end{aligned}$$

2. *The two paths are* cost equivalent *i.e.*

$$\begin{aligned} &\forall p_1 \in \text{Subpath}(P_1) \\ &\quad \exists p_2 \in \text{Subpath}(P_2) s.t. \\ &\quad f^{\text{C}}(p_1) = f^{\text{C}}(p_2) \end{aligned}$$

If we write $P_1 \; S_P \; P_2$ to mean the transport paths P_1 and P_2 are similar then it is easy see that S is reflexive, symmetric, and transitive and is therefore an equivalence relation.

Definition 5.1.5 *Two edges e_1 and e_2 are* edge similar, *$e_1 S_e e_2$, if they are members of similar transport paths and travel between the same source and destination stops. More formally, if p_1 and p_2 are transport paths,*

$$\begin{aligned} e_1 \; S_e \; e_2 \quad \Leftrightarrow \quad & p_1 \; S_P \; p_2 \\ \wedge \quad & e_1 \; S_e \; e_2 \\ \wedge \quad & source(e_1) = source(e_2) \\ \wedge \quad & destination(e_1) = destination(e_2) \end{aligned}$$

It is easy to see that S_e is an equivalence relation. We can use S_e to partition the set of edges E into disjoint sets of similar edges.

Let $dom(p, p')$ mean that a path p_1 dominates p_2 according to the domination criteria

$$\begin{aligned} dom(p, p') \quad &\Leftrightarrow \quad (f^{\mathrm{C}}(p) < f^{\mathrm{C}}(p')) \\ &\wedge \quad (f^{\mathrm{T}}(p) < f^{\mathrm{T}}(p')) \end{aligned}$$

Lemma 5.1.1 *If two similar edges extend a path then either one of the extended paths must be dominated by the other or the two edges have the same time of departure. More formally, let p be a path through G and let $\circ$ be the path concatenation operator,*

$$\begin{aligned} e \; S_e \; e' \quad &\Rightarrow \quad dom(p \circ e, p \circ e') \vee dom(p \circ e', p \circ e) \\ &\vee \quad departure(e) = departure(e') \end{aligned}$$

Proof: If $e \; S_e \; e'$

- Case1: When e departs before e'. $p \circ e$ must dominate $p \circ e'$ since it arrives earlier, hence has a lower f^{T} but the objective function f^{C} remains unchanged.
- Case 2: When e' departs before e, $dom(p \circ e, p \circ e')$ by symmetry with Case 1.
- Case 3: If neither case 1 or 2 holds then clearly $e.t^{\mathrm{DEP}} = e'.t^{\mathrm{DEP}}$.

■

Lemma 5.1.1 implies that given as set of similar edges we need only pick one edge from the set to extend a Pareto path. Picking more than one

edge from a given set will automatically result in one of the two edges being dominated. By adopting a strategy that picks only one edge from each similar edge set we reduce the amount of work done in a scanning operation significantly.

5.1.2 Building similar edge sets

Partitioning of the edges in the graph E into equivalent edges is done by the MakeSimilarEdgeSets algorithm shown in 5.1.2. The algorithm is straight forward, the edges in the graph are assembled into transport paths, the transport paths are then partitioned based on the path similarity relation and are finally similar edge sets are written to disk.

MakeSimilarEdgeSets(G)

1 $E =$ GetEdges(G)

2 $Transports \leftarrow$ MakeTransportPaths(E)

3 SortTransportSequences(TransportCompare, $Transports$)

4 $SimilarTransports \leftarrow$ Partition($Transports, S_P$)

5 WriteSimilarEdgeSetsToDisk($SimilarTransports$)

Figure 5.1: The MakeSimilarEdgeSets algorithm

MakeTransportSequences creates transport sequences out of the edge set by appending the edges for a single transport to get the path which uses the most edges. This corresponds to the route that the vehicle takes. The transport sequences are then sorted using the comparison operation in

5.1.2, placing transports with identical stop sequences in a contiguous region in an ordered collection of transport sequences.

```
TRANSPORTCOMPARE(seq_1, seq_2)
 1  if length(seq_1) < length(seq_2)
 2     then return TRUE
 3  if length(seq_1) > length(seq_2)
 4     then return FALSE
 5  length ← length(seq_q)
 6  for i ← 0 to length
 7  do if src(seq_1[i]) < src(seq_2[i])
 8        then return TRUE
 9     if src(seq_1[i]) > src(seq_2[i])
10        then return FALSE
11     if dst(seq_1[i]) < dst(seq_2[i])
12        then return TRUE
13        else  return FALSE
```

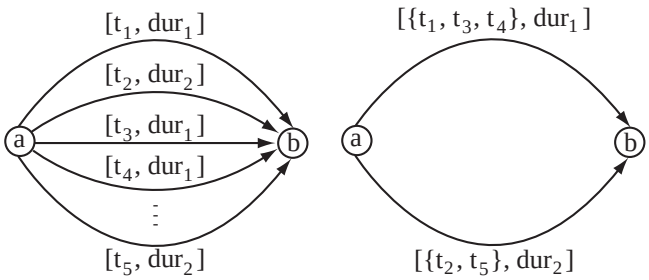
Figure 5.2: A comparison operation to sort transport sequences

PARTITION creates the equivalence classes of the transport paths by scanning through the paths in the set and grouping them based on similarity, as defined by S_P. The most obvious implementation of this is $O(n^2)$ in the number of transport sequences, i.e. simply scanning though the collection of transport sequences and slotting each transport into the

appropriate group based on the path similarity relation S_P
Finally, WriteSimilarEdgeSetsToDisk splits a transport sequence into its component edges and for every set of similar edges, as defined by S_e stores the edges to disk. For efficiency a B-Tree data structure from the BerekleyDB system is used to create a fast disk efficient search tree, although in theory any indexing scheme could be used. If the similar edge sets were to be placed in main memory only, other data structures, such as a hash table, might be more appropriate. After MakeSimilarEdgeSets has run each adjacency in the transit graph forms an index in the B-Tree, and at each index there is a set of similar edge sets. The similar edge sets need not be in any particular order but the edges represented within the sets are sorted in order of their departure time to allow binary searches to be performed when seeking a particular edges by departure time.
The partitioning is performed as a precomputation on G. The reduced multi-graph G_R, is used to implement a variant of the ParetoPath algorithm that exploits edge similarity.

5.1.3 Using similarity in the ParetoPath algorithm

As stated earlier Lemma 5.1.1 implies that only one edge need be considered from each similar edge set when extending paths along a given adjacency. Given a path $p \in P_i$ the edge selected from each set is the first edge that leaves after $arrives(p)$. Since edges are ordered by departure time this edge can be found efficiently using a binary search. The



(a) The unreduced graph.

(b) The reduced graph.

Figure 5.3: Reduced multi-graph for similar sets.

algorithm EXTEND is shown in 5.1.3.

EXTEND(P_i, j)

1 $extended \leftarrow \{\}$

2 $S \leftarrow$ GETSIMILAREDGESETS(i, j)

3 **for** each $p \in P_i$ s.t. p is not marked

4 **do for** each $s \in S$

5 **do** $t^{\text{ARR}} \leftarrow arrives(p)$

6 $e \leftarrow$ FINDFIRSTEDGETHATLEAVESAFTER(s, t^{ARR})

7 $extended \leftarrow extended \cup$ CONCATENATE(p, e)

8 **return** $extended$

Figure 5.4: EXTEND for similar edge sets.

EXTEND must select one edge from each similar edge set. The effect using

similar edge sets for routing can easily been seen as a reduction of the transit graph. Instead of viewing the transit network as a dynamic multi-graph with one edge for each vehicle that travels between two vertices, the graph can be thought of as being a dynamic multi-graph where each edge has a labeling function that alters only the time of departure along the edge.

Lemma 5.1.2 PARETOPATH *using similar edge sets is correct.*

Proof: From lemma 5.1.1 the algorithm EXTEND (Fig. 5.1.3) will create only one path from each similar edge set. Any other path created by selecting more than one edge from a single similar edge set will ensure that one of the two paths is dominated. ■

The effect of using similar edge sets on the performance of PARETOPATH is dramatic, especially in mass commuter systems where trains run frequently. In the north-east corridor transit graph for 77 different transit providers, the total number of edges in the transit graph is approximately 2.7 million edges. In the reduced graph there are still 2.7 million edges but these edges are represented by only 5,892 similar edge sets. Apart from the savings in the cost of creating and checking dominance the reduced edge set representation also reduces the cost of disk activity.

5.2 The All Times Single Source Pareto Path Problem

Calculating the Pareto-path in the itinerary setting is time dependent and the ParetoPath needs to take the time of embarkation of a journey, the t^{START} parameter, into account. For the precomputations this implies that solutions need to be generated for every time of day that a vehicle leaves a destination. The naïve approach, to generate an itinerary for every time of day that a vehicle leaves a source vertex would certainly work but there are deteriorating performance implications to this approach.

The following sections concentrate on speeding up the ParetoPath algorithm by performing more static analysis of the transit graph. The analysis requires an examination of sets of paths from one stop to another starting at all times of day. While it is possible to perform this analysis by repeated calls to the ParetoPath algorithm this section presents a more elegant solution, one that computes the Pareto paths from a given source at all departure times. In addition to being more efficient, running the AllTimesParetoPath effectively avoids some of the problems that arise when merging the results from the individual runs. For a given source and destination two different times of departure lead to the same time of arrival. This can, and frequently does happen because waiting is allowed at vertices. Consider the example in figure 5.2. Vehicle A leaves a vertex s at a time t_1 while B leaves s some time later at t_2. At some stage during the

path a switch is made to C and both paths arrive at a vertex d at time t_3. It is possible that both paths are in the Pareto path set for P_d. Clearly one of these paths is redundant. If there is no advantage to leaving earlier then the the path that starts at t_1 need not be in the solution set. When merging the solution sets for runs performed at different times the redundant solutions would need to be eliminated.

An alternative is to use a simple graph reduction, one that take the time of departure into account and solves the problem on this new graph with a modified version of the ParetoPath algorithm. If s is the source transfer vertex, then for every non-transfer edge e that departs from s at a time t create a new virtual vertex v_t with a starting time of 0, a duration of time $departure(e)$ and a transport field with the same value as e. In addition, a new start vertex is connected to each of the new virtual vertices by a transfer edge.

Pseudo-code to perform this reduction is shown in figure 5.2.

The reduction performed by AllTimesReduction effectively transforms the Pareto optimal path problem into a single source problem. Let G' is the new graph returned by running AllTimesReduction on G. Running ParetoPath as it is currently on G' with a starting time of 0 would not generate the Pareto paths for all departure times. Paths generated from earlier departure times will, in most cases, dominate paths from later departure times, in addition the domination check at the end of the ParetoPath algorithm would remove solutions from the Pareto set

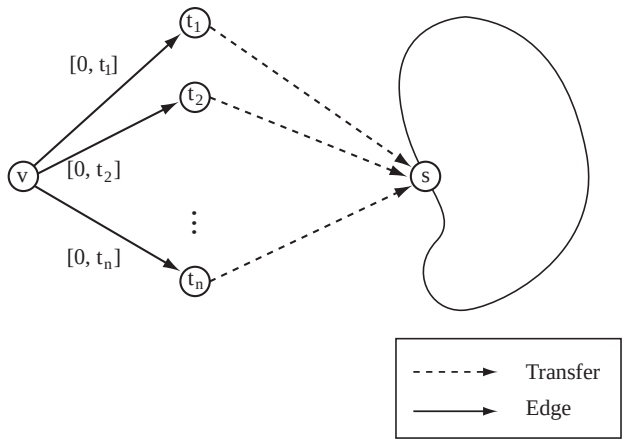


Figure 5.5: The new virtual source is connected to a set of new vertices, each representing an edge departing the source vertex s.

AllTimesReduction$(G = (V, E), s)$

1 $V' \leftarrow V \cup v_{start}$

2 **for** each $e \in E$ *s.t.* $src(e) = s \wedge not\_a\_transfer(e)$

3 **do** $t \leftarrow departure(e)$

4 $\quad V' \leftarrow V' \cup v_t$

5 $\quad E \leftarrow E \cup \{(s, v_t, 0, t, transport(e)), (v_{start}, v_t, \square, 0, 0)\}$

6 **return** (V', E)

Figure 5.6: The single source problem is turned into an all times single source problem by creating a new source, one for each time of departure

.

incorrectly. The domination predicate that follows the call to Expand should only be applied to two paths if they start by traveling to the same

virtual vertex.

The domination predicate *dom* is now given by equation 5.1.

$$\begin{aligned} dom(p,p') \quad &\Leftrightarrow \quad (f^{\mathrm{C}}(p) < f^{\mathrm{C}}(p')) \\ &\wedge \quad (f^{\mathrm{T}}(p) < f^{\mathrm{T}}(p')) \\ &\wedge \quad (f^{\mathrm{XFR}}(p) < f^{\mathrm{XFR}}(p')) \\ &\wedge \quad (first\_vertex(p) = first\_vertex(p')) \end{aligned} \tag{5.1}$$

In addition, if two paths share the same predecessor edge and are identical in all their cost functions then the path with the earlier departure time is redundant and should not be inserted into the Pareto set.

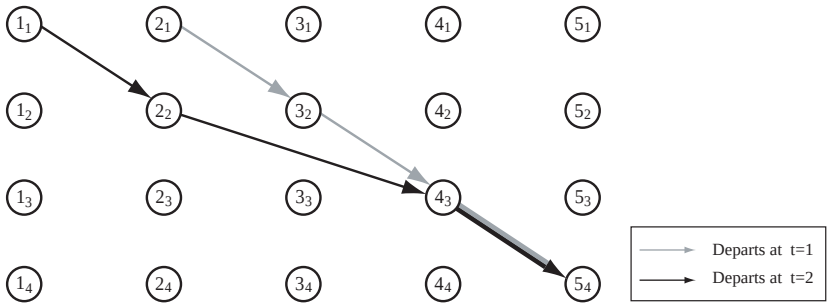


Figure 5.7: The space-time network above illustrates when two solutions are 'comparable' under domination and when we can prune the redundant solutions

The algorithm AllTimesParetoPath is shown in figure 5.2.

This version of the AllTimesParetoPath algorithm solves the problem for a single source. It is trivial to extend this version for a set of sources.

```
AllTimesParetoPath(G, s)
 1  S ← AllTimesReduction(G, s)
 2  Q.InitializeSPT(S)
 3  while Q.NotEmpty
 4  do i ← Q.Select()
 5      for j ∈ Adj[i]
 6      do P*_j ← Extend(P_i, j)
 7          P*_j ← RemoveDominated(P_j, P*_j)
 8          P_j ← RemoveRedundant(P*_j)
 9          if ∃p ∈ P_j s.t. p is not marked scanned
10              then Q.Insert(j)
11      MarkScanned(P_i)
12  return {P_D \ {P_s ∪ P_S}}
```

Figure 5.8: Line 1 performs the reduction. The set S returned is the starting set of virtual starting vertices. Thus making the problem the same as starting the search from multiple sources as before. This version of the ParetoPath algorithm uses the *dom* predicate described in equation 5.1. RemoveRedunant compares paths that have an identical predecessor edge to see if they are equal on all the cost functions. If they are the path with the earlier departure time is removed. A domination check is not performed between all destinations in the last step as before.

5.3 Generating itineraries for large server loads

In designing a Passenger information systems there are soft real-time requirements that need to be met, responses to queries need to be returned to travelers in relatively short periods of time. Under high loads the core algorithm in the system needs to be extremely fast at computing the Pareto solution set. This section presents a strategy to implement a passenger information system that solves the itinerary problem online. While this strategy works well in coping with very high server loads the execution of it fast becomes impractical. In a later section similar strategy is presented that helps reduce the online response time. Both strategies involve using precomputation of Pareto routes through the dynamic graph. The first strategy involves precomputing and storing solutions to small subsets of vertices and then uses these precomputed solutions to generate itineraries. The second approach focuses on using the precomputed solutions to guide the online search.

5.3.1 The two tier itinerary generation strategy

In theory, given that the entire graph structure is static and all departure times are known in advance, a possible approach to scaling the itinerary problem would be to precompute the solution set for all pairs of vertices at all times of the day. In practice, this approach would require far too much space to store the results of such a computation. The two tier approach instead aims to precompute and therefore store a smaller set of solutions

that can be used to reduce the online response time.

The transit graph naturally decomposes into its individual sub-graphs, one for each fare-region. The two tier approach aims to reduce the online computation to finding the quickest way between the source and destination fare regions. This is in essence a more aggressive version of the component routing method independently discovered by [14]. In the two tier method paths from every transfer point, a vertex connected to a global transfer arc, to every other vertex in the same fare region as the transfer point. The resulting set of paths is compressed into a set path edges, one for each Pareto path generated. This set of edges, combined with the global transfer edges and the transfer points forms a graph that is used to compute the Pareto optimal paths from fare regions to fare regions. These solutions are stored in a database as well. Given that we have already computed the Pareto-paths from every transfer point to every other vertex in its fare region, we can store these solutions in a database as well. To generate a Pareto-path from any vertex to any other vertex online now becomes a simple matter of computing, using the transit graph for the starting fare region, a set of solutions to every transfer point in the fare region and then performing two database look ups, the first to find the Pareto optimal solutions to each of the destination fare regions, and the second to find the Pareto-optimal solutions from the transfer points in the destination fare regions to the final destinations.

It is possible to remove the first stages call to the ParetoPath problem

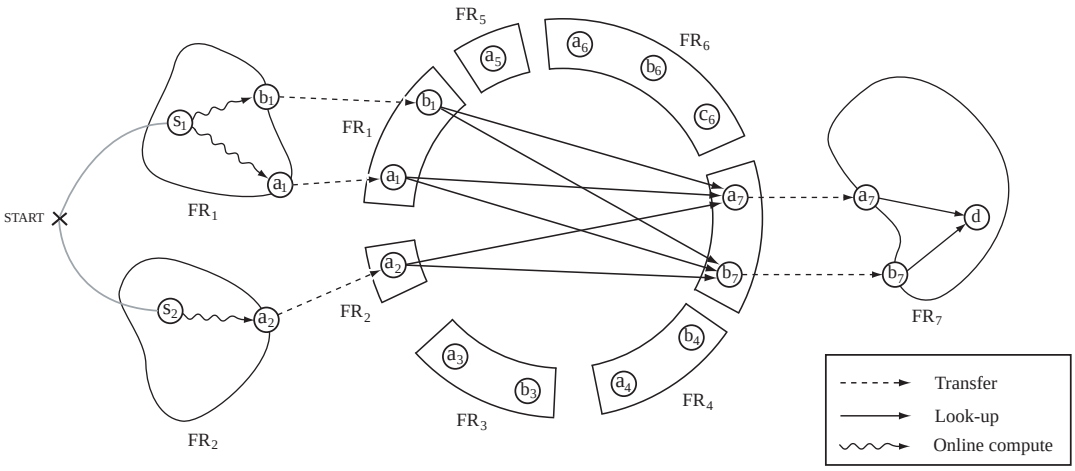


Figure 5.9: Only the first stage of the two tier generation strategy involves a call to ParetoPath. Stages two and three are simple database lookups.

when online by precomputing all the paths from every stop in a fare region to the fare-region's transfer point, but this would be a far more expensive precompute. In fact, the second set of database lookups could well have been a call to the ParetoPath algorithm. The reason it is a database query however is due to the single source nature of the ParetoPath algorithm. Even if the strategy was to only precompute the Pareto paths from a transfer point in a fare region to the other transfer points in the same fare region the algorithm, short of a set of stopping conditions to terminate the algorithm after all paths to the destinations have been found, would have to compute the set of paths to all the other vertices in the fare region. Since this computation is already being performed, the result are stored and used to speed up the final stage, generating a path

from the reduced graph to the destinations.

5.3.2 Generating the second tier graph

Generating the second tier graph is performed by first running the AllTimesParetoPath on each transfer point in every fare region and storing the results of this computation in the database. Every path between two transfer points in this database will now be represented by an edge in the reduced inter-regional transit graph.

Definition 5.3.1 *An* inter-regional edge *is an edge that represents a path from an exit point to an exit point in the transit graph. An edge between two vertices u and v has the following attributes, the time of departure t^{DEP}, the duration of the journey t^{DUR}, the cost f^{C} of the journey and the number of transfers on this journey f^{XFR}.*
More formally, if p is a path between two exit points u and v in a fare region then e the edge that represents p in the inter-regional transit graph is given by

$$e = (u, v, departure(p), duration(p), f^{\text{C}}(p), f^{\text{XFR}}(p))$$

Let P be the set of all Parter paths computed at all times from exit point to exit point in the transit graph $G = (V, E)$. The inter-regional transfer graph is given by $G^* = (V^*, E^*)$ where:
V^* = the set of all transfer points in G.
$E^* = \bigcup_{p \in P}(src(p), dst(p), departure(p), duration(p), f^{\text{C}}(p), f^{\text{XFR}}(p))$

+ All transfer edges in G

Calculating the Pareto optimal paths from the exit points of one fare region to another becomes a matter of calling AllTimesParetoPath on G^* for every vertex $v \in V^*$. Because the edges in G^* carry the costs f^{C} and f^{XFR} for the paths that they represent the concatenation operation in Expand is additive on each of the objective functions and there is no need for a fare lookup table in this computation. The results from this stage of the computation are saved in a database and are known as the tier two paths.

5.3.3 Generating an online response

Generating an online response is now a matter of finding a Pareto path from a source vertex to the transfer points in the same fare region, an online call to the ParetoPath algorithm and then performing a series of lookups, one set to get to the destination fare regions, the next set to get from the transfer points of those fare regions to the actual destinations. Since there may be more than one source and destination running the first stage of computation, the call to ParetoPath algorithm, will need to be carried once for each fare-region. The call to ParetoPath for a fare-region will take only the intra-regional component and the sources in the fare-region as its arguments. The destinations are the transfer points in each fare region. These calls to ParetoPath are independent of each other and are usually executed in parallel to further reduce the response

time. After the paths to the exit points of each fare region have been computed a lookup on the tier two path database will in single step obtain a path from the transfer points of the source fare region. The rest of the computation is performed by concatenating the look-ups from the tier two database, and then finally the tier one database. The process is shown in figure 5.3.3

5.3.4 Performance of offline computation

The response time of this algorithm is very good. On a set of 1000 randomly chosen routes the average response time was 0.04 seconds through the entire transit graph. A multi-threaded implementation of the ParetoPath algorithm responded to the first stage requests in parallel, the second and third stages are rapid BerekleyDB accesses.
The problem with this approach however is that the precomputation while large requires storing a vast amount of data to disk. In terms of the file sizes of the tier one and tier two databases for long distance trains the file sizes are 21 Gb for a large system like the New York Subway, and 43 Gb for the tier two computes.
The offline precomputation stage takes a very long time to complete. In a graph representing the transit network in the Greater Boston and New York City area the precomputation stage took 28 hours on cluster with 20 machines, each with a 1 GHz Pentium III CPU. Surprisingly, writing the

data to the database took 37 hours. With a graph that included all local bus lines, the size of the databases would be drastically larger, in the order of terabytes. This would be the case even if one were to not store the tier one solutions but just the tier two paths.

5.4 Speeding up online response using Fare Region restriction

The two-tier system described above works very well online. However it has several disadvantages that make it an unworkable system for practical use. There is a middle ground where once again precomputation can help in improving the response time of the ParetoPath algorithm. The problem with the previous approach is that it attempted far too much precomputation and storage of data. A second approach, along the lines of restricting the search horizon as in [35, 14] is described in this section.
The transit graph G naturally decomposes into its constituent sub-graphs on a fare region by fare region basis. Its resemblance to Bellman-Ford-Moore implies every vertex in the graph is scanned at least once, although for a given source and destination, many vertices will never contribute to a Pareto path. The approach taken to speeding up the online algorithm here is to prevent exploration in fare regions that can never lead to a Pareto path.
Using AllTimesParetoPath algorithm on G for every transfer point in

the graph, we can learn, in short order, which fare regions are involved in generating Pareto paths from one fare-region to another. We store, in a table, for every source and destination fare region, the set of all fare-regions traversed when computing the Pareto paths. If the number of fare-regions in G is n then the table has n^2 entries. If u and v are fare-regions then each entry (u, v) is a *guiding set* containing the values of the fare-regions traversed in creating a Pareto-path from fare-region u to fare-region v. The ParetoPath algorithm uses these guiding sets in determining when to call Expand. It is only necessary to check the guiding set table when the call to expand is about to consider edges that are global transfer edges. The modified version of the ParetoPath algorithm using guiding sets is shown in figure 5.4

This approach drastically reduces the amount of scanning and hence disk usage necessary to generate itineraries. The search horizon of the algorithm can be further restricted using any of the techniques discussed in the papers referenced above. One of the reasons that the two tier approach was initially explored was that for a large number of itineraries requested, there are usually more than two fare regions involved. This implies that the ParetoPath algorithms spends a fair amount of time making its way from one exit point to another. During the precomputation stage that creates the guiding sets, the AllTimesParetoPath algorithm could also save the sets of vertices along a the Pareto paths between two fare regions. With this information, when there are more than two fare regions involved

```
ParetoPath(G, S, D, t^START)
 1  Q.InitializeSPT(S)
 2  while Q.NotEmpty
 3  do i ← Q.Select()
 4     for j ∈ Adj[i]
 5     do if fare_region(i) ≠ fare_region(j)
 6           then if fare_region(j) ∉ GuidingSets(S, D)
 7                   then continue
 8        P*_j ← Extend(P_i, j)
 9        P_j ← RemoveDominated(P_j, P*_j)
10        if ∃p ∈ P_j s.t. p is not marked scanned
11           then Q.Insert(j)
12        MarkScanned(P_i)
13     P_D ← RemoveDominated(∪_{k∈D} P_k)
14  return P_D
```

Figure 5.10: The guided ParetoPath algorithm uses the GuidingSets procedure to steer the search away from fare-regions that are never used in generating a Pareto path.

in generating the Pareto paths a GuidingVertices predicate could also guide the search through intermediate fare regions. Storing this set of vertices in memory is simple enough and the size of these guiding vertex sets is relatively small.

This chapter has shown how to effectively solve the itinerary problem and a number of its variants. The technique described, for the most part, leaves the basic ParetoPath algorithm largely unchanged. The use of the domination predicate is central to allowing the algorithm to adapt to the different tasks at hand as the technique of defining paths to be equivalent under a similarity relation is defined by the notion of dominance. Together these concepts allow paths to be expanded optimistically and subsequently pruned back to ensure correctness. The similarity relation, for example, is implicitly used to determine which paths are comparable under domination when solving the all times Pareto path problem. In another example, it is possible to solve the transfer minimization variant of the problem without resorting to reintroduction of more direct solutions, if the similarity relation for edges was based on where a path might lead along an edge. However that approach was not explored because it would have led to too many intermediate, non-dominated, solutions.

For a high capacity server the best approach is currently to use the guided variant of the ParetoPath algorithm. It is easily optimized and can take advantage of a number of search-horizon restricting techniques to speed its search. Avoiding the large amounts of offline computation needed by the two-tier approach allows the service to adapt to real-time updates in the data. Also, since the algorithm is sensitive to poor data, which is all too common when constructing graphs from many different sources, correcting errors often means rerunning the offline computation. Since there are two

stages of offline runs this can take a long time [2]. For example, where an edge weight has changed in a fare region, the GuidingSets predicate could choose to be cautious and force search through redundant fare regions.

5.5 Other Issues

The section focuses on two other techniques used to further improve the performance of the ParetoPath algorithm.

5.5.1 Reducing disk activity

Since disk activity affects the performance of the system more than any other operation in the ParetoPaths algorithm we aim to prevent as much redundant exploration through the graph as possible. In transit graphs for every pair of vertices i and j that are adjacent to each other there are edges in both directions. Exploiting this fact we guard against expanding a path p toward a vertex that is already on that path. This is possible because all of our cost functions monotonically increase as the path grows longer, therefore going back to a node, already visited, can never improve a label. This observation leads to an obvious addition to the code, a simple guard condition OnPath, to see if a vertex is already on the path, precedes each call to Expand to prevent exploration from a node. The OnPath guard can be replaced with a less expensive heuristic based predicate NoPredecessor. This only predicate only checks to see if the

Simple	23.3 seconds	
Similar Sets	3.2 seconds	
Precomputation	0.38 seconds	MySql server
Fare region restriction	0.9 seconds	

Table 5.1: Peformance of the the ParetoPath variants.

predecessor of the current path is the same as the node we are about to explore. This approach was suggested because for the most part the graph is extremely sparse and often a vertex is adjacent to only two other vertices.

The performance increase in terms of time to compute the number of routes is shown in table 5.5.1. In these experiments the graph is scaled back to avoid the use of public bus systems, so that we may study the effects of a disk based system verses one where the entire database is in main memory. The total size of the database without public bus systems is approximately 5 GB.

It is not surprising to see that the naïve version of the ParetoPath algorithm does poorly when computing an itinerary. Profiling the run reveals over 99% of the process time is spent in disk access. The similar edge sets version of ParetoPaths is much faster, while adding the OnPath presents a further improvement. Surprisingly, the LastVisited heuristic leads to a drop in performance over the OnPath performance. Not surprisingly, placing the entire database into memory and thereby avoiding any disk access result in the fastest performance of all. Here the

OnPath heuristic still produces the fastest results but only by a very small margin. This is explained by the fact that fewer intermediate solutions are created before the domination check over the simple ParteoPaths with similar sets, whereas the LastVisited guarded variant despite being far simpler still allows more solutions to be created and the overhead of object creation greater than in the OnPath variant.

5.5.2 Reducing the cost of memory management

As is the case with many graph algorithms the cost of memory management causes a significant overhead to computation. Unlike the case with disk access however, there is little that can be done to avoid using memory. One can, however reduce the amount of overhead significantly by using memory pooling strategies. The ParetoPath algorithm is implemented using C++ and a Pareto optimal path is represented by a linked list of `ParetoPath` objects. Each object contains a pointer to its predecessor and a copy of the last edge traversed. Each object instantiation results in a call to the system `malloc`. In applications where there is a large amount of object instantiation and destruction it is far more efficient to declare a memory pool of pre-instantiated objects and then allocate and deallocate memory by checking out and returning objects to a free pool. This common strategy is outlined in a number of texts on object oriented programming [5]. However, there is still a fair amount of work that needs to be done in managing this pool.

In implementing the ParetoPath algorithm we use an unusual memory pooling strategy that makes the cost of instantiating and reclaiming space negligible, this gain however comes at the cost of efficiency in reclaiming memory. During the course of scanning a vertex only a small percentage of the paths created are Pareto paths, most are dominated immediately and are therefore instantly destroyed. Those that are assigned to the labels for a vertex need to persist for the duration of the run. The implementation uses two pools the first `Permanent`, for the label sets, and a second pool `Temporary`, for the temporary solutions.

The pools are pre-allocated when the system initializes, each pool being a large contiguous block of memory. Each pool maintains a single pointer to the first free byte of memory. Space allocation for each object is carried out by overloading the classes placement `new` operator to claim as many bytes as needed from the pool and then to increment the pointer to the first free byte. During the course of execution objects that are created by Extend are assigned from the temporary pool. After the call to RemoveDominated the undominated solutions are copied to the permanent pool. The implementation does not contain any memory reclamation code other than to reset the pointer to the first free byte in `Temporary` to the beginning of the pool. In a server application where the use of ParetoPath is reentrant the memory allocated on the permanent pool is reclaimed in the same manner.

The gains from memory pooling are in line with those reported in [5].

There is essentially no overhead, in terms of procedure calls, in allocating and reclaiming space in the graph. These gains however are comparatively small in relation to the gains from reducing disk access.

Chapter 6

Conclusion

This thesis has described the algorithmic toolkit we have developed to solve the itinerary problem. Solving multi-criteria optimization problems for transit networks in the presence of response time constraints is achievable, and the algorithms here have proved themselves in a commercial environment. Before summarizing our work, this chapter will discuss how to extend these techniques in future work.

6.1 Future directions

As the size of the transit graph grows so does the need for further optimization to speed up the response time of the online algorithm. This requirement becomes particularly important when dealing with transit networks that represent bus systems. Bus graphs are far more complex than those that describe railway and other fixed guide-way modes of

transport. Bus systems tend to be highly interconnected and therefore less sparse in nature suggesting that more work needs to be done in implementing a queuing strategy that minimizes scanning. The topological ordering models of Goldberg and Radzik prove more robust in minimizing scanning for different classes of graphs than the simple FIFO model of the Bellman-Ford-Moore algorithm.

Our transit graph decomposed naturally into individual sub-graphs (components) based on fare-regions. To further eliminates wasted exploration of the graph in the fare-region restriction techniques, we used these components to determine whether an online search should scan a set of vertices when extending paths. However, the size of some of the components are disproportionately large. The New York City subway for example has over 25% of all edges in the graph. If the components were more evenly sized the performance in the larger fare-regions of the graph would be further reduced. Preliminary work suggests that if we were to reduce the granularity of the components we could improve the online performance even further. However, smaller component sizes mean more transfer points and therefore larger demands on the precomputation process and requires that the test for whether to expand the search in a certain direction be executed more often.

More work needs to be done in determining the balance between the increase in online performance from the reduced scanning of nodes and the overhead of testing whether it is necessary for the search to enter a

component. One approach to managing this balance is to recursively subdivide the graph and use the recursively defined components to avoid searching redundant scans.

There are many other time-dependent multi-criteria optimization problems that this thesis does not consider that can be solved using the techniques described in this thesis. Many of these ideas come from interviews with executives in transit agencies. One of the more obvious itinerary planning applications is giving a desired arrival time in the problem instead of a departure time. To solve this problem our notion of time, and how to extend a path would need to be reversed. Minimizing the amount of time spent waiting during transfers is another factor that can be added to the domination predicate. We could expand the domination predicate to say that if two solutions are identical in all of their costs neither dominates the other unless one of the solutions involves less waiting time than the other. Other future areas of work can involve mode-restriction. For example, one may want to travel using only buses or trains but never a subway. This problem is simple enough to solve if there is no precomputation involved in speeding up the online response time. However when using search restriction techniques like the ones discussed above, more than one set of precomputes may be necessary, or the precomputation would need to maintain multiple sets of information, one for each restricted mode.

6.2 Final Remarks

The work here compares favorably to that of other systems such as the HaFas system (that runs the DeutchBahn site for European regional and inter-city rail travel), the Transcom system (that powers the passenger advisory systems for the Southern California transit authorities), or even the Orbitz system (for flight itinerary planning and booking). Many of these systems use novel technologies, each highly tailored to their own problem domain. Transcom offers an extremely efficient routing engine that utilizes heuristics to solve single criteria shortest path problems on a single Pentium class machine. The Orbitz technology runs on a cluster with over a thousand servers to handle the traffic levels that they achieve. Each query takes an average of 10 seconds to complete. All of these systems, like the ParetoPath variants introduced in Chapter 5 rely on static analysis of the transit graph to achieve their online performance goals.

In designing the ParetoPath algorithm we have chosen to stay close to the standard Bellman-Ford-Moore model despite trying to solve a multi-objective problem. This strategy has allowed us to leverage many of the existing optimization techniques already in the literature. The techniques introduced in this thesis have worked well in reducing the complexity of the algorithm in practice. In this respect the idea of reducing the graph to sets of similar edges based on an equivalence relation has proved effective in speeding up the computation, even when the edge set is stored on disk.

A common technique, used in many of the algorithms we have developed is the ability to use a domination predicate not only to minimize the number of solutions kept at any stage of the computation but to implement the different variants of the ParetoPath algorithm. Where appropriate, the domination predicate is specialized to allow the same basic Bellman-Ford-Moore framework to solve different problems. Some good examples of this technique are the addition of stop priorities to the domination predicate to allow solutions that start with nearby stops to persist even when there are other technically superior solutions, and the All Times Pareto Path algorithm that by adding the *start_stop* condition to the predicate restricts the sets of solutions allowed to dominate each other.
In many ways the transfer problem that presented itself when we added the directness measure f^{XFR} to the domination predicate could also be dealt with if we could decide ahead of time, which sets of solutions can dominate each other. One approach might be to define a similarity relation that placed edges in a partition to guarantee equivalent behavior when transferring trains. In this case domination would only be allowed between similar sets. While this approach may have no need for direct edges as, our solution did, we feel that the number of non-dominated solutions at each vertex would become too large to make the algorithm practical.
Another advantage of following the general framework of label correcting algorithms is that the algorithms presented here lend themselves to a vast array of optimization techniques discussed in the literature. The search

restriction and node selection techniques discussed in [14] can be easily applied here. Static analysis of the graph, also a common practice to speed searches in graph algorithms also contribute a great deal to improving the performance. Along these lines the fare region restriction technique in Chapter 5 has also proved very successful.

Some techniques, such as the precompute intensive two tier variant, described in Chapter 5 , have proved less successful in practice. While extremely fast in terms of online performance, the time taken for the precomputes proves to be impractical. In this approach the sizes of the precomputed solution set for a moderately sized subset of the transit network modeled grew to over 10Gb for some of the larger fare regions, and over 50Gb for the second tier solutions. Indexing a database of that size can take an extremely long time. On a dual 1GHz Pentium III server with a fast RAID subsystem the indexing operations alone took close to 24 hours.

In contrast, the fare region restriction method of 5.3.4 offers vast improvements in online speed but requires a relatively small precomputation phase.

In closing, we believe that the correct approach to implementing a solution to the itinerary problem should be to run the ParetoPath algorithm on similar edge sets using a set of the restriction techniques like fare-region restriction. Guiding the search through the reduced graph meets the online performance requirements of a practical system without the cumbersome

overhead of an intensive precomputation phase. As pointed out in the survey section, most of literature aimed at improving the performance of shortest path list type algorithms focuses on reducing the amount of scanning. Where edge sets are stored on disk the work done in scanning vertices must be kept to a minimum. We feel that the research presented here takes a good first step at creating an efficient multi-criteria itinerary problem solver.

Bibliography

[1] Rakesh Agrawal and H. V. Jagadish. Algorithms for searching massive graphs. *TKDE*, 6(2):225–238, 1994.

[2] G Berger. Private communication. Conversation about the impracticality of the two-tier approach, 6 2001.

[3] D Bertsekas. A simple and fast label correcting algorithm for shortest paths. *Networks*, 23:73–709, 3 1992.

[4] Gerth Stlting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries, 9 2001.

[5] D Bulka and D Mayhew. *Efficient C++: Peformance and Programming Techniques*. Addison-Wesley, 2000.

[6] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Records*, (1645):170–175, 1998.

[7] Cherkassky, Goldberg, and Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *SODA: ACM SIAM Symposium on*

Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms), 1994.

[8] L Cooke and E Halsey. The shortest route through a network with time-dependent internodal transit times. *Journal of Mathematical Analysis and Applications*, (14):492–498, 1966.

[9] J. Hao D. Goldfarb and S-R Kai. Efficient shortest path simplex algotrithms. *Operations Research*, 38:624–628, 1990.

[10] C de Marcken. Inside orbitz. Internet posting, 1 2001. Article appeared on Slashdot.com about how Orbitz built their itinerary planning system in LISP.

[11] R. Dial. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633, 11 1969.

[12] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerishe Matematik*, 1:269–271, 1959.

[13] R. Glover F. Glover and D. Klingman. Computational study of an improved shortest path algorithm. *Networks*, 14:25–36, 1984.

[14] D. Wagner F. Schulz and K Weihe. Dijkstra's algorithm online: An empirical case study from public railroad transport. *LNCS:Algorithm Engineering*, 1668:110–123, 1999.

[15] A. V. Goldberg and T. Radzik. A heuristic improvement of the

bellman-ford algorithm. *AMLETS: Applied Mathematics Letters*, 6, 1993.

[16] A. Guttman. "r-trees: a dynamic index structure for spatial searching". *In Proceedings ACM SIGMOD*, pages 47–57, 1984.

[17] P Hansen. Bicriterion path problems. in multiple criteria decision making: Theory and applications. *Lecture Notes in Economic and Mathematical Systems*, 177:109–127, 1980.

[18] E. L. Johnson. A note on dijkstra's shortest path algorithm. *Journal of the ACM*, 20(3):385–388, 1973.

[19] A. Kershenbaum. A note on finding shortest path trees. *Networks*, 11:399–400, 1981.

[20] P Klein. From the tennis club to the golf course in .005 seconds or less: a commercial shortest-path engine that employs preprocessing. NYU Theory Day, 04 2002.

[21] S. Pallottino M. Nonato and B. Xuewen. SPT_L shortest path algorithms: review new proposals and some experimental results. Technical Report 16, Universita Di Pisa, 7 1999.

[22] M.Garey and D. Johnson. "strong" np-completness results: Motivation, examples and implications. *Journal of the ACM*, 25(3):499–508, 7 1978.

[23] M.Garey and D. Johnson. *Computers and Intractability - A guide to the Theory of NP-Completeness*. Freeman, San Fransico, 1979.

[24] M. Muller-Hannemann and K. Weihe. Pareto shortest paths is often feasible in practice. volume 2141, pages 185–197, 2001.

[25] Orda and Rom. Minimum weight paths in time-dependent networks. *NETWORKS: Networks: An International Journal*, 21, 1991.

[26] Ariel Orda and Raphael Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[27] Stefano Pallottino and Maria Grazia Scutella. Shortest path algorithms in transportation models: classical and innovative aspects. Technical Report TR-97-06, 14, 1997.

[28] U. Pape. Implementation and efficiency of moore algorithms for the shortest root problem. *Mathematical Programming*, 7:212–222, 1974.

[29] H. Safer and J. Orlin. Fast approximation schemes for multi-criteria combinatorial optimization. Working Paper, 1995.

[30] K. Scott and D. Bernstein. Solving a best path problem when the value of time function is nonlinear. Technical report, New Jersey TIDE Center, New Jersey Institute of Technology, 11 1997.

[31] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The

r+-tree: A dynamic index for multi-dimensional objects. In *The VLDB Journal*, pages 507–518, 1987.

[32] A. Siegel and R. Cole. Introduction to algorithms. Draft of textbook on algorithms used internally at New York University.

[33] R. Tarjan. Shortest paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[34] R.E. Tarjan. Data structures and network algorithms. In *SIAM: ACM SIAM Philadelphia, PA*, 1983.

[35] Karsten Weihe. Reuse of algorithms: still a challenge to object-oriented programming. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 34–48. ACM Press, 1997.

[36] A. Ziliaskopoulos. *Optimum path algorithms on multidimensional networks: Analysis, design, implementation and computational experience*. PhD thesis, University of Texas at Austin, 1994.

[37] A. Ziliaskopoulos and W. Wardell. Design and implementation of an intermodal optimum path algorithm for multimodal networks with dynamic arc travel times and switching delays. *European Journal of Operational Research*, 125:486–502, 2000.