

**Improved Methods for Solving Traffic Flow  
Problems in Dynamic Networks**

by

Nathaniel J. Grier

B.S. Civil Engineering, Massachusetts Institute of Technology (2000)

Submitted to the Department of Civil and Environmental Engineering  
in partial fulfillment of the requirements for the degree of

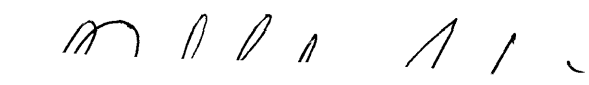
Master of Science in Transportation

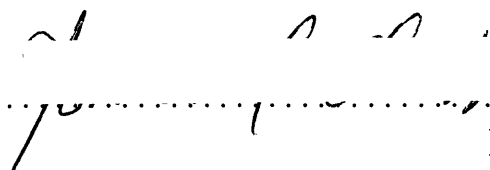
at the

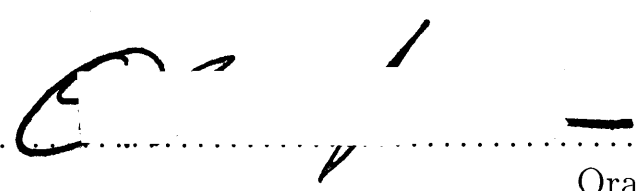
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

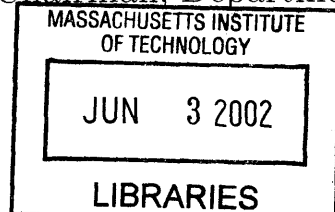
June 2002

© Massachusetts Institute of Technology 2002. All rights reserved.

  
Author .....  
Department of Civil and Environmental Engineering  
May 24, 2002

  
Certified by .....  
Ismail Chabini  
Associate Professor of Civil and Environmental Engineering  
Thesis Supervisor

  
Accepted by .....  
Oral Buyukozturk  
Chairman, Department Committee on Graduate Studies



BARKER



# Improved Methods for Solving Traffic Flow Problems in Dynamic Networks

by

Nathaniel J. Grier

Submitted to the Department of Civil and Environmental Engineering  
on May 24, 2002, in partial fulfillment of the  
requirements for the degree of  
Master of Science in Transportation

## Abstract

Dynamic networks are pervasive, present in many transportation and non-transportation contexts. We present improved methods for solving two of the primary problems in dynamic networks: dynamic shortest paths and the Dynamic Network Loading Problem (DNLP). In each case we also propose a solution algorithm and an implementation of the algorithm.

We first explore the one-to-all dynamic shortest path problem for discrete time networks for all departure times. A new framework for the problem is proposed in which the problem is viewed as series of static reoptimization problems. By posing the problem in this manner, we are able to reuse the information regarding the shortest path trees calculated for earlier departure times. The results of computational tests are provided showing significant savings in computation times over traditional methods when the percentage of dynamic links is small.

We next present a method for achieving an exact solution to a class of the continuous-time and space model formulation of the DNLP. The model of a link is based on that originally proposed by Lighthill and Whitham and Richards. We characterize the network in terms of the traffic density on the roadway. This allows for the accurate modeling of discontinuities in the roadway, including queues, their dynamics and associated phenomena such as spillback, and temporary events in the network. The model is first presented for a stretch of highway without on- or off-ramps and subsequently extended to network topologies with multi-path flow. The densities at the network entrances are assumed to be stepwise constant and the density-flow relationships on the arcs are assumed to be concave and piecewise linear. Finally, we describe an implementation of the solution method and provide the results of its application to test networks.

Thesis Supervisor: Ismail Chabini

Title: Associate Professor of Civil and Environmental Engineering



## Acknowledgments

The author would like to thank Ismail Chabini for his insights and guidance over the past several years. He would like to thank the members of his research group for their assistance, motivation and friendship. And to his family he would like to express his love and thanks for their support and unwavering confidence in his ability to succeed. And lastly, he would like to thank Ariana Sutton, without whom, none of this would have been possible. Words could never express all that I owe to you.



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>   | <b>17</b> |
| 1.1      | Motivation for the Research . . . . .   | 18        |
| 1.2      | Objectives of the Thesis . . . . .  | 19        |
| 1.3      | Thesis Outline . . . . .  | 20        |
| <b>2</b> | <b>A New Approach to Compute Minimum Time Path Trees in FIFO<br/>Time Dependent Networks</b>                                    | <b>21</b> |
| 2.1      | Notation . . . . .  | 23        |
| 2.2      | Theory of Reoptimization . . . . .  | 24        |
| 2.2.1    | Previous Work . . . . .   | 24        |
| 2.2.2    | The Reoptimization Problem: A Basic Solution . . . . .  | 25        |
| 2.3      | Reoptimization in Dynamic Networks . . . . .  | 27        |
| 2.4      | The Dynamic Reoptimization Algorithm . . . . .  | 30        |
| 2.4.1    | Algorithm Description . . . . .   | 30        |
| 2.4.2    | An Example . . . . .  | 33        |
| 2.5      | Computational Results . . . . .   | 35        |
| 2.6      | Conclusions . . . . .   | 45        |
| <b>3</b> | <b>A Continuous Space and Time Representation of Dynamic Road<br/>Traffic Flows Consistent with Hydrodynamic Traffic Theory</b> | <b>47</b> |
| 3.1      | Description of the Approach . . . . .   | 50        |
| 3.1.1    | Blocks of Constant Density . . . . .  | 50        |
| 3.1.2    | Boundary Propagation . . . . .  | 51        |

|          |  |            |
|----------|--|------------|
| 3.2      | Modeling Roadway Discontinuities . . . . .               | 53         |
| 3.2.1    | Bottlenecks . . . . .                                    | 54         |
| 3.2.2    | Incidents . . . . .                                      | 55         |
| 3.2.3    | Expansions . . . . .                                     | 57         |
| 3.3      | The Network Model with Known Turn Percentages . . . . .  | 58         |
| 3.3.1    | Network Structure . . . . .                              | 59         |
| 3.3.2    | A Note on Notation . . . . .                             | 60         |
| 3.3.3    | Modeling Diverges with Known Turn Percentages . . . . .  | 61         |
| 3.3.4    | Modeling Merges . . . . .                                | 64         |
| 3.4      | The Network Model with Multi-Path Flow . . . . .         | 66         |
| 3.4.1    | Modeling Multi-Path Flow . . . . .                       | 67         |
| 3.4.2    | The Diverge in a Network with Multi-Path Flow . . . . .  | 68         |
| 3.4.3    | The Merge in a Network with Multi-Path Flow . . . . .    | 69         |
| 3.5      | Algorithm Description . . . . .                          | 70         |
| 3.5.1    | Pseudocode Implementation . . . . .                      | 70         |
| 3.5.2    | Java™ Implementation . . . . .                           | 78         |
| 3.6      | Loading Examples . . . . .                               | 80         |
| 3.6.1    | Stretch of Highway . . . . .                             | 80         |
| 3.6.2    | A Network Example . . . . .                              | 81         |
| 3.7      | Conclusions . . . . .                                    | 84         |
| <b>4</b> | <b>Conclusions and Future Directions of Research</b>     | <b>87</b>  |
| <b>A</b> | <b>A Static Shortest Paths Reoptimization Algorithm</b>  | <b>91</b>  |
| <b>B</b> | <b>Network Loading Algorithm Implementation Details</b>  | <b>93</b>  |
| B.1      | Class List . . . . .                                     | 93         |
| B.2      | Class Hierarchy . . . . .                                | 97         |
| <b>C</b> | <b>Examples Tested with DNLP Solution Implementation</b> | <b>101</b> |
| C.1      | Example Networks . . . . .                               | 101        |
| C.2      | Example Network File . . . . .                           | 104        |







# List of Figures

|     |  |    |
|-----|--|----|
| 2-1 | Pseudocode for the dynamic shortest paths reoptimization algorithm.  | 32 |
| 2-2 | Pseudocode for the UPDATE-PROJECTIONS called by the DYNAMIC-REOPT-<br>IMIZATION algorithm. . . . .   | 32 |
| 2-3 | A sample network used to demonstrate the fundamentals of the dy-<br>namic shortest path reoptimization algorithm. Link travel times are<br>given in Table 2.1. . . . .   | 33 |
| 2-4 | The total running time (reoptimization time plus initialization time)<br>of algorithm DR as a function of problem size is shown for multiple<br>percentages of dynamic links. Also shown is the running time for the<br>modified Dijkstra's algorithm (SSP). We note the essentially linear in-<br>crease in running time with problem size for both algorithms. . . . .   | 37 |
| 2-5 | The graph depicts a close-up of the running times shown in Fig. 2-4.   | 37 |
| 2-6 | In the above graph we show the total time spent in the reoptimization<br>routine as a function of problem size and the percentage of the links in<br>the network which are dynamic. For comparison we show the amount<br>of time spent running the iterative Dijkstra's algorithm. One sees that<br>even with the unoptimized implementation used in these tests, the<br>reoptimizer is faster than the iterative Dijkstra's algorithm for all but<br>the most dynamic networks. . . . . | 38 |
| 2-7 | In the above graph we show the amount of time spent in finding the<br>first change which affects $SPT(1)$ for each arc. The growth in running<br>time with respect to problem size is approximately linear. . . . .  | 39 |

2-8 The dependence of the running time of the algorithm DR on the number of arcs in the test network is show above, using a network with 3000 arcs. In this network 15% of the links are dynamic. . . . . 40

2-9 In the above graph we show the dependence of the algorithm DR on the number of arcs in the network, using a network with 1000 nodes. In this network 15% of the links are dynamic. . . . . 41

2-10 The above graph illustrates the variability in runtime of algorithm DR as a function of the time horizon, using a network with 1000 nodes and 3000 arcs. In this network 15% of the links are dynamic. . . . . 42

2-11 The run times of the four test algorithms are shown as a function of network size. In all cases 10 percent of the links were dynamic, the network had a time horizon  $T$  of 200, and contained three times as many links as nodes. . . . . 44

2-12 The run times of the four algorithms are shown with respect to the time horizon. In all cases the network consisted of 600 nodes and 1800 arcs, 10 percent of which were dynamic. . . . . 45

3-1 Determination of the boundary speed between upstream state  $U$  and downstream state  $D$ . Parts (a) and (b) of the figure represent the condition where the downstream density is above the critical density while the upstream is in free flow condition. Parts (c) and (d) depict the reversed situation; note the creation of the new state  $M$  between state  $U$  and state  $D$ . . . . . 53

3-2 Two density flow diagrams which illustrate the treatment of a bottleneck. 55

3-3 The space-time diagram representing an incident is shown on the right. On the left is the corresponding density-flow diagram. . . . . 56

3-4 Two density flow diagrams which illustrate the treatment of an expansion in the roadway capacity. . . . . 57

|      |  |    |
|------|--|----|
| 3-5  | In part (a) we show an allowable merge (above) and diverge (below).<br>In part (b) we show samples of disallowed junctions (left) together with<br>a proposed transformation into an allowable representation (right). . . . .   | 60 |
| 3-6  | The pseudocode description of the DNLP solution algorithm. . . . .   | 71 |
| 3-7  | The space-time diagram resulting from the example output and the<br>fundamental diagram for the arcs. . . . .  | 80 |
| 3-8  | Output from the example. In the lines concerning the creation of new<br>blocks, a unique identifier of each density block is output [its address in<br>memory]. Also note that DSDB refers to the downstream density block.<br>In addition, $x$ refers to the one dimensional position along the arc, $t$<br>refers to the time at which the event occurs and $A\#$ refers to the arc<br>number on which the event occurred. . . . . | 82 |
| 3-9  | A diagram of the test network, including arc numbers. Arcs 0 through<br>4 represent the highway, 9 and 10 on- and off-ramps, respectively, and<br>the remainder are arterial links. . . . .  | 83 |
| 3-10 | The network loading at $t = 0.062$ hours or just under 4 minutes. The<br>traffic has almost reached the edge of the network. . . . .   | 84 |
| 3-11 | The network loading at $t = 0.20$ hours or 12 minutes. We see that the<br>queue due to the bottleneck has grown substantially. . . . .   | 84 |
| 3-12 | The network loading at $t = 0.433$ hours or 26 minutes. The queue has<br>now almost disappeared. . . . .   | 84 |
| 3-13 | The network loading at $t = 0.500$ hours or 30 minutes. The network<br>has just returned to steady-state conditions. . . . .   | 85 |



# List of Tables

|     |  |     |
|-----|--|-----|
| 2.1 | Travel times on the network shown in Figure 2-3 as a function of time. In order to emphasize the changes in travel time, the table only includes entries for changed travel times; all travel times are assumed constant until the next entry in the table. . . . .  | 34  |
| C.1 | A summary of the examples used in testing the DNLP implementation.   | 102 |
| C.2 | A summary of the arc types used in the examples, including all relevant characteristics. . . . .   | 104 |
| C.3 | A summary of the density-flow relations used in the examples, including all relevant characteristics. Note that the underlying units in the first example differ from the remaining relations. . . . .   | 105 |
| C.4 | A summary of the events used in the testing of the networks. Note that the Duration and Delay columns apply to cyclic events: duration refers to the duration of the blockage and delay to the amount of time which passes after the blockage is removed before it reappears (therefore the sum of the two is equal to the period or cycle time of the event). As these only apply to cyclic events, these columns are blank in non-cyclic events. . . . . | 105 |





# Chapter 1

## Introduction

Dynamic networks are present in nearly all areas of transportation. This is particularly true in highway networks where the traffic conditions are constantly changing. Many areas of transportation rely on models to predict the state of a highway network under time-dependent conditions. Practitioners from the transportation planner to real-time traffic manager rely on efficient methods to predict the future state of the network.

One of the primary problems in a dynamic network is the determination of shortest paths. Because the state of the network is constantly changing, the shortest path between a given origin and destination must be calculated as a function of the departure time from the origin. This information is used not only for pre-trip planning (based on the assumption that the traveler is seeking the shortest path), but increasingly today in a real-time context of route-guidance while en-route to the destination.

A second major problem in dynamic networks is the Dynamic Network Loading Problem (DNLP). This problem seeks to provide information on the link dynamics – particularly link travel times – given a set of time-dependent origin-destination demands. Such a model can be used to predict how a traffic network will respond to an incident or how traffic flow will evolve over the course of a day based on historical data. By accurately modeling the queuing in a network, the DNLP is also important to vehicle emissions models. The DNLP is at the heart of the Dynamic Traffic Assignment problem, which seeks to assign origin-destination demand to a particular set

of paths in the network, usually a set of the shortest paths. In this thesis we explore both the dynamic shortest paths problem and the DNLP, and propose advances to each.

## 1.1 Motivation for the Research

In the case of the dynamic shortest paths problem, previous algorithms often attempt to solve the problem at hand without much thought of reusing the knowledge of the problem obtained during the solution, specifically shortest path information. While some algorithms have sought to use this information, it is normally only used to provide bounds on the length of the shortest path rather than incorporate it directly into subsequent solutions. As it is common for a network to have a substantial percentage of links where the travel time is constant or nearly so, we would like to be able to calculate a shortest path once, and then only recalculate it if the travel times along the arcs in the path change.

At the same time, because we use dynamic shortest paths algorithms in real-time situations, we rarely know the travel times along all the arcs in the network for all future times. Instead, we might be able to predict when an arc's travel time will next change, but no further. In this thesis we describe an algorithm which addresses both of these issues.

Previous discrete DNLP models approximate the behavior of the network. Time, and often space, are broken up into a finite number of intervals and behavior of the network is estimated at this level. While the accuracy of these models can be improved by increasing the resolution of the discretization, this increases the computational power needed to solve them. Not only do they suffer from this accuracy-computation trade-off, the approximations inherent in discrete models often result in additional errors because of their inability to truly model the dynamics of the traffic. A model which is continuous in space and time, however, can allow us to develop an exact solution to the problem. In addition to providing an exact solution, such models may actually perform less work than a discrete solution method. Just as the arcs may be

static in the dynamic shortest paths problem, it is often the case that the input data to the DNLP is constant over a time interval. Rather than explicitly calculate the network state at every point in time and every discrete location in space, we need only concern ourselves with those points at which the state changes.

While a few DNLP solution methods exist which treat space and time as continuous, they are limited in their scope, not modeling all aspects of traffic flow. While some do not model queues and their spillback, others are limited to network topologies consisting of a stretch of highway. Moreover, to be of use, a solution method to the DNLP must be implementable in order to solve the complex problems which are not practical to carry out by hand.

The primary motivation for the new DNLP model of this thesis, then, is to bridge an existing gap between realism and theory. Historically DNLP models have been forced to make a trade-off between these two aspects of the models. Macroscopic traffic theory relied on the volume-delay function to calculate link travel times. While such functions are relatively accurate when a link is uncongested, they are not when the traffic becomes congested and queues develop. On the other hand, practitioners who study the dynamics of flows on highways have long known that a relationship exists between the density of traffic, and the flow and the speed of vehicles on the roadway. Yet the complexity of these relationships is such that the exact solution techniques have been abandoned in favor of approximations. Thus by presenting a continuous time and space solution method to the DNLP which builds upon the observed properties of highway dynamics, we aim to bridge the gap between the model realism to capture the dynamic aspect of highway traffic and their tractability from a computational standpoint.

## 1.2 Objectives of the Thesis

The objectives of this thesis are to:

- present a framework for viewing the one-to-all dynamic shortest (minimum-time) path problem which improves upon existing algorithms when the network

contains a small percentage of links with dynamic travel times.

- develop a method which provides an exact solution to the continuous time and space model formulation of the Dynamic Network Loading Problem, based on the hydrodynamic theory of traffic flow, assuming that the input densities are stepwise and the link density-flow relationships are concave and piecewise linear.
- implement these methods and test them in order to verify their correctness.

### 1.3 Thesis Outline

The goal of this thesis is to formulate and implement improved methods for solving problems in dynamic networks. We focus on two of the primary problems in dynamic traffic networks: dynamic shortest paths and the Dynamic Network Loading Problem.

In Chapter 2 we address the one-to-all dynamic shortest paths problem. We focus on the variant of the problem where only a small percentage of the links in the network have dynamic travel times. After providing a brief overview of reoptimization in static networks, we describe how the dynamic shortest paths problem can be viewed as a series of static reoptimization problems by introducing the concept of the projection. We walk through an example using this solution method, followed by comparison of the computational efficiency of a sample implementation to existing solution methods.

In Chapter 3 we present a space and time continuous model which provides an exact solution to the Dynamic Network Loading Problem, given certain input data. We discuss the advantages of this model over other well-studied methods. The model is first described for a stretch of highway, followed by a discussion of how network topologies are modeled. We provide an extensive description of an algorithm based on this method and an implementation of this algorithm. We end by providing numerical results from a sample network loading.

We conclude in Chapter 4 by discussing the strengths and weaknesses of the algorithms presented in Chapters 2 and 3 and presenting some directions for future research.

## Chapter 2

# A New Approach to Compute Minimum Time Path Trees in FIFO Time Dependent Networks

The problem of shortest paths in dynamic networks has been studied extensively in recent years and several solution algorithms have been proposed. In transportation, shortest path problems lie at the heart of the route guidance dynamic traffic assignment (DTA) problems. The shortest path problem also arises in many other application fields including telecommunications. As these networks are dynamic in nature, we desire algorithms which can operate on dynamic networks. There are many sub problems of the minimum time path problem. The solution of the one to all shortest path problem for all departure times via an iterative Dijkstra's method is a celebrated result [17, 22, 1]. In this chapter we shall focus on the one to all minimum time path problem for all departure times in FIFO networks.

The First-in-First-Out (FIFO) property for an arc holds if and only if an individual leaving the source node cannot arrive at the end node earlier by departing later. For the network to be FIFO, this property must hold for all departure times on all arcs. If the network is FIFO, it follows that the arrival times at each destination node must be monotonically increasing.

When considering dynamic networks, one of the first observations is that, in prac-

tice, many dynamic networks are not fully dynamic. That is, not all the links have time-dependent travel times, and for those that do, a significant amount of time may elapse between changes in their travel time. Recent developments in continuous-time solutions to the dynamic shortest paths problem can reduce the computational time needed to solve this problem [8]. Chabini et. al. [7] proposes an algorithm which uses previous results as bounds in order to speed up the computation for subsequent departure times. Yet despite these potential gains, most of the work in the literature investigating the direct reuse of shortest path trees has been limited to the investigation of static shortest paths.

The study of static shortest paths reoptimization aims to find a new shortest path tree following a change in the network, using a previous shortest path tree as the basis for such calculations. Reoptimization algorithms traditionally have focused on changes to a single arc travel time, though some work has been done to find efficient methods of finding a new optimal tree given several simultaneous changes in link travel times [29, 19].

In this chapter, we propose an algorithm which utilizes the classical notion of reoptimization to reduce the computational time necessary to solve the one to all shortest path problem in discrete-time dynamic networks for all departure times. The field of reoptimization has undergone many advances in recent years. To this end, we propose a framework via which a dynamic shortest path problem can be transformed into a series of static reoptimization problems. The framework is such that any such algorithm may be used, allowing the problem of dynamic reoptimization to benefit from advances in the study of static reoptimization.

This chapter is organized as follows. In Section 2.1 we summarize the notation we use in the chapter. We then provide a brief background on the theory of static reoptimization in Section 2.2. In Section 2.3 we describe how the dynamic shortest path problem may be viewed as a series of static shortest paths reoptimizations. This is followed by a description of a solution algorithm and an example solution in Section 2.4. Finally, we give the results of some basic computational testing in Section 2.5

## 2.1 Notation

In our discussion we will use the following notation. Let  $G = (N, A)$  be a graph, where  $N$  is the set of nodes and  $A$  the set of arcs. Let the number of nodes in  $G$  be  $n = |N|$  and the number of arcs  $m = |A|$ . We associate with each arc  $(i, j)$  a travel time  $d_{ij}(t)$ , where  $t$  denotes the time of entrance onto the link. The travel time from the origin node  $q$  to a node  $i$  departing the origin at time  $t$  is denoted by  $d_i(t)$  and the arrival time at node  $i$  when departing the origin at time  $t$  by  $a_i(t)$ ; thus  $t + d_i(t) = a_i(t)$ . For a node  $i$ , let  $A(i) = \{(i, j) : \forall j \in N, (i, j) \in A\}$  and  $A(i) = \{(j, i) : \forall j \in N, (j, i) \in A\}$ . Let the set of nodes whose shortest path includes node  $i$  be indicated by  $R(i)$ ; in terms of the shortest path tree,  $R(i)$  is identical to the subtree rooted at node  $i$ . We utilize the concept of the *reduced cost* of an arc  $(i, j)$  at time  $t$ , with respect to travel time, such that  $\bar{c}_{ij}(t) = d_i(t) + d_{ij}(a_i(t)) - d_j(t)$ ; while the “reduced travel time” is a more accurate description, we will use “reduced cost” as we feel it is more intuitive for most readers.

In the following discussion, we make use of a time horizon  $T$ , the time after which we are no longer interested in changes in the network so that arc travel times are assumed static. Furthermore, we take the first time for which the shortest path tree is known  $t_0$  to be 0 without loss of generality. For each arc  $(i, j)$  the number of times that the arc changes travel time between  $t_0 = 0$  and  $T$  is referred to by  $K_{ij}$ . We denote the time of each such change by  $B_{ij}(k)$ , with  $B_{ij}(0) = 0$ ,  $\forall (i, j) \in A$ . Thus by  $d_{ij}(B_{ij}(k))$  we refer to the travel time along arc  $(i, j)$ , departing  $i$  at the time of the  $k$ th change in travel time along the arc; for notational simplicity we may also refer to this travel time as simply  $d(B_{ij}(k))$ . Note that the travel time on arc  $(i, j)$  is assumed constant for the interval  $[B_{ij}(k), B_{ij}(k + 1))$ . Additionally, we will sometimes refer to the  $k$ th change on arc  $(i, j)$  as  $k_{ij}$ . We will also sometimes refer to the act of reoptimizing the shortest path tree with respect to a change in travel time as “processing a change”. Finally, we will introduce additional notation, as necessary, to refer to concepts developed in the following discussion.

## 2.2 Theory of Reoptimization

By reoptimization we refer to the calculation of new shortest paths following the change in a characteristic of the network. Normally this refers to a change in travel time or cost, but the algorithm discussed in this chapter is valid for topological changes as well. These change are simply reflected as changes in cost and travel time: the removal of an arc can be modeled by setting its travel time to infinity. From this point on, though, when referring to a change in the network, we mean simply a change in travel time.

Reoptimization seeks to reuse information about the network and its shortest paths in order to efficiently calculate the new shortest path tree. This is usually accomplished by reoptimizing with respect to the shortest path tree which was optimal prior to the change(s) in the network, referred to herein as the “base” tree. One can seek to reoptimize with respect to a single change in the network or several at a time.

### 2.2.1 Previous Work

While the idea of reusing previously computed shortest paths is not new, most of the previous work in the area has focused on efficient methods to solve subproblems of the general reoptimization problem. Gallo [20], for instance, has proposed an efficient method for recalculating the shortest path tree when the origin node has changed or the cost of one arc is reduced. Another subproblem of the shortest path reoptimization has been addressed by Fujishige and his suggestion for an efficient algorithm to update the shortest path tree when the set of arcs incident to a common node is given new costs, lower than those previous [19]. Other research into the reuse of known information to more efficiently solve the shortest path problem has focused on the utilization of previous results as lower and upper bounds for the new shortest path lengths. Glenn [21], for example, recently introduced a new method for reusing known shortest paths to find improved travel time bounds in dynamic networks . While the method efficiently reuses previous results, it does not directly incorporate this information into subsequent shortest path trees.



There has also been a renewed interest in a more general treatment of the reoptimization problem, such as that of Pallottino and Scutellà [29]. They propose a generic algorithm for reoptimizing shortest paths in a static network. Their work draws on the earlier work by Fujishige and Gallo [19, 20]. To the best of our knowledge, however, there has been no research on the reoptimization of dynamic networks.

### 2.2.2 The Reoptimization Problem: A Basic Solution

Before discussing the treatment of reoptimization in a dynamic network, we provide here a brief description of one such reoptimization algorithm for shortest paths in static networks. As our solution to the dynamic shortest path problem relies on the use of a static reoptimization algorithm, it is important that the reader be familiar with the static problem.

We process such changes one at a time, reoptimizing the shortest path tree after updating the arc's travel time on an arc. The algorithm classifies each change with respect to two parameters: 1) whether the changed arc was in the shortest path tree prior to its change in travel time; and 2) the new reduced cost for the arc, where the reduced cost is in terms of travel time (See Section 2.1 for more on notation). By defining the change in terms of these two parameters, we identify four categories of changes.

In the case that the arc is not in the tree and its new reduced cost is non-negative, clearly the shortest path tree does not change and no additional work to reoptimize the tree is necessary.

If the reduced cost of an arc  $(k, l)$  becomes negative, by the Bellman-Ford optimality conditions, the arc must now be in the shortest path tree. (See, for instance, [2] for further discussion of these optimality conditions.) This is for both the categories in which the arc was previously in the shortest path tree and that in which it was not; for both categories the reoptimization procedure is the same. The first step is to add the arc to the shortest path tree, if necessary. Next, the minimum travel times  $d_i(\cdot)$  to the nodes in  $R(l)$  must be updated. Finally, one must recalculate the reduced costs in this subtree, updating the shortest path tree beneath this node as necessary.

Methods for reoptimizing this subtree have been proposed in [28, 27, 29] and are the subject of ongoing research.

The fourth category occurs when the travel time along an arc  $(k, l)$  in the shortest-path tree increases. In this case, the travel times to the nodes in  $R(l)$  must be updated as must the reduced costs for each arc in this subtree. The optimality of the current shortest paths to all nodes in  $R(i)$  is unknown: given the increased travel time on  $(k, l)$ , the nodes in  $R(i)$  might now be reached earlier by using a path which does not utilize  $(k, l)$ . In order to recalculate the optimal shortest path tree, it is necessary to examine the new reduced costs of a greater number of arcs than for the previous categories of changes as the arcs connecting the optimal tree to the subtree must also be examined. The process of reoptimization can be the similar to that used for the case when an arc travel time decreases.<sup>1</sup> While the use of a similar procedure simplifies the implementation of a static reoptimization algorithm, it is not the most efficient manner of reoptimization for this type of change (See [28, 27, 29] for instance).

The method for the treatment of the four cases described above provides a means of finding the new optimal shortest path tree following a change in travel time in the network. Because the shortest path tree is optimal following the processing of each such change, a set of changes to the network may be processed in any order and will ultimately yield the new optimal static shortest path tree. We note that this observation is key and shall return to it in later discussion. Although the treatment above was general, we do not wish to belabor the point of static reoptimization as the methods for such are numerous and complex. For further illustration, we refer the reader to Appendix A for a psuedocode implementation of a simple static reoptimization algorithm which was developed as part of this research.

---

<sup>1</sup>For example, in the case when an arc travel time decreases, one starts at the root of the affected subtree, rehanging as necessary. In this case, one rehanges not from the root of the subtree but from any of the nodes in the optimal tree which connect to the subtree.

## 2.3 Reoptimization in Dynamic Networks

In a dynamic network there are several shortest path trees, each corresponding to a departure from the origin at some time  $t$ , which we denote by  $SPT(t)$ . Given the tree for time  $t$ , subsequent changes in arc travel times may be such that  $SPT(t+1)$  is disjoint from that at time  $t$  or identical. In networks with a small proportion of dynamic links, one would expect the subsequent shortest path trees to be quite similar.

Given the shortest path tree of the previous time index  $t$ , the key, then, is to determine which changes in arc travel times will affect a departure at time  $t+1$ . To aid us in this process we introduce the concept of a projection.

**Definition 1** *We define the projection,  $p_{ij}(k)$  of the  $k$ th change on arc  $(i, j)$  as the earliest departure time  $t$  such that  $a_i(t)$  is later than the time at which this change occurs  $B_{ij}(k)$ :*

$$p_{ij}(k) = \min_{a_i(t) \geq B_{ij}(k)} t.$$

In words, the projection allows us to identify the first departure time to experience a change in the travel time along an arc (assuming that only shortest paths are used). By utilizing the idea of the projection, we can transform the dynamic shortest path problem into a series of static reoptimization problems: we can identify a series of changes affecting a single base tree, with one such tree for each departure time. To do so we must first answer two questions: 1) what shortest path tree is reoptimized when processing a change (or which is the ‘base’ tree and how can it be determined); and 2) for a given departure time, which changes should be processed. We find the answer to our first question by realizing that if there were no changes in the network affecting a departure at time  $t+1$ , then  $SPT(t+1)$  must be identical to  $SPT(t)$ . It follows, then, that  $SPT(t)$  should be used as the initial optimal tree for changes affecting departures at time  $t+1$ .

To answer the second question, we see that by Definition 1, in order to determine the shortest path tree for a given departure time  $t+1$ , the reoptimization need only consider those changes whose projections are equal to the current departure time

$t + 1$ . This conclusion is only valid, however, if we reoptimize in increasing order of departure time, using  $SPT(t)$  as the initial shortest path tree for departure time  $t + 1$ .

The projection, then, allows us to transform the dynamic reoptimization problem into a series of static reoptimization problems. Given an initial shortest path tree, we have an idealized algorithm for the dynamic reoptimization problem which we summarize below.

### **Idealized Dynamic Reoptimization** ( $G, q, T$ )

```

 $SPT(0) \leftarrow SSP(G, q, 0)$ 
for  $t$  from 1 to  $T$ 
   $SPT(t) = SPT(t - 1)$ 
  for each  $k$  on each arc  $(i, j)$  such that  $p_{ij}(k) = t$ 
     $STATIC\text{-}REOPTIMIZATION(G, SPT(t), k_{ij})$ 

```

In the above description,  $G$  denotes the network,  $q$  the source node and  $T$  the time horizon. We use  $SSP(G, q, 0)$  to refer to an algorithm to determine the one-to-all shortest paths in  $G$  departing from  $q$  at time 0. Similarly we refer to  $STATIC\text{-}REOPTIMIZATION$ , a static shortest path reoptimization algorithm, an example of which is presented in the Appendix; this function's arguments are the network, the shortest path tree to reoptimize and the change in the network. As noted previously, we assume that arc travel times are known for a departure from the origin at  $t_0$ . After determining  $SPT(0)$ , the algorithm calculates  $SPT(t)$ , in order of increasing  $t$ , by reoptimizing  $SPT(t - 1)$  to account for all changes in the network for which their projection is equal to  $t$ .

We note that the only unknown in the above algorithm are the projections. The concept of the approximate projection, defined below, will be used to operationalize the idealized dynamic reoptimization algorithm.

**Definition 2** *We define the approximate projection of the  $k$ th change on arc  $(i, j)$ , relative to a departure from the origin at time  $t'$ , and denote it by  $p_{ij}(k, t')$ . It is equal to  $B_{ij}(k) - d_i(t')$ :*

$$p_{ij}(k, t') = B_{ij}(k) - d_i(t').$$

We note that while the projection of a change occurring at time  $B_{ij}(k)$  is defined independently of the time of the projection, an approximate projection is a “current best guess” as to whether a downstream change in the network will be encountered by the current departure time. Rather than calculate the shortest path length anew for each departure time, the approximate projection allows us to use information for earlier departure times to bound the arrival times.

By using the approximate projection for a given change, we aim to obtain increasingly tighter bounds so that ultimately  $p_{ij}(k, t_n) = p_{ij}(k)$ , assuming  $p_{ij}(k) = t_n$ . The algorithm makes a series of such approximate projections which are updated after each change to the shortest path distances. In addition, when stepping from reoptimizing  $SPT(t)$  to  $SPT(t + 1)$ , these approximate projections are updated based on the shortest path travel times from the previous tree. By updating the approximate projection the algorithm will ultimately determine the actual projection, thus processing only those changes which affect the current departure time,  $t + 1$ .

To ensure that the true projection is obtained, one must process the changes in a specific order, particularly in the case where the network is not strictly FIFO. If one were to process the changes affecting a given departure time without considering the arrival times, one would likely process changes which do not affect the current departure time. This occurs because the set of changes which affect the current departure time is determined from the shortest path tree of the previous departure time. If we do not process the changes which indicate upstream arc costs have decreased (and thus the arrival time at some downstream node  $i$  remains constant), one would inaccurately determine that the projection of some change  $p_{ij}(k) = t$ , when in fact  $p_{ij}(k) > t$ . To prevent such premature consideration of changes to the network, one must process the changes for a given departure time  $t$  in order of increasing arrival time. This is summarized in Proposition 1.

**Proposition 1** *By considering the changes affecting a departure time  $t$  in increasing order of arrival time  $a_i(t)$  and updating the approximate projections for every node in the network as the result of each change, when each change is processed its approximate projection  $p_{ij}(k, t)$  is equal to  $p_{ij}(k)$ . In this manner one determines an optimal*

*shortest path tree  $SPT(t)$  in a dynamic FIFO network.*

**Proof** The proof is similar to the proof of the correctness of Dijkstra’s algorithm; for space considerations some steps are not included. Denote the current time by  $t$ . For all arcs in  $A(q)$ ,  $a_q(t) = t$  and the approximate projection of any changes on these arcs is  $t + 0 = t = p_{ij}(t)$ . Therefore the tree is optimal with respect to this set of changes. We take as the induction hypothesis that when the set of changes emanating from a node  $i$  is processed  $d_i(t)$  is optimal and therefore  $p_{ij}(k, t) = p_{ij}(k) = t$ . Because node arrival times are FIFO, all changes which would affect any shortest path between the origin  $q$  and some destination  $r$  must have been processed prior to the changes at  $r$  because their shortest path travel times are lower. Because all such changes will have been processed,  $d_r(t)$  must then be optimal and thus for any arcs whose  $k$ th change affects departure time  $t$ ,  $p_{rj}(k, t) = p_{rj}(k) = t$ ,  $\forall(r, j) \in A$ . Thus we have proved the induction hypothesis and thereby the Proposition.  $\diamond$

## 2.4 The Dynamic Reoptimization Algorithm

In this section we provide a description of the dynamic shortest paths reoptimization algorithm in addition to a pseudocode implementation. The algorithm itself follows closely the theoretical discussion in Sections 2.2 and 2.3. Following the algorithm description we provide an example where we use the algorithm to determine the shortest path trees on a sample network.

### 2.4.1 Algorithm Description

For the following discussion we introduce the following notation. Let  $C$  be the set of all changes in the dynamic network. This set will be ordered by increasing time of change for each arc; we denote the next unprocessed change in travel time on arc  $(i, j)$  by  $C_{ij}$ . We also store a list of the projections  $p_{ij}(C_{ij})$  which we denote  $P$ . To prevent notational overburdening, we omit the redundant subscripts and refer to  $p_{ij}(C_{ij})$  as  $p(C_{ij})$  and will refer to the approximate projection in a similar manner. Let  $P(t)$  be

the set of all such changes which project onto time  $t$ :  $P(t) = \{C_{ij} : p(C_{ij}) = t\}$ .

We now describe the algorithm, the pseudocode for which is given in Fig. 2-1 and which we refer to as **DYNAMIC-REOPTIMIZATION**. The first step in the algorithm is to find the initial shortest path tree,  $SPT(0)$ . Having determined an initial shortest path tree, we must make the initial projections. Note that we discard all changes that project to a departure time earlier than our initial departure time as these have no bearing on the problem at hand.

Following this initial projection of the changes, we begin the process of reoptimization. As discussed above, the shortest path tree begins as the tree from the previous time index. As noted in Proposition 1, we must examine the changes for which  $p_{ij}(k) \leq t$  in order of increasing  $a_i(t)$ . We explain following Definitions 1 and 2 that the inequality results from discontinuity in the arrival time function. Having selected an arc with minimum arrival time, we choose the change in travel time on this arc occurring latest in time such that its projection is less than or equal to  $t$ :  $k = \arg \max_{0 \leq k' \leq K_{ij} : p_{ij}(k') \leq t} B_{ij}(k')$ . We note that for a given arc  $(i, j)$ , if  $a_i(t) > T$ ,  $\forall t < T$ , we cannot ignore the last such change  $k = \arg \max_{0 \leq k' \leq K_{ij}} B_{ij}(k')$  (we remind the reader that  $B_{ij}(k) < T \forall k$ ). This is for the simple reason that even though the arc cannot be reached before the network becomes static, all future departures will experience this final travel time.

Having selected the appropriate change in the network to process, the algorithm makes use of a subroutine for reoptimizing the shortest path tree in a static network, denoted as **STATIC-REOPTIMIZATION**. As described in Section 2.2, any such static reoptimization algorithm may be used. The procedure **STATIC-REOPTIMIZATION** is passed a copy of the current shortest path tree, the change and a copy of the network  $G$  where the arc travel times  $d_{ij}$  have been set equal to  $d_{ij}(a_i(t))$ . The procedure **UPDATE-PROJECTIONS** is used to calculate the new approximate projections given the shortest path tree determined by the reoptimizing subroutine. The pseudocode for this subroutine is shown in Fig 2-2 .

**Procedure Dynamic Reoptimization (G, C, q, T)**

```

SPT(0) ← SSP(G, q, 0)
for each arc  $(i, j) \in A$ 
  while  $C_{ij} \neq \emptyset$  AND  $p(C_{ij}, 0) < 1$ 
     $C \leftarrow C \setminus C_{ij}$ 
     $P(p(C_{ij}, 0)) \leftarrow P(p(C_{ij}, 0)) \cup C_{ij}$ 
for  $t \leftarrow 1$  to  $T$  do
   $SPT(t) = SPT(t - 1)$ 
  while  $P(t) \neq \emptyset$ 
     $k_{ij} = \arg \min_{k'_{ij} \in P(t)} a_i(t)$ 
     $P(t) \leftarrow P(t) \setminus k_{ij}$ 
     $C \leftarrow C \setminus C_{ij}$ 
    while  $(p_{ij}(k, t) < t$  AND  $C_{ij} \neq \emptyset$  AND
       $p(C_{ij}, t) \leq t)$  do
       $k_{ij} = C_{ij}$ 
       $C \leftarrow C \setminus C_{ij}$ 
  STATIC-REOPTIMIZATION(G, SPT(t),  $k_{ij}$ )
  UPDATE-PROJECTIONS(P, t, d(·))
   $P(p(C_{ij}, t)) \leftarrow P(p(C_{ij}, t)) \cup C_{ij}$ 

```

Figure 2-1: Pseudocode for the dynamic shortest paths reoptimization algorithm.

**Procedure Update-Projections (P, t, d)**

```

for each node  $i \in N$  do
  if  $d_i(t)$  has changed
    for each node  $j \in A(i)$  do
      if  $(C_{ij} \neq \emptyset$  AND  $(p_{old}(C_{ij}, t) \leq t$  OR
         $t_c - d_i(t) \leq t)$ )
         $P(p_{old}(C_{ij}, t)) \leftarrow P(p_{old}(C_{ij}, t)) \setminus C_{ij}$ 
         $t' = \max\{t, p(C_{ij}, t)\}$ 
         $P(t') \leftarrow P(t') \cup C_{ij}$ 

```

Figure 2-2: Pseudocode for the UPDATE-PROJECTIONS called by the DYNAMIC-REOPTIMIZATION algorithm.



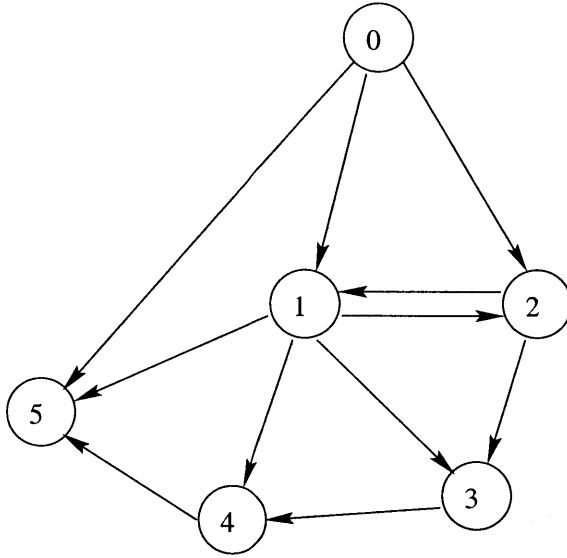


Figure 2-3: A sample network used to demonstrate the fundamentals of the dynamic shortest path reoptimization algorithm. Link travel times are given in Table 2.1.

### 2.4.2 An Example

To assist the reader in better understanding the algorithm presented in this chapter, we provide a small example. The example network is shown in Figure 2-3 accompanied by Table 2.1 which summarizes the costs upon the network. In the following paragraphs we will walk through the steps the algorithm would take in reoptimizing this network with respect to the source node of 0.

The first step is to solve the shortest paths problem to determine  $SPT(0)$ . The reader can readily verify that using Dijkstra's algorithm on the expanded time-space network produces a shortest path tree containing the set of arcs  $\{(0,2), (0,1), (1,3), (1,4), (1,5)\}$ . The distances to the nodes are, by increasing node number  $\{0, 1, 3, 3, 2, 4\}$ .

The second step is to initialize the set of changes. We note that the change on arc  $(2,3)$  occurring at time 2 does not need to be processed since  $a_2(0) = 3 > 2 = t_c$ . Therefore we have  $P(1) = \{k = 1 \text{ on arc } (0, 5)\}$  and set  $SPT(1)$  initially equal to  $SPT(0)$ . As  $\bar{c}_{0,5} = -1$  and  $(0, 5)$  is not in  $SPT(1)$ , we must first remove  $(1, 5)$  from the tree, replacing it with  $(0, 5)$ . Since there are no outgoing arcs from node 5, the

| Arc \ t | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| (0,2)   | 3 |   |   |   |   |   |
| (0,1)   | 1 |   | 2 | 1 |   |   |
| (0,5)   | 4 | 3 |   |   |   |   |
| (1,2)   | 2 |   |   |   |   |   |
| (1,3)   | 2 |   |   | 1 |   | 2 |
| (1,4)   | 1 |   |   | 3 |   |   |
| (1,5)   | 3 |   |   |   |   |   |
| (2,1)   | 1 |   |   |   |   |   |
| (2,3)   | 2 |   | 1 |   |   |   |
| (3,4)   | 1 |   |   |   |   |   |
| (4,5)   | 2 |   |   |   |   |   |

Table 2.1: Travel times on the network shown in Figure 2-3 as a function of time. In order to emphasize the changes in travel time, the table only includes entries for changed travel times; all travel times are assumed constant until the next entry in the table.

tree reoptimization is completed once we set  $d_5(1) = 3$  and  $a_5(1) = 4$ . At this point we call UPDATE-PROJECTIONS to recalculate the approximate projections. Because no arcs emanate from node 5, such a call would exert no computational effort.

An examination of Table 2.1 reveals that the changes of travel time along three arcs project into  $P(2)$  given the initial estimate of arrival times  $a_i(2)$ . Sorted by increasing arrival time, breaking ties arbitrarily, we have  $P(2) = \{k = 1 \text{ on arc } (0, 1), k = 1 \text{ on arc } (1, 3), k = 1 \text{ on arc } (1, 4)\}$ . The first change in this list occurs to an arc in the shortest path tree which experiences an increase in travel time. Because no other arc would provide an earlier arrival time to node 1 or any of the other nodes in  $SPT_1(2)$ ,  $a_1(2)$ ,  $a_3(2)$  and  $a_4(2)$  all increase by 1. This increase in arrival times, requires the recalculation of projections. An examination of Table 2.1 shows, though, that this increase in arrival time does not add any changes to  $P(2)$ . It does, however, illustrate the need for setting the “true” projection as given in Definition 1: the minimum  $t$  such that  $a_i(t) \geq t_c$ . If we lacked the inequality we would ignore the change for which  $k = 1$  on both arcs  $(1, 3)$  and  $(1, 4)$  which would clearly be incorrect. These two changes, which may be processed in any order, collectively result in the changes of  $a_3(2)$  and  $a_4(2)$  as well as  $SPT(2)$ .

When we begin reoptimization for departures from the origin at  $t$  equals 3, the last two changes in our small network are both in  $P(3)$  such that  $P(3) = \{k = 2 \text{ on arc } (0, 1), k = 2 \text{ on arc } (1, 3)\}$ . We must first process the  $k = 2$ nd change on arc  $(0, 1)$  as  $a_0(3) < a_1(3)$ . This reduction in travel time results in the decrease of  $a_1(3)$  by 1 unit. At this point UPDATE-PROJECTIONS is called, setting  $p_{1,3}(2) = 4$ . Since there are no longer any changes for which the current approximate projection is equal to the current departure time, we must increment the departure time and examine changes affecting departures at  $t = 4$ .

Finally, for  $t = 4$ ,  $SPT(4)$  would be reoptimized to account for the second and final change on arc  $(1, 3)$ . As there are no additional changes to the network, no work would be expended for  $t = 5$ , after which point the reoptimization algorithm would exit.

## 2.5 Computational Results

Our initial aim in the implementation was to examine the feasibility of the algorithm presented in the previous section. Therefore, our initial tests relied on a single implementation of the algorithm. The objectives of the computational study were to analyze the running time as functions of the following parameters:

1. The size of the network for a fixed network density;
2. The percent of links whose travel times changed from one time interval to the next;
3. The number of nodes;
4. The number of arcs; and
5. The value of the time horizon.

The computational tests are based on a C++ implementation of the algorithm described in Section 2.4 which, for ease of discussion, we shall refer to as algorithm DR. Tests were performed on a 733MHz Pentium III machine with 64MB of RAM.

All networks were randomly generated. The reported run times were obtained by averaging the running times of algorithm DR on a given network for each problem instance. For comparison, we also report the solution time of the repeated application of Dijkstra’s algorithm as modified to find shortest paths in a FIFO dynamic network, using a heap data structure. For simplicity, we will refer to this algorithm as **Repeated SSP** or simply **SSP**. The algorithm is called once per departure time, using the same test network as for the reoptimization algorithm (for a description of the algorithm, see [17, 22]). Finally, we note that for all tests, arc travel times varied between 1 and 50.

Figures 2-4 through 2-7 summarize the running time results of the implementation of the dynamic shortest paths reoptimization algorithm for networks with a constant ratio of number of arcs to number of nodes(also called density). We also show the effect of varying the percentage of link travel times that change between one time interval and the next. In each of Figures 2-4 through 2-7, we plot the number of nodes  $n$  against the running time in seconds, with  $m/n = 3$  and a time horizon  $T$  of 100. In Fig. 2-4, the increase in running time with respect to the increase in number of nodes in the test network is slightly faster than linear. We note, though, that due to the small amount of memory on the test machine, much of the slowdown on large problem sizes is likely due to the increased use of swap space. One also notes that the running times increase with the percentage of dynamic links in the network, as is expected. We note that the running times for 10% and 15% dynamic networks are nearly equivalent. We also note that for the 25% dynamic network, algorithm DR has a running time nearly equal that of **Repeated SSP**. This can be better seen in Fig. 2-5.

Fig. 2-6 shows the amount of time spent in the reoptimization subroutine, excluding time spent in initialization and other “housekeeping” tasks. It is clear that algorithm DR is faster than **Repeated SSP** except in cases when the network has a high proportion of dynamic links. This is quite promising as the implementation used in these tests was not fully optimized to take advantage of the latest developments in static shortest paths reoptimization. Again, we see that the running time increases

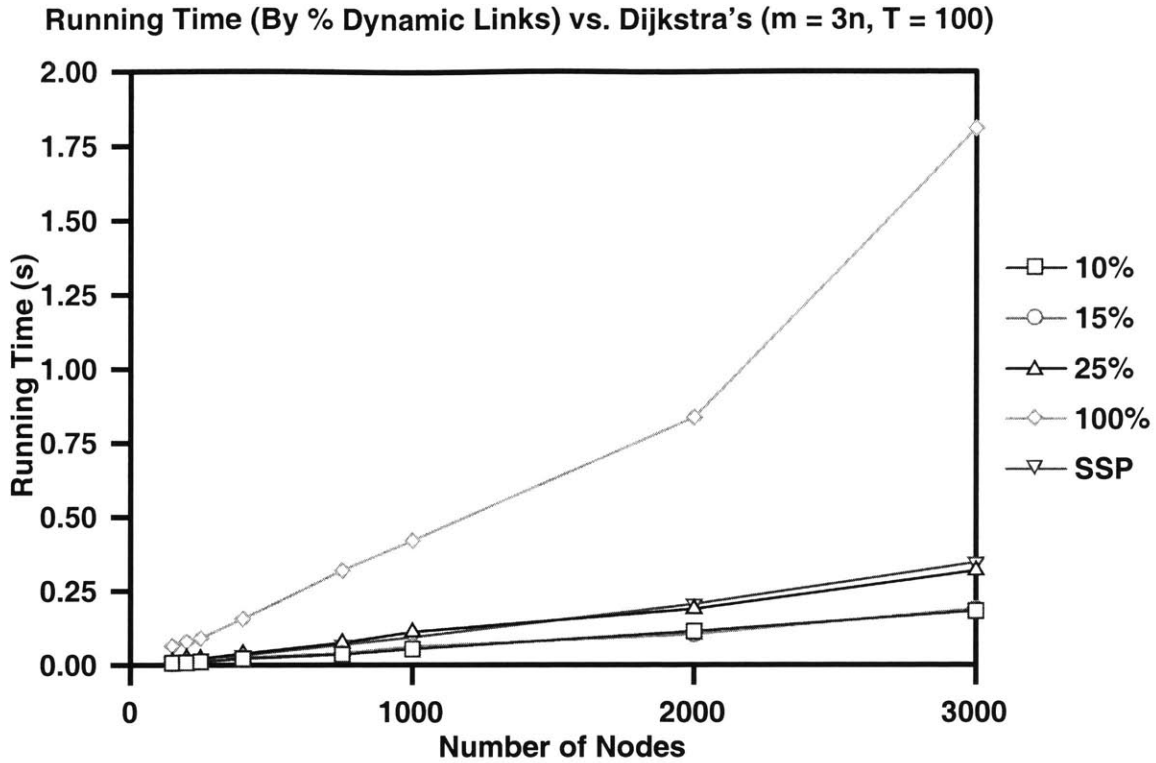


Figure 2-4: The total running time (reoptimization time plus initialization time) of algorithm DR as a function of problem size is shown for multiple percentages of dynamic links. Also shown is the running time for the modified Dijkstra's algorithm (SSP). We note the essentially linear increase in running time with problem size for both algorithms.

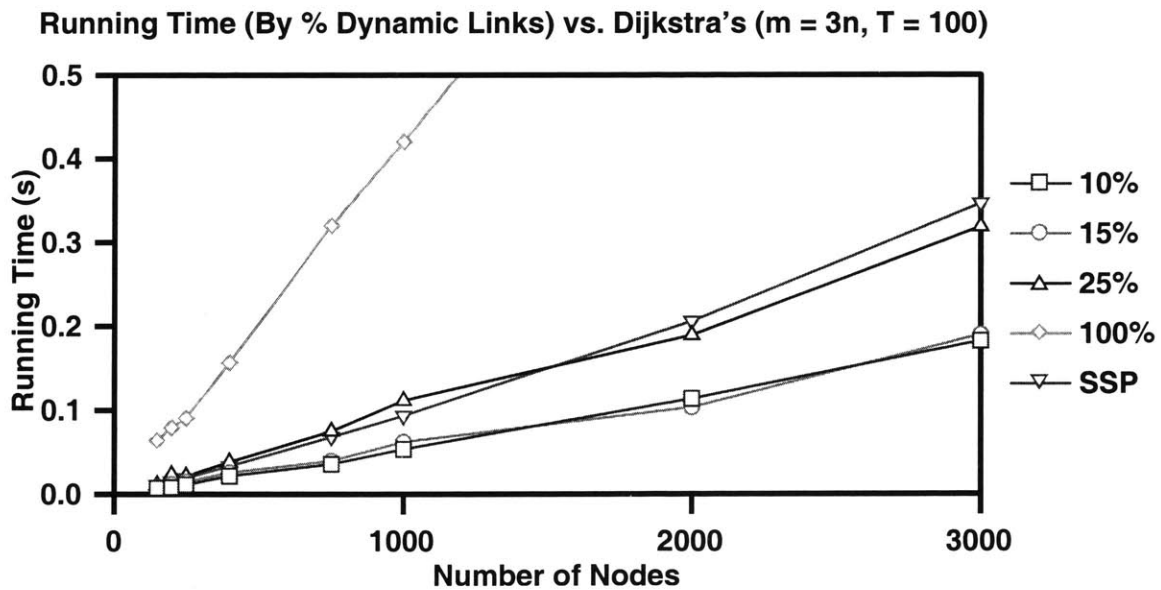


Figure 2-5: The graph depicts a close-up of the running times shown in Fig. 2-4.

**Time Spent Reoptimizing (By % Dynamic Links) vs Dijkstra's  
( $m = 3n$ ,  $T = 100$ )**

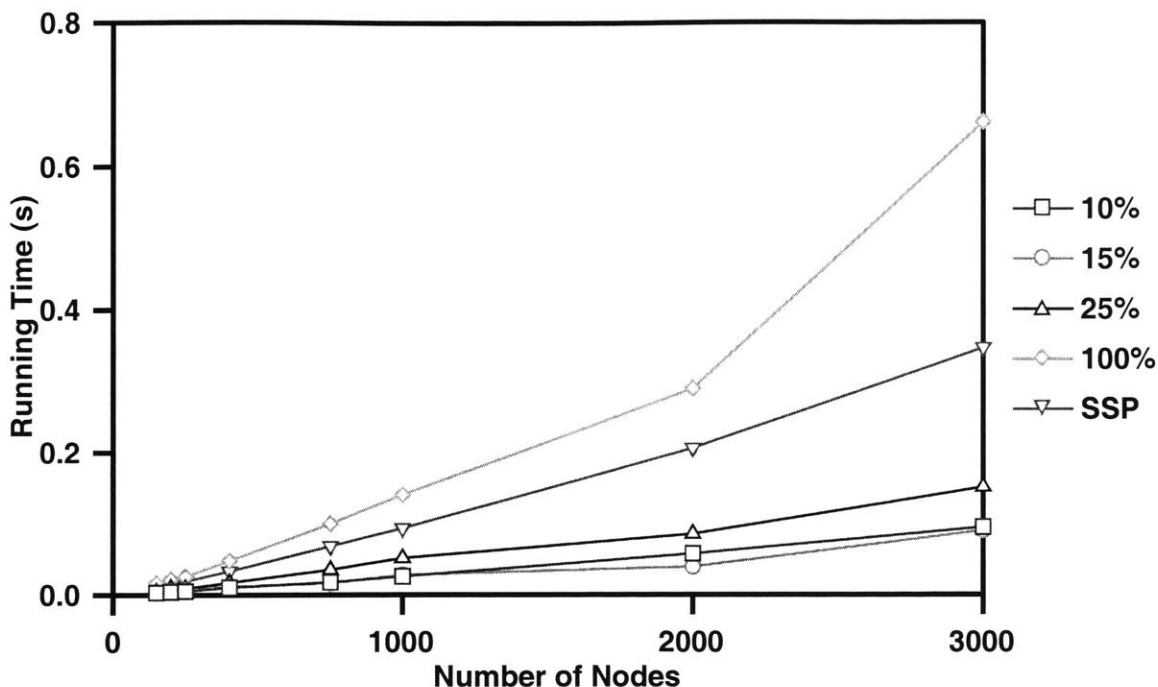


Figure 2-6: In the above graph we show the total time spent in the reoptimization routine as a function of problem size and the percentage of the links in the network which are dynamic. For comparison we show the amount of time spent running the iterative Dijkstra's algorithm. One sees that even with the unoptimized implementation used in these tests, the reoptimizer is faster than the iterative Dijkstra's algorithm for all but the most dynamic networks.

linearly with the problem size which is expected as the reoptimization time is largely dependent on the number of changes in travel times in the network.

Finally, in Fig. 2-7 the amount of time spent finding the first change to be processed for each arc, as a function of problem size, is shown; this is the time spent in the first for loop in the algorithm Dynamic Reoptimization. As expected, the increase is approximately linear in both problem size and proportion of dynamic links in the network. We note that this time is quite large in comparison to other aspects of the algorithm. For less dynamic networks, it is on the order of the entire time spent in the reoptimization subroutine. For very dynamic networks, such as the fully dynamic network shown, this time is noticeably longer than the time spent in the entire reoptimization phase. In light of these results, the sample implementation would greatly

**Time Spent Finding First Change on Each Arc  
(By % Dynamic Links) vs Dijkstra's  
( $m = 3n$ ,  $T = 100$ )**

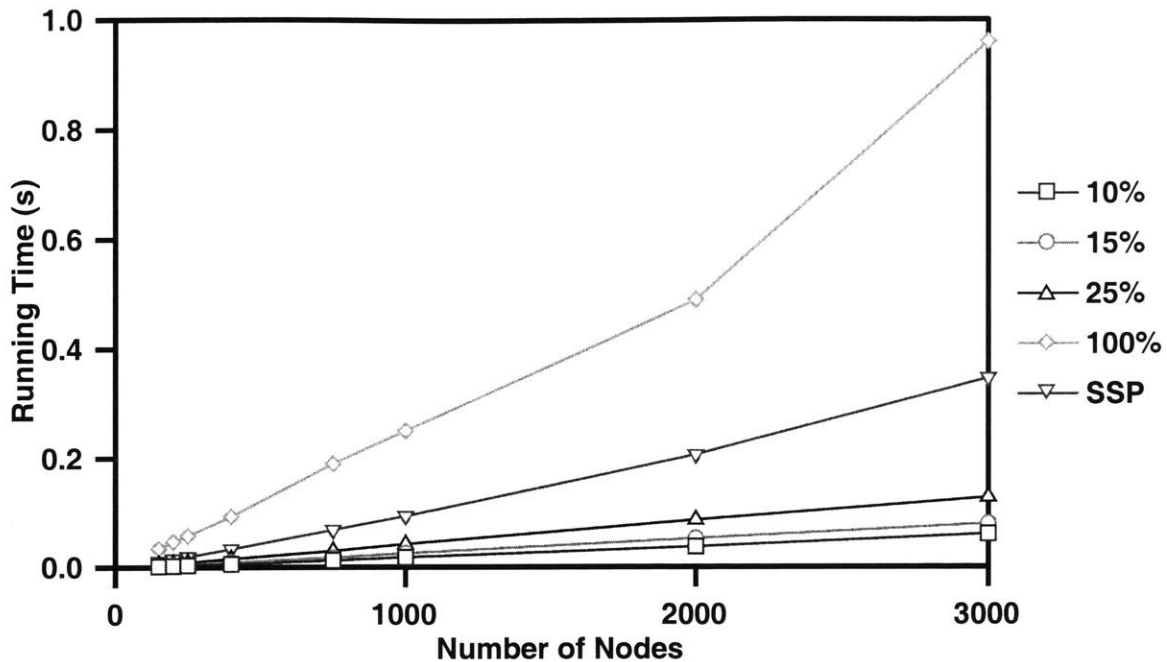


Figure 2-7: In the above graph we show the amount of time spent in finding the first change which affects  $SPT(1)$  for each arc. The growth in running time with respect to problem size is approximately linear.

benefit from ways in which the algorithm might be redesigned in order to reduce this computation time. By waiting to process the changes until after determining  $SPT(0)$ , for instance, this initialization time can be nearly eliminated in FIFO networks.

In Fig. 2-8 and 2-9, the effect of the number of nodes and arcs on computation time, respectively, is shown. In Fig. 2-8 we see that for increasing number of nodes, and a constant number of arcs constant (3000 in this case), the running time of the reoptimization algorithm actually decreases while the running time of Repeated SSP increases. This can be explained as follows. When updating the shortest path tree in the reoptimization algorithm, the running time is strongly dependent on the out degree of the end node of the arc whose travel time has changed. By holding the number of arcs constant while increasing the number of nodes, we see that the average out degree will decrease, thus decreasing the amount of work. In the case of the label-setting algorithm, however, as the number of nodes increases, so does the

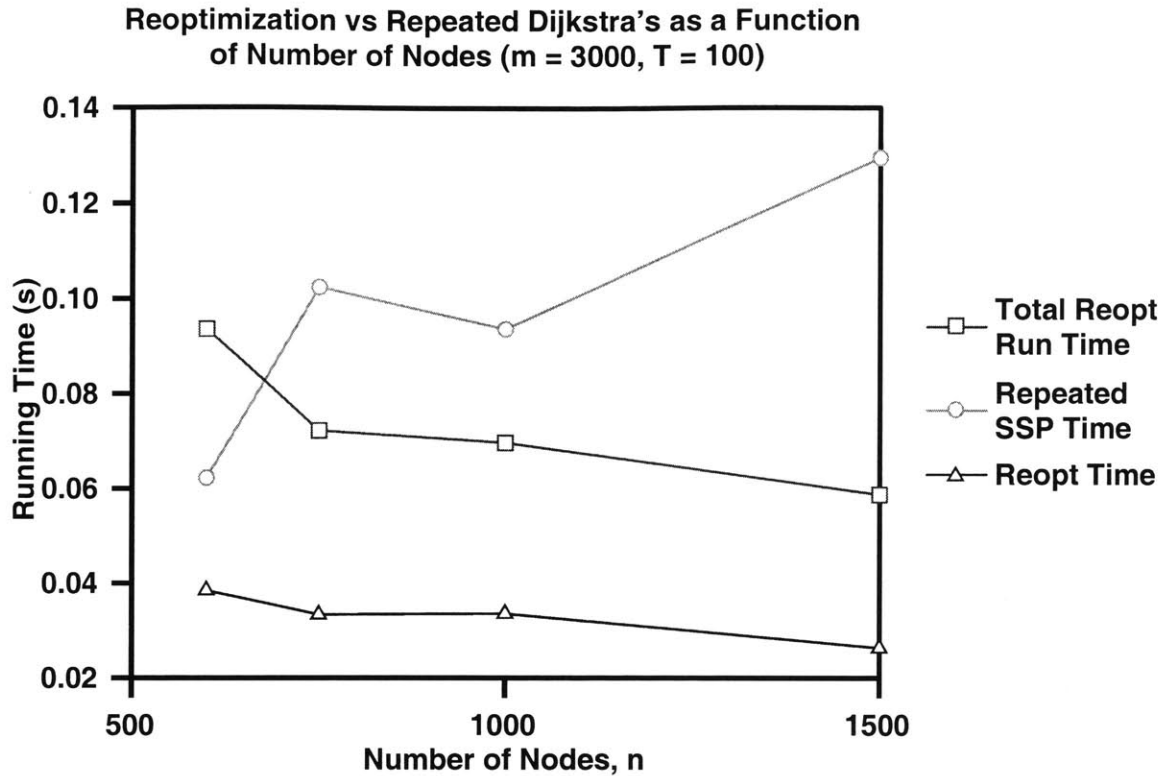


Figure 2-8: The dependence of the running time of the algorithm DR on the number of arcs in the test network is show above, using a network with 3000 arcs. In this network 15% of the links are dynamic.

amount of time required to solve the shortest path problem.

Fig. 2-9 relates the number of arcs in the network to the running time of the algorithm, for a constant number of nodes (1000 for our test). We see that, as expected, the running time of both algorithms increases with the number of arcs in the network. The sudden increase in running time at 5000 arcs is unexpected, although we expect it is due to the increased memory demand and the subsequent increase in use of the swap space. Except for this increase, the two seem to grow at approximately the same rate.

In Fig. 2-10 we see the effects of increasing the time horizon  $T$  on the running time of the algorithm. While the modified label-setting algorithm grows linearly, as expected, the increase in the overall running time is faster than linear. This increase seems to be evenly split between the reoptimizing subroutine and growth in the overhead of the data structures. This occurs because as the time horizon increases,



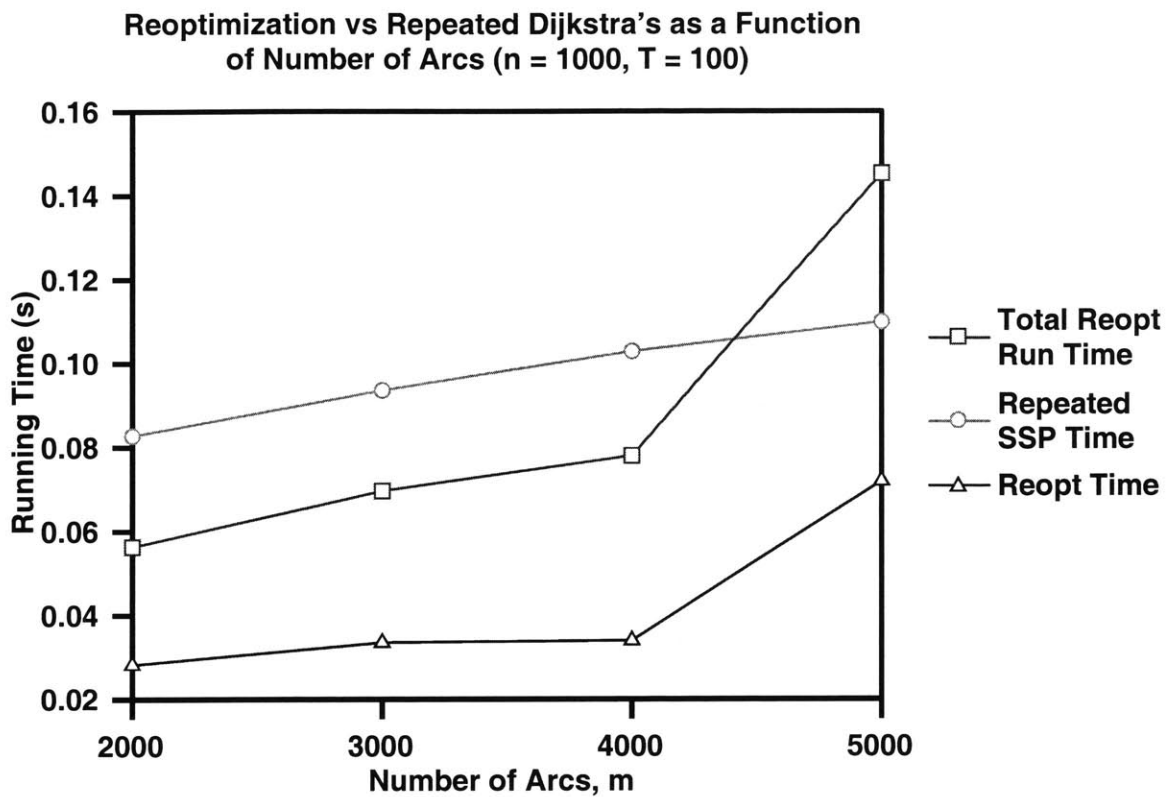


Figure 2-9: In the above graph we show the dependence of the algorithm DR on the number of arcs in the network, using a network with 1000 nodes. In this network 15% of the links are dynamic.

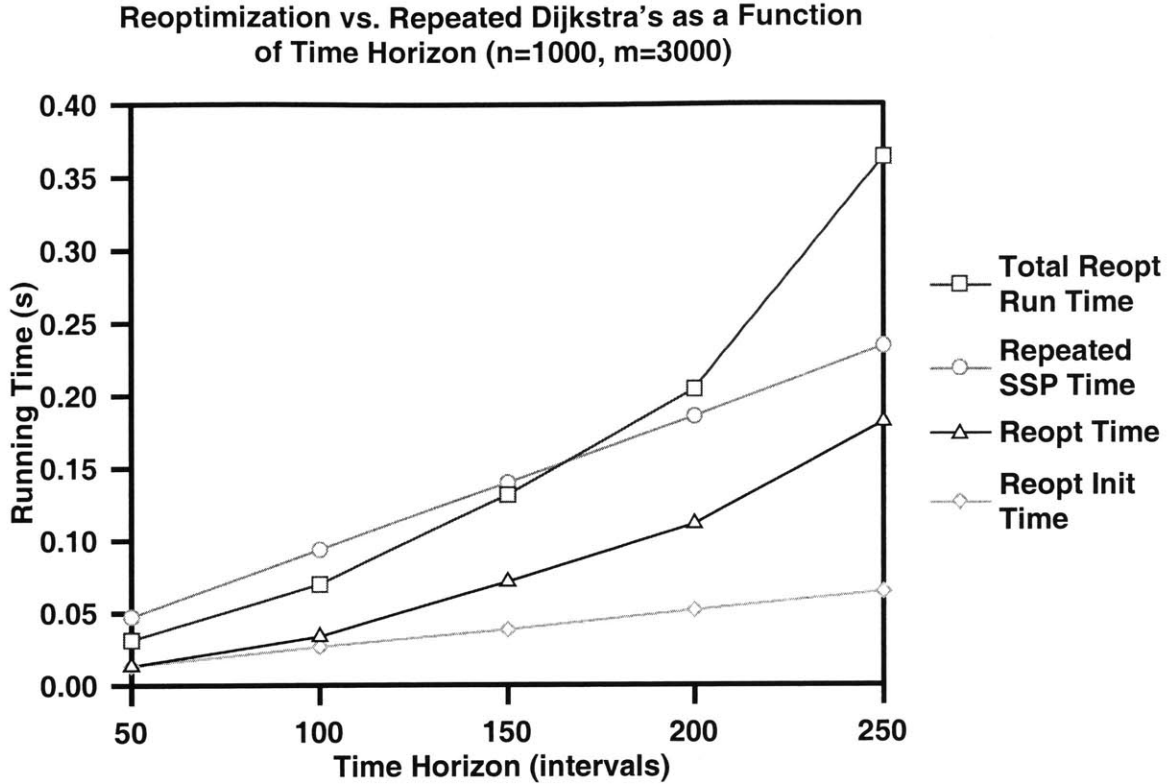


Figure 2-10: The above graph illustrates the variability in runtime of algorithm DR as a function of the time horizon, using a network with 1000 nodes and 3000 arcs. In this network 15% of the links are dynamic.

the number of nodes reached before the network becomes static (for  $t > T$ ) increases. Thus we would expect this continued rapid increase until such a time as nearly all nodes in the network were reachable before the network became static. For larger values of the time horizon, we would expect a linear increase in the running time, as the number of changes would increase linearly with the value of the time horizon.

**Comparison with Other Algorithms** Having completed a basic computational analysis of the algorithm, we now compare it with other discrete-time dynamic shortest paths algorithms to evaluate its overall performance. This set of tests aimed not to be exhaustive; rather we sought simply to measure its overall performance. The tests were performed on a Linux-based workstation with a Pentium III operating at 933 MhZ and 256 MB of RAM.

The algorithm was compared with a variant of the reoptimization algorithm pre-

sented in [7], which was described above. Specifically, the heap implementation which generates tighter bounds was used. While [7] contains algorithms for calculating shortest paths in both increasing and decreasing order of time, the latter was shown to be significantly more efficient, and thus this algorithm was used in the analysis of this section. For ease of discussion we shall refer to this algorithm as CGPS after the initials of the last names of its authors.

Algorithm DOT, as presented in [5], was used as the second comparison algorithm. While algorithm DOT solves the all-to-one dynamic shortest paths problem for all departure times, it is an optimal algorithm and thus serves as a good benchmark.

Finally, for comparison purposes, we recorded the amount of time needed to execute the algorithm Repeated SSP, as described above, on each test case. For space considerations, we again refer to it as SSP in the figures.

The algorithms' performance was tested along two axes: network size; and time horizon,  $T$ . Arc travel times ranged between 1 and 20 for this set of tests. We also note that because the algorithm CGPS operates based on knowing which arcs are static and which are dynamic, the measure of "percent dynamic" given in the following results represents the percentage of arcs whose travel times change with time; for all other arcs the travel time is constant. The reader will note that this is a slightly different interpretation of "percent dynamic" than was used above, as algorithm DR only cares about the number of changes which occur at a given time, not which arcs change.

Each of the four algorithms was test on a particular instance of a problem and the results reported are averaged over five runs. Due to various factors in the implementation, the results as reported by the implementation of CGPS were a factor higher than for the implementations of the other algorithms. As the implementation of CGPS also reported the time required to find the dynamic shortest paths using algorithm Repeated SSP, its running times were scaled by the ratio of the running times of the two implementations of Repeated SSP for the particular problem instance. This ratio ranged between 1.93 and 4.56 with an average value of 3.28. Finally, in the following tests, the total run-time for algorithm DR, as stated in Fig. 2-1 is reported. This is done for completeness, though, as was previously noted, this overstates the actual

Run Time vs Network Size ( $T=200$ , 10% Dynamic,  $m = 3n$ )

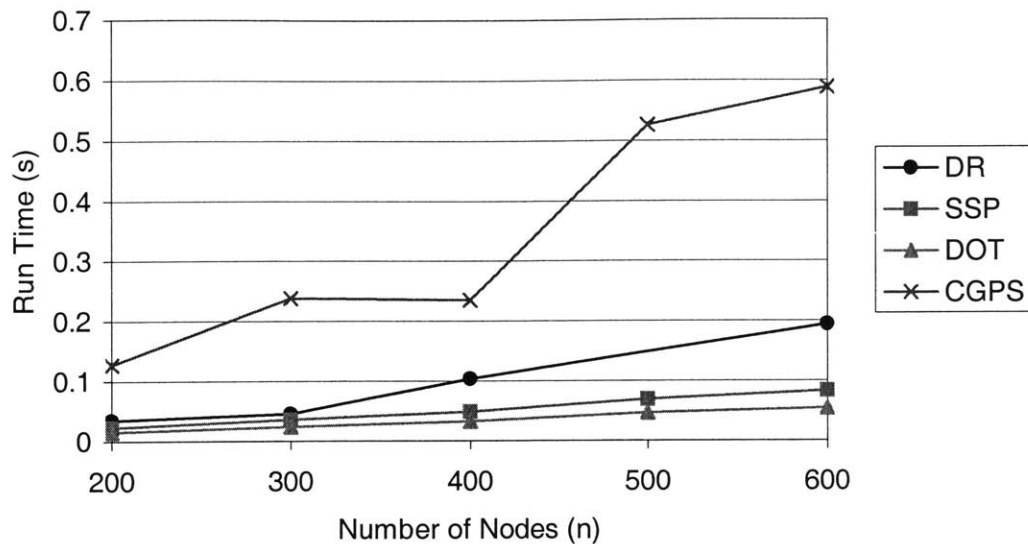


Figure 2-11: The run times of the four test algorithms are shown as a function of network size. In all cases 10 percent of the links were dynamic, the network had a time horizon  $T$  of 200, and contained three times as many links as nodes.

work needed to solve the shortest paths problem by as much as two times.

In Fig. 2-11, the running times of the four algorithms is shown as a function of network size. In each case the ratio of arcs to nodes in the network,  $m/n$ , was kept constant at three. We see there is a small but steady increase in the running time of algorithms DOT and Repeated SSP. The runtime for both of the reoptimization-based algorithms increases as well, though at a faster rate. In all cases algorithm DR took less time to complete than algorithm CGPS.

Fig. 2-12 shows the performance of the four algorithms as a function of the time horizon. We note that for low values of the time horizon  $T$ , algorithm DR performs quite well, besting even algorithm DOT. The reasons for its improved performance at low time horizons was explained above. We see that after a sharp increase for middle values of  $T$ , the rate of increase levels off and appears to be at about the same pace as algorithm Repeated SSP. We note that algorithm CGPS increases linearly with the time horizon; it is not clear whether the decrease in run time at time horizon  $T$  is

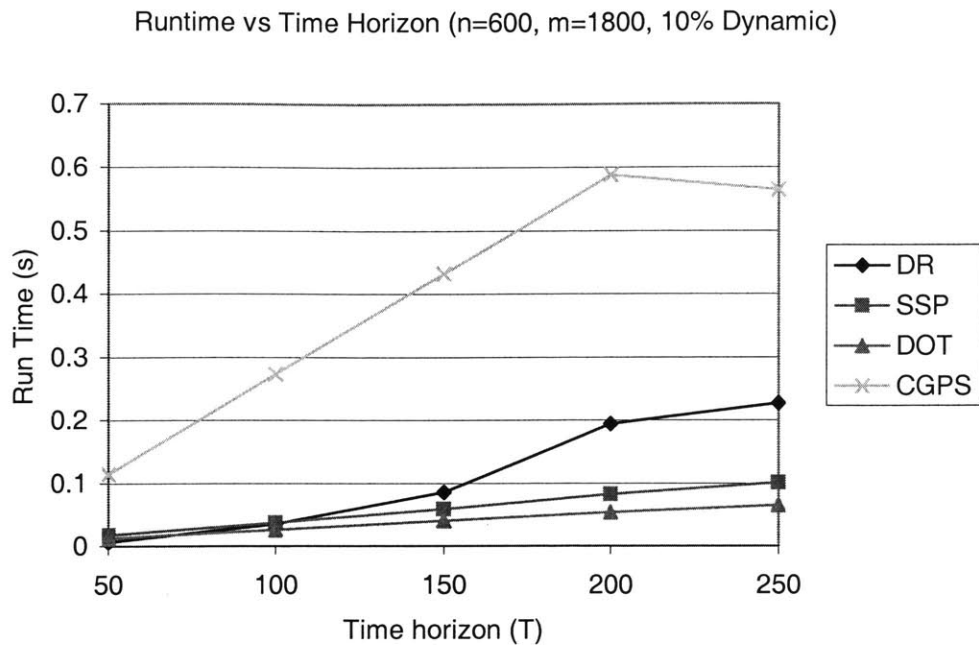


Figure 2-12: The run times of the four algorithms are shown with respect to the time horizon. In all cases the network consisted of 600 nodes and 1800 arcs, 10 percent of which were dynamic.

actually a pay-off of the algorithm or the more likely result of an artifact of the data.

## 2.6 Conclusions

In this chapter we have proposed a new method of solving the discrete time dynamic shortest paths problem for all departure times. A brief overview of static reoptimization was provided, as well as a basic static reoptimization algorithm. We have shown that, via the concept of the projection, one may view the dynamic shortest paths problem as a series of static reoptimization problems.

Given this framework, we propose a new algorithm for solving the dynamic shortest paths problem. We concluded the chapter by providing the results of a series of computational tests performed on an implementation of the algorithm. Given the known limitations of this implementation, as discussed above, the sample results are quite promising. Given this promise, we would suggest that one area of future research

would be the effect of improving the static reoptimization algorithm used. Also of interest is the potential savings from improved data-structures in the implementation.

# Chapter 3

## A Continuous Space and Time Representation of Dynamic Road Traffic Flows Consistent with Hydrodynamic Traffic Theory

The dynamic network loading problem (DNLP) is central to traffic operations and planning methods. In essence, it is the problem of finding the time-dependent link travel times, given a set of time-dependent origin-destination path demands in a network and a static link-performance model for each link. This problem is often posed within the larger context of Dynamic Traffic Assignment (DTA) [18].

Over the years of traffic research, a number of solution approaches have been developed to solve the DNLP. Of particular interest are those approaches which are consistent with the hydrodynamic traffic theory as first proposed by Lighthill and Whitham [23] and Richards [30]. Established nearly 50 years ago, this theory has become a cornerstone of analytical macroscopic traffic flow analysis. The theory provides for a description of the flow with a continuous representation of space and time.

Despite the appeal of such approaches, their number is surprisingly small. For

many years, the literature focused on the use of volume-delay functions to predict link travel times. While these models are easy to solve, they do not capture important aspects of traffic flow such as queues, spillback, and forward and backward propagations of traffic densities.

The best-known solution method consistent with the original LWR theory is the Cell-Transmission method by Daganzo [12]. In his model, Daganzo discretizes the network into a number of cells, each of which is small enough that a vehicle could traverse at most one such cell per unit of time. Using rules derived from the LWR traffic theory, Daganzo's method determines the flow which advances from one cell to the next at each clock tick. While the model is easy to understand and implement, its use of discretization produces only an approximate solution of the model. Daganzo has analyzed the accuracy of his methods and introduced a revised model to increase the accuracy of the Cell-Transmission Model [15].

At the same time Daganzo proposed improvements to his model, Cremer et al. proposed a method of discrete moving cells [10]. In contrast to Daganzo's Cell-Transmission method and other fixed-cell based models, Cremer et al.'s model suggests the use of movable cells with a fixed number of vehicles in each cell. As the vehicle speeds change in response to roadway conditions, the cell boundaries would expand or contract, resulting in a change of density within the cell. While the model provides an interesting alternative to fixed-cell models, the presentation Cremer et al.'s model is limited to a stretch of highway and its relation to an exact solution to the LWR model was not investigated.

Because of the error introduced by discretization, exact solutions to the DNLP are of particular interest. Although solution methods of continuous time and space traffic models do exist, they are complex and researchers have been reluctant to attempt to construct computer implementations of these methods. Yet if they are to be of any use to practitioners, solution approaches to the DNLP must be implemented on a computer. Farver proposed a continuous solution approach to the DNLP [18], and succeeded in its implementation, but the model does not capture queues and associated phenomena such as spillback, a property of critical importance for realistic



traffic modeling but lacking from most analytical DNLP solutions.

Another continuous time and space solution to the DNLP was proposed much earlier by Newell in a series of papers [24, 25, 26]. Newell proposes a method using cumulative flows and the network boundary conditions to generate a continuous solution to the LWR model. The method works by successively taking the minimum of the possible solutions to the cumulative flow diagrams. In [26], Newell extends the theory to multi-path flow. While the method appears to provide an exact solution to the LWR model, it has certain limitations. It is not clear that the model would extend well to a full network due to the model's reliance of prior knowledge of flows at junctions which, in a network model, would be an output of the model, not an input. Similarly, the number of iterations necessary to find the minimum solution envelope on a complex network will likely increase exponentially with relation to the increase in junctions and other discontinuities in the roadway properties. Perhaps for these reasons, Newell proposes discretizing the model when creating a computer implementation with the understanding that work-saving insights which can be made when solving the problem by hand cannot readily be made in an automated solution algorithm; insights without which a computer implementation would be overly complex and computationally intensive.

In [14], a method is sketched out for solving the DNLP in continuous space and time by hand for a homogeneous stretch of roadway. This method is similar to that for which we describe an automated process below capable of representing nonhomogeneous aspects of highways such as bottlenecks, expansions, incidents, multipath flows, and merges and diverges.

In this chapter, we propose an approach to solve the DNLP in continuous space and time which produces piecewise linear travel times. Moreover, the solution is an exact solution of the LWR hydrodynamic theory if the cumulative path flows are piecewise linear and the fundamental diagram is piecewise linear concave [16].

The sections within this chapter can be broken into three groups. The first presents the description of the model for a straight stretch of highway. In Section 3.1 we introduce the model and its underlying principles. Section 3.2 extends the method to

cases when there are changes in the roadway parameters such as bottlenecks and expansions. The second group of sections augments the material in the first to describe the model on networks with multi-path flow. Section 3.3 describes the model when the turning proportions are known and given as input to the model. This is followed in Section 3.4 by a formulation for multi-path flow. The final sections of the chapter present the algorithm and several samples. Section 3.5 provides the algorithm statement and description of the computer implementation. This is followed by a section discussing the results of several examples using this implementation.

## 3.1 Description of the Approach

Among the known methods for solving the DNLP, nearly all are described in terms of flows on the links as a function of space and time. It is generally accepted, however, that flows are actually a function of the density of the automobiles on the roadway. Macroscopic traffic theory implies that the only truly independent variable of traffic flow is the traffic density. That is, for no other traffic state-variable can a one-to-one functional relation be developed to describe the remaining state-variables. Rather than rely on flows, the model we introduce is described by the densities on the roadway as a function of space and time.

### 3.1.1 Blocks of Constant Density

On a roadway, it is reasonable to divide the roadway into segments such that throughout a segment, the density of the vehicles is approximately constant. We propose, then, to view the roadway as a series of “blocks” of constant density. For the remainder of this section, we will adopt the convention that when referring to a density block  $i$ , density block  $i + 1$  will refer to that block which is immediately downstream (that is in the direction of the flow of traffic), and density block  $i - 1$  will refer to that block which is immediately upstream of block  $i$ .

We can wholly describe the location of a block by tracking its upstream and downstream boundaries. We denote the downstream boundary of block  $i$  by  $s_i(t)$ ,

a function which gives the 1-dimensional position as a function of time; we denote the function describing the upstream boundary of block  $i$  by  $r_i(t)$ . Note that  $r_i(t) = s_{i-1}(t)$ .

We note that  $s_i(t') = r_i(t')$ , for at most two values of  $t'$ . The first time instant this equality holds represents the “birth” of the block; blocks can be created by a number of means which will be addressed later. The second time instant this equality holds occurs at the “death” of a block. This happens when the upstream and downstream boundaries cross, at which time the block ceases to exist. The effects of a block death on the propagation of the density block boundaries are discussed in Subsection 3.1.2.

While we speak of arbitrary boundary functions, the concept is perhaps best visualized using a time-space diagram. In such a diagram, the boundary functions define a closed region of time-space. This region is what we refer to as density block  $i$  as within it the traffic density is equal to a certain value  $k_i$ . Fig. 3-3 depicts an example of such a space-time diagram.

### 3.1.2 Boundary Propagation

Because we select the blocks to be of constant density, the description of their propagation through space-time is quite simple. According to the LWR model, the velocity of the propagation of the boundary between two different traffic densities is exactly equal to the slope of the line between the two points corresponding to these densities on the roadway’s density-flow relationship [12]. The density-flow relationship (known as the “fundamental diagram”) is a functional relationship between the traffic density and its corresponding flow in stationary conditions on long highways. In this thesis and related implementation, we limit ourselves to concave piece-wise linear relationships, a common assumption; for ease of discussion, we only treat triangular diagrams in the text, though.

By the above conclusion that the velocity of the boundary between two adjacent blocks is constant and the assumption of a piecewise linear fundamental diagram, all  $s_i(t)$  are piece-wise linear. Thus the problem of network loading is reduced to one of finding the intersections of these lines or boundaries. Once these blocks have been

created, each intersection, as discussed above, represents the death of a density block. If at some time  $t'$  block  $i$  were to cease to exist, we must now recalculate  $s_{i-1}(t)$  and  $s_{i+1}(t)$  for  $t > t'$ , based on the fact that blocks  $i - 1$  and  $i + 1$  are now adjacent.

Fig. 3-1 exhibits how the boundary velocity is calculated. In the figure we use  $U$  to represent the upstream state and  $D$  to represent the downstream state adjacent to the block at state  $U$ . Note that, for the most part, we use the term “state” to refer to the state of the traffic at some point, and in our model is synonymous with the traffic density; we will at times, though, use the term in a more general sense, such as “the state of the network”. In parts (a) and (b) of the figure, we show how the boundary velocity is calculated when the upstream state is in the free flow regime (that is  $k < k_c$ ) and the downstream is in the congested regime ( $k > k_c$ ). In part (a), the flow in state  $U$  is less than that in state  $D$ , so we would expect the queue represented by state  $D$  to shrink. According to the hydrodynamic theory, the speed of this value is exactly the slope of the line between the two states ( $\partial x/\partial t$ ), which is positive, indicating the boundary is, in fact, propagating downstream. Similarly in part (b), where the flow in state  $U$  exceeds that in state  $D$ , we would expect the queue to build; this is the case as the calculation shows the boundary propagating backward.

Parts (c) and (d) of Fig. 3-1 show how we handle the situation where, due to the death of a block or end of an incident, a congested state  $U$  is immediately upstream of a state with sub-critical density  $D$ . As this is an inherently unstable condition, we treat this situation by introducing an intermediate state  $M$ , which represents the maximum flow on the link. That this behavior parallels actual traffic flow is illustrated by examining an example in the extreme where the upstream state is a jam density and the downstream state is void of vehicles (such as after a red stoplight turns green or the clearing of a serious incident). In this case we would expect the vehicles at the head of the queue to leave it at the maximum flow rate allowed by the roadway, or state  $M$ . The velocity of the boundary between states  $U$  and  $M$  is given by  $\partial x_1/\partial t$ , and between states  $M$  and  $D$  by  $\partial x_2/\partial t$ . Note that the treatment of the states is

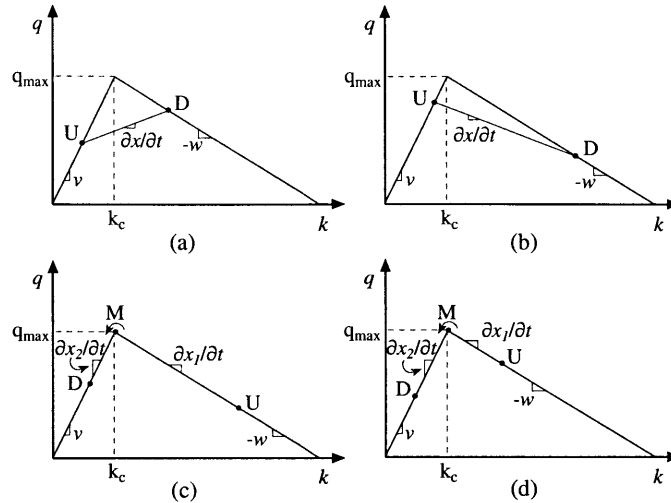


Figure 3-1: Determination of the boundary speed between upstream state  $U$  and downstream state  $D$ . Parts (a) and (b) of the figure represent the condition where the downstream density is above the critical density while the upstream is in free flow condition. Parts (c) and (d) depict the reversed situation; note the creation of the new state  $M$  between state  $U$  and state  $D$ .

independent of the relative flow rates of states  $U$  and  $D$ .<sup>1</sup>

### 3.2 Modeling Roadway Discontinuities

Given a homogeneous stretch of highway, the solution approach is quite straightforward: load the network by creating new blocks corresponding to the input densities and propagate the boundaries until all the blocks have left the network (i.e. reached their destination). Unfortunately the problem is not always so simple. Roadway capacity can decrease from one section of the highway to the next, creating a bottleneck, and can increase at expansions. In a more general model, we also want to be able to model the effect of an incident on the traffic flow. In the following discussion we

<sup>1</sup>While we do not treat it here, the theory can be extended to more general piecewise-linear, concave density-flow relationships, by creating a state at each breakpoint between states  $D$  and  $U$  [16]. In this case, the furthest upstream state created would represent the breakpoint closest to the state  $U$ . Traversing the fundamental diagram in a counter-clockwise direction, a state corresponding to each breakpoint on the diagram would be created downstream of the previous state until a block at state  $M$  had been created. Note that creating the blocks in any other order would result in the creation of blocks which would die as soon as they were created.

address all of these issues.

### 3.2.1 Bottlenecks

A bottleneck is defined as the point of the highway where the capacity decreases with respect to the road upstream of this point. While this is normally associated with a reduction in the number of lanes, it could occur due to any other number of changes in the fundamental diagram, including degradation of the roadway surface or change in roadway design (e.g. where an interstate-quality highway becomes a parkway).

The treatment of bottlenecks, as with all discontinuities, relies on the concept of flow conservation. That is, the flow entering the bottleneck must exactly equal that leaving it. If we examine Fig. 3-2, we see that two cases exist when a density block intersects a bottleneck from upstream. If the upstream block has a density less than  $k_c$ , which corresponds to a flow less than  $q_{max,bn}$ , then the capacity of the bottleneck is not exceeded. In this case (which is not shown in the diagram), the state downstream of the bottleneck would be the same as that upstream.

On the other hand, if the upstream block is at state  $A$  as the block intersects the bottleneck, we would expect a queue to be formed immediately upstream of the bottleneck. We would expect that the maximum possible flow would leave from the queue, thus we create a density block at state  $D$  immediately downstream of the bottleneck – assuming that such a block does not already exist. By conservation of flow, we must create a block immediately upstream of the bottleneck at state  $U$ , which represents the growing queue. A simple check shows that, as expected, the velocity of the  $U$ - $D$  boundary is zero, and the velocity of the boundary between the blocks at states  $A$  and  $U$  is negative, indicating the queue is growing upstream of the bottleneck.

The treatment when a block downstream of the bottleneck crosses it is even simpler. This occurs when a queue has built up downstream of the bottleneck; thus in most cases, the downstream block will be congested. By the rule of flow conservation, we would create a block immediately upstream of the bottleneck which would be congested and have the same flow rate as in the density block immediately downstream

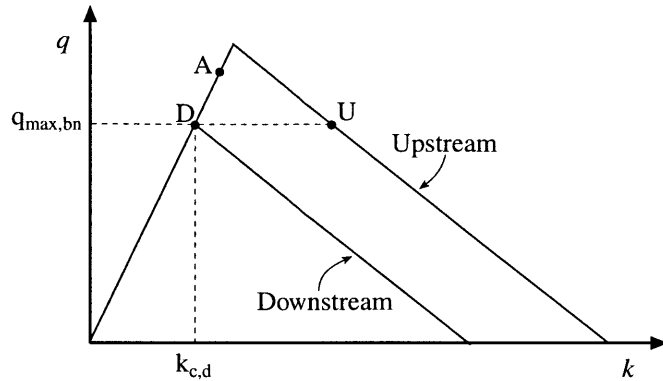


Figure 3-2: Two density flow diagrams which illustrate the treatment of a bottleneck.

of the bottleneck. If, however, the downstream block is uncongested, nothing is done; this is because its flow rate, by definition of the bottleneck, must be lower than the capacity of the bottleneck.

While we have illustrated a bottleneck where the free flow speed is the same on both sides of the bottleneck, it is relatively simple to show that the above treatment is readily extended to the more general case where this is not so. The only difference between the two treatments exists when the upstream state does not exceed the capacity of the bottleneck. In this case, one creates a density block downstream of the bottleneck with equivalent flow and in the free flow regime of the downstream flow-density relationship.

### 3.2.2 Incidents

The treatment of incidents is similar to that of bottlenecks as they are, in essence, temporary bottlenecks. When the incident occurs, we impose a flow restriction on the roadway. As with a bottleneck, if the traffic density of the block surrounding the incident is such that its flow exceeds the capacity of the incident, a queue is formed upstream of the incident. This is illustrated in Fig. 3-3.

In the case of Fig. 3-3, two states are created:  $d$  downstream of the incident and  $D$  upstream of the incident. Once the incident is cleared the condition is inherently unstable, as there is a queue upstream of free-flow traffic. As discussed above in

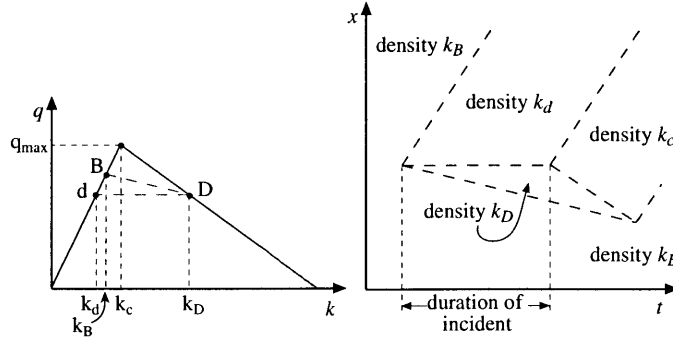


Figure 3-3: The space-time diagram representing an incident is shown on the right. On the left is the corresponding density-flow diagram.

section 3.1.2, when such an unstable condition arises, we create a new block with density  $k_c$  and flow rate  $q_{\max}$ . We can see in the figure that after some time the queue dissipates and the traffic returns to pre-incident conditions (state  $B$ ).

Note that during the course of an incident the initial queue may clear, only to have another block intersect the incident whose flow exceeds the capacity of the incident. In this case we would create new blocks as described above. If a block intersects the incident location from the downstream side, it is treated similarly to a bottleneck. If the block is congested, its flow rate must be lower than the capacity of the bottleneck (otherwise, the boundary velocity would be positive, preventing the block from intersecting the incident from the downstream side<sup>2</sup>). In this case, it passes through the incident unchanged and this queue continues to build upstream of the bottleneck.

In the case that the block approaching the incident from downstream is uncongested, the treatment is slightly more complex. As such blocks must be at the critical density – for otherwise the adjacent blocks would be in an unstable state and a block at the critical density would be created –, their flow will exceed the capacity of the incident. Moreover, they will only intersect the incident after a downstream queue has already spilled back through the incident for otherwise this boundary velocity

---

<sup>2</sup>Note that this is not strictly the case for more general, non-triangular fundamental diagrams. While we do not directly address such a case here, their treatment is readily derived from the above rules.



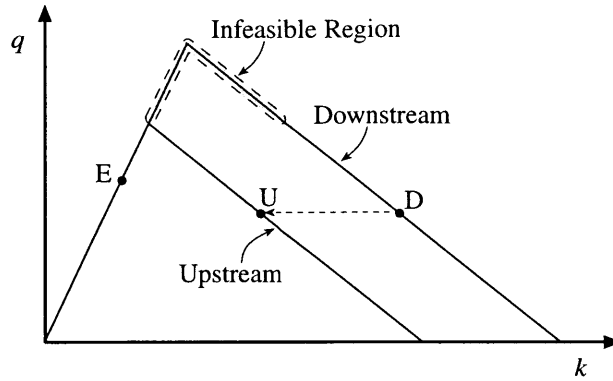


Figure 3-4: Two density flow diagrams which illustrate the treatment of an expansion in the roadway capacity.

would be positive and thus the boundary would not be traveling upstream. In this case, then, we must create a new block at the location of the incident which is uncongested and has a density equivalent to the flow rate capacity of the bottleneck. The reader will note that this is effectively the same treatment as for an expansion as is discussed below.

### 3.2.3 Expansions

The last of the discontinuities on a highway stretch we study is the expansion, or increase in roadway capacity. When a density block intersects an expansion from the upstream side, there are two conditions to consider, each of which is illustrated in Fig. 3-4. In the first case, the approaching traffic is below the critical density of the upstream roadway, for example state  $E$ . This condition persists across the expansion, by conservation of flow.

Alternately, a queue spilling back due to downstream congestion would approach the expansion from the downstream side, state  $D$ . By conservation of flow, a new state  $U$  would be created immediately upstream of the expansion and continue to propagate upstream. In Fig. 3-4, we note a region marked as “infeasible”; no density block in this region could ever pass across the expansion from the upstream side to the downstream side as otherwise its flow would be greater than the maximum flow

at the upstream side of the expansion. Therefore, with one exception, no block in this region can exist on the arc downstream of the expansion, and thus could never spill back across it. The lone exception to this rule is when an incident occurs downstream of the expansion. When it is cleared, a block will be created with density  $k_{c,\text{downstream}}$ . If the queue resulting from this incident had moved up beyond the expansion, this block with density  $k_{c,\text{downstream}}$  would intersect the expansion. In this case, similar to that described above for an incident, we create a block at the expansion which has density  $k_{c,\text{upstream}}$ ; this block is uncongested and has a flow rate equal to the maximum which can pass through the expansion. Note that by the basic rules of density block boundary propagation given above, this block will grow in both the upstream and downstream directions.

While the above description of the flow through an expansion has assumed that the free-flow speed of vehicles upstream and downstream of the expansion is the same, the treatment when this is not the case is straightforward. As with the bottleneck, the only change to the modeling of the expansion occurs when an uncongested block intersects the expansion from the upstream side. In this case, a block is created downstream which is uncongested; its density is such that its flow is equal to the flow of vehicles in the block upstream of the expansion.

### 3.3 The Network Model with Known Turn Percentages

While the model presented above, for a stretch of highway, is relatively uncomplicated, it is of little use in practical applications involving networks and flows with several origins and destinations. In the next two sections we expand the model presented above to model networks with multi-path flows. We begin by discussing the propagation of density blocks in a networks with given turn percentages. While such flow rarely exists in reality, it provides a convenient step to the development and comprehension of a model describing multi-path flow where the turning proportions

are an output of the model rather than an input.

### 3.3.1 Network Structure

While networks may have any topology and link characteristics in theory, attempting to model all possible aspects of a network unnecessarily complicates the model. Therefore, in order to simplify the calculations, we make some basic assumptions about the network structure. The assumptions presented below have essentially no effect on the generalizability of the model presented herein, and are not uncommon in network modeling (similar assumptions are made in [13] for example).

The first assumption we make is that the flow on each arc can be described using a single fundamental diagram, valid for all points along the arc. This is equivalent to saying that the roadway characteristics of an arc are homogeneous for the entirety of its length. This means that every arc, as defined in the abstract network, has uniform physical characteristics, such as number of lanes and lane width, as well as flow characteristics, including jam density and maximum flow rate. In the case where the properties of the roadway on the underlying physical network are heterogeneous, we simply represent it in the abstract network as a series of connected arcs, each with the same characteristics as the roadway segment it represents.

The second assumption we make is with regards to the topology of the network. We assume that all diverges consist of one incoming arc and two outgoing arcs. Similarly, we assume that merges consist only of two incoming arcs and one outgoing arc. Examples of acceptable and unacceptable junctions are shown in Fig. 3-5. In most highway networks, this assumption is not limiting as junctions of a higher degree are rare. In cases where such complex junctions are to be modeled, they can be divided into a series of allowable merges and/or diverges, each connected by arcs of near-zero length. As the model adopts a continuous representation of space, the only lower bound on the length of such connecting links is the resolution of the floating point arithmetic. Note that the properties of such links are considered an input to the

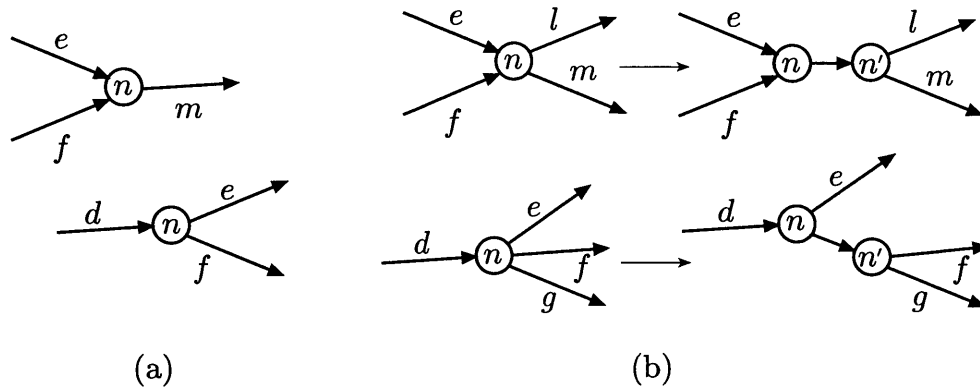


Figure 3-5: In part (a) we show an allowable merge (above) and diverge (below). In part (b) we show samples of disallowed junctions (left) together with a proposed transformation into an allowable representation (right).

model and thus their determination is not addressed here.<sup>3</sup>

Finally, as a merely operational point, we prevent a given density block from passing through a merge or diverge. While it is unlikely that the same density would exist both upstream and downstream of a junction, it is possible. We model this by creating a second block with equivalent density on the opposite side of the junction, thus effectively “disallowing” a block to pass through a junction. The reasons for this assumption will be more fully described below, in the algorithm statement, as they are purely operational.

### 3.3.2 A Note on Notation

Most of the notation used in this and the following subsections is defined below. As the terms to be defined are context sensitive, we will not attempt to describe them here. To aid the reader’s internalization of the notation, we wish to make a few explanatory notes. To begin, variables representing the actual state of the network, as currently determined by the model are given in lower case and normally subscripted with the arc to which they pertain. Thus  $k_d(x, t)$  would refer to the density on arc  $d$

<sup>3</sup>Alternately, if such a determination is not feasible, the model described below could be expanded to model junctions with a degree higher than three, but as this would complicate the presentation, it is not done so here.

at time  $t$  at position  $x$  – where  $x$  is between 0 (the origin of the arc) and the length of the arc  $d$  (its destination). To reduce the clutter, however, the position will sometime be omitted in favor of a “relational” indicator. Thus if a discussion pertains to a particular diverge node, the ‘+’ mark will be used to indicate a position immediately downstream from the node, while the ‘-’ mark will be used to indicate a position immediately upstream. Using the previous example,  $k_{d-}(t)$  would refer to the density on arc  $d$  at time  $t$  immediately upstream of the node being discuss – which is also the node at the terminus of arc  $d$ . We also use lower case to refer to other model parameters, such as the critical density  $k_c$  and turn percentages  $b(t)$ .

We distinguish values which are calculated in one way or another by upper case. While these values may become the actual state of the network, they are calculated up until such a time and are so distinguished. For example, we shall use  $Q_d$  to represent the density flow relationship on arc  $d$ ; thus by  $Q_d(k_{d-}(t))$  we refer to the value of flow associated with the specified density,  $k_{d-}(t)$ . The general exception to this rule is when we use an uppercase symbol to represent a set, such as the set of all arcs in the network  $A$ .

In our discussions of junctions, we shall commonly refer to the pair of arcs  $e$  and  $f$  either exiting a diverge or entering a merge; these bare no significance other than that they are a sequential pairing. Finally, the reader should note that commonly used graph notation is defined in Section 2.1, and these definitions are not reproduced here.

### 3.3.3 Modeling Diverges with Known Turn Percentages

Given the above limitations on network topology, we now turn our attention to the modeling of flow at a junction, beginning with the diverge. As mentioned earlier, we assume for this discussion that for a given diverge node  $i$ , the proportion of the flow from the upstream arc  $d$  turning onto arc  $e$ ,  $b_{d,e}(t)$ , is known for all  $t$ . Similarly, we assume that the percentage of vehicles turning onto the other arc  $f$ ,  $b_{d,f}(t)$ , is known and that  $b_{d,e}(t) + b_{d,f}(t) = 1$ . The nature of the turn percentages  $b(t)$  and how they are calculated is of no consequence to the current discussion.

The flow through a diverge is limited by two things: the flows on the arc entering and leaving the diverge and the roadway characteristics of these arcs. At some time  $t$ , given a flow  $q_{d^-}(t)$  on arc  $d$  immediately upstream of the diverge, we know, by definition of the turn percentages, that the desired flow onto arc  $e$  is  $b_{d,e}(t)q_{d^-}(t)$ ; similarly the flow of vehicles which would like to enter arc  $f$  is  $b_{d,f}(t)q_{d^-}(t)$ . In order to calculate the actual flow rates, however, we must account for the downstream conditions at time  $t$ . If an arc is uncongested, its maximum flow acceptance rate is equal to the maximum flow on the arc  $q_{\max}$ , as defined by its fundamental diagram. If the flow is congested immediately downstream of the diverge, however, the acceptance rate is limited to the flow at this point. For example, if arc  $e$  is congested immediately downstream of the diverge, its maximum acceptance rate is  $Q_e(k_{e^+}(t))$ , where  $Q_e$  represents the density-flow relation on arc  $e$  and  $k_{e^+}(t)$  is the (congested) density on the arc  $e$  immediately downstream of the diverge at time  $t$ .

In addition to the above considerations, we must impose an additional restriction on the flow at a diverge to maintain the specified turn percentages. If we allowed each arc to accept up to its maximum acceptance rate, the percentage of flow leaving the upstream arc for each downstream arc would be different than specified. Moreover, we note that allowing such an allocation would violate the FIFO property on arc  $d$ . This is most readily apparent in the case that one of the two outgoing arcs is completely blocked, but vehicles routed for the other arc are still allowed to progress. To maintain the proportions, then, we impose the restriction that if the desired flow onto either downstream arc is greater than its capacity, the other outgoing link is similarly restricted. Mathematically, we then define the flow rate which can pass the end of the upstream arc  $d$ ,  $X_d$ , as

$$X_d(t) = \min \left\{ q_{d^-}(t), \frac{U_e(t)}{b_{d,e}(t)}, \frac{U_f(t)}{b_{d,f}(t)} \right\},$$

where  $U_e$  and  $U_f$  are, respectively, the acceptance capacities of arcs  $e$  and  $f$  at time  $t$ . As  $U_e$  and  $U_f$  represent the upper bound on the flow which can enter each link, their calculation is straightforward. If link is congested, it can only accept flow at the same

rate as exists immediately downstream of the junction. If it is uncongested, however, it can accept flow at up to the maximum possible rate for the arc,  $q_{\max}$ . Thus  $U_e$  and  $U_f$  are simply as follows:

$$U_e(t) = \begin{cases} q_{\max,e} & \text{if } k_{e^+}(t) \leq k_{c,e} \\ Q_e(k_{e^+}(t)) & \text{otherwise;} \end{cases}$$

$$U_f(t) = \begin{cases} q_{\max,f} & \text{if } k_{f^+}(t) \leq k_{c,f} \\ Q_f(k_{f^+}(t)) & \text{otherwise.} \end{cases}$$

While the above description of a diverge is accurate when the flow on arc  $d$  is uncongested, it is incomplete if the flow is congested. If the traffic upstream of the diverge is congested, and the downstream becomes uncongested, one would expect a region of maximal flow to be created on the upstream arc  $d$ , dissipating the queue. In this case we introduce a term  $U_d$ , representing the exit-flow capacity of the arc  $d$ , and define it to be:

$$U_d(t) = \begin{cases} Q_d(k_{d^-}(t)) & \text{if } k_{d^-}(t) \leq k_{c,d} \\ q_{\max,d} & \text{otherwise.} \end{cases}$$

Incorporating this term into our previous definition of the flow leaving arc  $d$ , we obtain:

$$X_d(t) = \min \left\{ U_d(t), \frac{U_e(t)}{b_{d,e}(t)}, \frac{U_f(t)}{b_{d,f}(t)} \right\}.$$

As discussed earlier, the flows entering arcs  $e$  and  $f$ , are by definition a fraction of the total flow which leaves arc  $d$ , as dictated by the  $b(t)$ . Therefore these entrance flows are simply calculated as follows:

$$W_e(t) = b_{d,e}(t) \cdot X_d(t)$$

$$W_f(t) = b_{d,f}(t) \cdot X_d(t)$$

Once the flow through the diverge have been calculated, based on the densities of the blocks both immediately downstream and upstream of the diverge, if necessary, we create new blocks, corresponding to these flows. For example, if a density block arrives at a diverge from its upstream side, we would calculate the desired and actual flows which pass through the diverge to each of the two downstream links. If the acceptance rate of both links is sufficient to serve the desired flow, we simply create new blocks on each link, in the uncongested state, with a density corresponding to this flow. For a link  $e$ , then, we would create a density block immediately downstream of the diverge with density  $k_{e+}$ , where

$$k_{e+}(t) = Q_e^{-1}(W_e(t), \text{uncongested}),$$

and similarly for arc  $f$ ; note that  $Q_e^{-1}$  is the functional inverse of  $Q_e$ . If, however, the acceptance rate of the downstream links is lower than the desired flow rate, a block in the congested region of the density-flow diagram is created immediately upstream of the block with density

$$k_{d-}(t) = Q_e^{-1}(X_d(t), \text{congested}).$$

The densities of the blocks created downstream of the diverge are calculated as above, the only difference being whether these blocks are congested or not: if the downstream arc was congested, so too will the new block; otherwise the new block will be in the uncongested regime. As discussed above, if there is a queue upstream of the diverge and a block in the uncongested regime spills back to the diverge node, it is possible to create a block upstream of the diverge which is in the uncongested regime.

### 3.3.4 Modeling Merges

The principles behind modeling a merge are similar to those presented above for the modeling of a diverge. We introduce the concept of the merge priority, as suggested in [13]. The priority of a link is a reflection of the proportion of the downstream



capacity it is allocated. The assignment of priorities could be based on physical reasons – whether a merge is of two highways, or simply an on-ramp – or other “control” reasons, such as to simulate a signalized intersection. These priorities reflect the proportion of the total downstream flow that comes from a given upstream arc, assuming that both upstream arcs exceed their allotment of the downstream flow. As with the turn proportions, we assume that for every merge with upstream arcs  $e$  and  $f$  and downstream arc  $m$ , the priorities,  $p_{e,m}(t)$  and  $p_{f,m}(t)$  are known for all  $t$  and sum to 1.

The calculations of the flows through the merge are accomplished as follows. The desired flow rates leaving arcs  $e$  and  $f$  are given by:

$$U_e(t) = \begin{cases} Q_e(k_{e^-}(t)) & \text{if } k_{e^-}(t) \leq k_{c,e} \\ q_{\max,e} & \text{otherwise} \end{cases}$$

$$U_f(t) = \begin{cases} Q_f(k_{f^-}(t)) & \text{if } k_{f^-}(t) \leq k_{c,f} \\ q_{\max,f} & \text{otherwise,} \end{cases}$$

and the acceptance rate of the downstream arc is given by

$$U_m(t) = \begin{cases} q_{\max,m} & \text{if } k_{m^+}(t) \leq k_{c,m} \\ Q_e(k_{m^+}(t)) & \text{otherwise.} \end{cases}$$

The reader will note that these are the same formulas that regulate the sending and receiving capacities of a link at a diverge. If  $U_e + U_f \leq U_m$ , all the desired flow passes through the merge and the priorities are ignored. If, as will often be the case, the desired flow exceeds the capacity of the merge, the priorities are used to assign flow. The one exception, however, is to allow the unused portion of any “assigned” flow to be used by the other link. An example of such a situation would be an on-ramp with a stop or yield sign, thus effectively giving it a priority of 0. Without this last provision, the queue might build up indefinitely; instead we know in the real world that as many vehicles as could merge into the highway would do so, effectively

allocating the unused portion of the merge capacity to the on-ramp.

These three criteria form a simple linear program, the solution to which, as discussed in [13], is simply the middle point of the three possible values. Thus, we obtain that the exit flows from arcs  $e$  and  $f$  are given by:

$$\begin{aligned} X_e(t) &= \text{mid} \{U_e(t), p_{e,m}(t) \cdot U_m(t), U_m(t) - U_f(t)\} \\ X_f(t) &= \text{mid} \{U_f(t), p_{f,m}(t) \cdot U_m(t), U_m(t) - U_e(t)\}. \end{aligned}$$

Note that the calculated entrance flow rate on to arc  $m$  is simply the sum of these two exit flow rates:  $W_m(t) = U_e(t) + U_f(t)$ .

Once the actual flows passing through the merge have been determined, the creation of new density blocks is straight forward. If the exit flow from an upstream arc is less than the desired flow rate –  $X_e(t) < U_e(t)$  for example – the block created upstream of the merge will be congested; otherwise it is in the uncongested regime. Similarly, if the block controlling the downstream acceptance rate is congested, so too would any new block created downstream of the merge; otherwise it will be uncongested. Note that above formulas and rules are equally applicable if the downstream block backs up to the merge and results in a queue spilling back on to the upstream arcs.

### 3.4 The Network Model with Multi-Path Flow

Having presented the way in which junctions are modeled when the turn percentages are known, we now turn to the way in which we model multi-path flow. For our discussion we shall refer to a path from node  $r$  to node  $s$  by  $p_{rs}$  and the set of all such paths from  $s$  to  $r$  by  $P_{rs}$ . We shall refer to the  $i$ th path in  $P_{rs}$  by  $p_{rs}^i$ , and the  $j$ th arc on this path by  $p_{rs}^{ij}$ .

### 3.4.1 Modeling Multi-Path Flow

Unlike a discrete model where the path information can be tagged with the individual vehicle, multi-path flow in continuous space and time is somewhat more complex as we must be able to define the path flow rates as continuous function of space and time for all points in the network. We accomplish this by taking the standard discrete model of “packets” and defining it at a differential scale. We define  $\alpha_{j,rs}^i(x, t)$  to be the *proportion of the total flow* on arc  $j$  on path  $p_{rs}^i$  at point  $(x, t)$ . Thus  $\sum_{rs} \sum_{i \in P_{rs}} \alpha_{j,rs}^i(x, t) = 1$  for all points  $(x, t)$  on  $j$  and all  $j$  in  $A$ .

#### Model Input

The data needed for the model is equivalent to that provided to a discrete model. Simply, one must define the proportion of the flow at the network boundaries that is on each path, where the input flow is a function of the provided input densities. In other words, the model requires as input

$$\alpha_{j,rs}^i(0, t) \quad \forall j \in \{j : j \in p_{rs}^{i,0}\}; i \in \{i : p_{rs}^i \in P_{rs}\}; r, s \in N.$$

#### Proportions in the Network

As these proportions represent the mixture of flow for an infinitesimally small segment of the roadway, it is trivial to show that they propagate with the same speed as traffic on the roadway. That is for some arc  $j = (u, v)$ ,  $\alpha_{j,rs}^i(0, t) = \alpha_{j,rs}^i(L_j, a_{uv}(t))$  where  $L_j$  is the length of arc  $j$  and  $a_{uv}(t)$  is the arrival time at node  $v$  having departed node  $u$  at time  $t$  (and traveling along arc  $(u, v)$ ). Therefore we can “propagate” the proportions through the network using this relation, as the proportion of flow on each path at the exit of an arc is simply the same as when it entered the arc. Furthermore when the node at the end of an arc is not a merge or diverge, we know that the proportions at the beginning of the downstream arc are equal to those at the end of the upstream arc. We now address how these proportions affect the flow at junctions, and how the proportions are calculated on the downstream side of a junction.

### 3.4.2 The Diverge in a Network with Multi-Path Flow

The reader will recall that when describing the determination of flows through a diverge in the discussion above, the means of calculation of the  $b(t)$  was inconsequential to the modeling of the diverge. Because all that has changed is the manner in which the  $b(t)$  are calculated, it should be apparent that the modeling of a diverge is unchanged. That is because the  $b(t)$  are simply functions of the proportions, the values of which we have already shown how to calculate.

The only additional step, then, in calculating the flow through a diverge, is the calculation of the turning percentages from the proportions. For a diverge from arc  $d$  to arcs  $e$  and  $f$ , the turning percentages  $b_{d,e}(t)$ ,  $b_{d,f}(t)$  are simply:

$$b_{d,e}(t) = \sum_{r,s} \sum_{p_{rs}^i: e \in p_{rs}^i} \alpha_{d,rs}^i(L_d, t)$$

$$b_{d,f}(t) = \sum_{r,s} \sum_{p_{rs}^i: f \in p_{rs}^i} \alpha_{d,rs}^i(L_d, t).$$

The remainder of the calculations are identical to those described above. We note that because the flow on an arc is inherently FIFO in our model, and the modeling of the diverge is independent of the calculation of the turn percentages, the flows through the diverge are inherently FIFO. This is in contrast to discrete models where destination information is associated with individual vehicles (or packets thereof) where the vehicles are stored in buckets of finite size. In these models great care must be taken to ensure that the arc upstream of the diverge verifies the FIFO condition.

The only additional calculation which must be performed when modeling a diverge under multi-path flow is that of the proportions immediately downstream of the diverge on each of the arcs. As the proportion is simply the proportion of flow on a given path, the recalculation is simply a scaling of the proportions based on the ratio

of the flows upstream and downstream of the diverge:

$$\begin{aligned}\alpha_{e,rs}^i(0,t) &= \alpha_{d,rs}^i(L_d,t) \cdot \frac{X_d(t)}{W_e(t)} & \forall i,r,s : p_{rs}^i \in P_{rs}, r,s \in N \\ \alpha_{f,rs}^i(0,t) &= \alpha_{d,rs}^i(L_d,t) \cdot \frac{X_d(t)}{W_f(t)} & \forall i,r,s : p_{rs}^i \in P_{rs}, r,s \in N.\end{aligned}$$

Note that if there is no flow on a given path – that is  $\alpha_{d,rs}^i(L_d,t) = 0$  –, the proportion is defined as 0 on the downstream arc. By extension, if there is no flow on a downstream arc, the proportion for all paths on that arc will be 0. Similarly, the proportions for those paths which do not include an arc are defined to be 0 and as such need not be calculated.

### 3.4.3 The Merge in a Network with Multi-Path Flow

Just as the modeling of the diverge is similar in multi-path flow as for otherwise known turn percentages, so too is the modeling of the merge. The underlying principles of traffic flow the same. While we assume that the merge priorities are exogenously determined, it would have no affect on the modeling of the merge should one want to make the priorities a function of the flow. It has been proposed, for instance, that merge priorities are better modeled as functions of the total flow on each link of the merge, than a fixed priority [6]. Just as the turn percentages for the diverge were calculated above as a function of the flow, so too could the merge priorities – all without affect on the method of determining the flows through a merge.

The only additional step necessary when calculating flows at a merge under multi-path flow is the calculation of the path flow proportions downstream of the merge. As with the diverge, this is simply a rescaling of the upstream proportions:

$$\alpha_{m,rs}^i(0,t) = \alpha_{e,rs}^i(L_e,t) \cdot \frac{X_e(t)}{W_m(t)} + \alpha_{f,rs}^i(L_f,t) \cdot \frac{X_f(t)}{W_m(t)} \quad \forall i,r,s : p_{rs}^i \in P_{rs}, r,s \in N.$$

Note that while the downstream proportion is the sum of the scaled upstream proportions, one of these terms will always be 0 as a given path  $p_{rs}^i$  will include arc  $e$  or  $f$ , but not both (assuming non-cyclic paths). Also, as with the diverge, if there is no

flow on an arc and for paths which do not utilize an arc, the proportions are 0.

## 3.5 Algorithm Description

Thus far in the paper we have presented the framework and background for our new approach to solving the DNLP. We now present the algorithm by which we actually solve the problem. We first give a pseudocode description of the algorithm followed by a description of the algorithm as implemented, and two examples solved using this implementation.

### 3.5.1 Pseudocode Implementation

Prior to presenting the algorithm we summarize the assumptions we have made in the prior discussion of this chapter. We assume the link fundamental diagrams are piecewise linear and concave. We assume that the network input densities are stepwise constant. Finally, for ease of presentation, we assume in the following algorithm that the network is initially empty; this has no effect on the generalizability of the model as we only omit the routines to validate the initial densities and calculate initial boundary velocities. For simplicity, we also assume that density-flow relationship of a given link is constant along that link.

The model we present below can be viewed as an event-based simulation. Events are generated for any of the following: block deaths; block intersection with the end of an arc (i.e. bottlenecks, expansions and junctions); transient events that modify the roadway characteristics for a limited time period (i.e. incidents); and proportion events (i.e. when a change in the path proportions intersects the end of an arc). As the events are generated they are time-stamped and added to a time-sorted list  $Q$  and then processed in order of increasing time at which they occur. In order to ensure that the effects of a change in input density are properly accounted for, a reference is maintained as to the next such change (referred to as  $t_{\text{next}}$  in the description below). All events are processed up to this time, at which point new blocks are generated at the network entrances, as necessary. To allow for the evaluation to last only a given

```

Procedure CTDNLP( $t_{\text{end}}, G, z, L$ )

for each  $l \in L$ 
     $Q = Q \cup l$ 
for each  $i \in A_E$ 
     $B = B \cup \text{create\_block}(z_i(0), 0)$ 
     $Q = Q \cup \text{add\_proportion\_event}(\alpha_i(0, 0))$ 
 $t_{\text{next}} = \text{get\_next\_change}(z, \alpha)$ 
for each  $b \in B$ 
     $Q = Q \cup \text{get\_next\_event}(b, t_{\text{next}})$ 
while  $t_{\text{next}} < t_{\text{end}}$ 
    for each  $q \in Q(t_{\text{next}})$ 
         $\text{process\_event}(q, G, B)$ 
         $Q = Q \setminus q$ 
    for each  $i \in A_E$ 
         $B = B \cup \text{create\_block}(z_i(t_{\text{next}}), t_{\text{next}})$ 
         $Q = Q \cup \text{add\_proportion\_event}(\alpha_i(0, 0))$ 
     $t_{\text{next}} = \text{get\_next\_change}(z, \alpha)$ 
    for each  $b \in B$ 
         $Q = Q \cup \text{get\_next\_event}(b, t_{\text{next}})$ 
for each  $q \in Q(t_{\text{end}})$ 
     $\text{process\_event}(q, G, B)$ 
     $Q = Q \setminus q$ 

```

Figure 3-6: The pseudocode description of the DNLP solution algorithm.

duration, the loading is stopped when some maximum time  $t_{\text{end}}$  is reached or when the network becomes and remains static.

In the description given in Fig. 3-6 we utilize the following additional notation. Let  $A_E$  refer to the set of arcs that are entrances to the network. Let  $z$  refer to the time-dependent network input densities, with  $z_i(t)$  referring to the input density on arc  $i$  at time  $t$ . We refer to the set of transient events (e.g. incidents) by  $L$ ; for simplicity of presentation we assume that this list is known prior to the loading. Let the set  $B$  refer to the set of active density blocks; that is, those blocks which have not died.

In Fig. 3-6 we make use of several functions which we now describe. `create_block` takes the information about the input density and creates a new density block if

warranted. That is, if  $z_i(t)$  differs from the current density at the beginning of arc  $i$ , a new block is created with density  $z_i(t)$ . The function `get_next_change` simply examines the list of input densities and flow proportions, and returns the next time at which either input data changes. The actual calculations this function performs are dependent on the data structure used to store the input densities and proportions. The remaining functions are somewhat more complex and described below. Note that because of the way the proportion events are processed, the function need only worry about changes in input density, so long as the `add_proportion_event` function queues up all such events which occur prior to the next change in input density.

#### `get_next_event`

The function `get_next_event` returns the next event, if any, for the specified density block given  $t_{\text{next}}$ . It returns the earliest occurring event if there are several events which might occur. In order to eliminate redundancy in the algorithm, all event-checking is done with respect to a density blocks downstream boundary. We remind the reader that as a block's upstream boundary is one and the same as its upstream block's downstream boundary, this has no effect on the correctness of the model. Events are examined in the following order and only the first encountered is returned:

1. death of the downstream density block. As described earlier, this occurs when the downstream density of this block  $b$  crosses the downstream boundary of block  $b + 1$ . This event is only generated if it occurs prior to  $t_{\text{next}}$ .
2. death of this density block. This check is only performed when this function is called from the `process_event` function. Otherwise it is unnecessary as when checking for events on all density blocks, the check will be performed when calling `get_next_event` on this block's upstream block.
3. intersection of this block's downstream boundary with a temporary event.
4. intersection of this block's downstream boundary with the end of the arc it is currently on. Note that if the velocity of this boundary is negative, this will be



with the arc's origin, rather than its terminus.

Note that we keep a reference to the next scheduled event for each block  $b$ . If there is already an event scheduled for block  $b$ , the time at which it is scheduled to occur is compared with the time at which the new event would occur. If the new event would occur before the next scheduled event, this new event replaces the old event, in the event queue. If the time of the new event is later than that of the existing event, though, the new event is not added and the search for the next event proceeds to the next item in the above list of checks. If no new events are found to occur before the scheduled event, the event list remains unchanged and the function terminates. Note that as long as a density block is still active, at least one event (intersection with the end of the current arc) will always be found.

#### `process_event`

The function `process_event`, does just that: process an event. Its action depends on the type of event and the state of the network. We generally distinguish the following types of events:

1. density block death events;
2. the intersection of a density block's downstream boundary and an end of the arc it is on;
3. the beginning of a transient event, such as an accident;
4. the end of a transient event;
5. the intersection of a density block's downstream boundary with a transient event;
6. proportion events.

The actual work that is done to process each event depends on the event type. In general, however, this function is responsible for updating boundary velocities; creating new blocks; the calculation of flows upstream and downstream of junctions; the

propagation of proportion events; and the determination of new events. Note that not all events added to the queue are processed. This is the case because other, intervening events could have occurred since the event was initially added to the event queue, such as the appearance of a transient event. While the steps necessary to process the events have been described in earlier sections, these steps can be complicated and somewhat cumbersome, so we summarize below what work is performed in the processing of each type of event.

**Density Block Deaths** After checking that no interceding events have occurred – which would indicate that this event should no longer be processed –, the block in question is marked as inactive and removed from the list of active blocks  $B$ . We then check to see if the two newly adjacent blocks are an unstable condition and if so, correct the situation by creating one or more new blocks as described in Section 3.1.2. Finally we call `get_next_event` on the block whose downstream block just died as well as on any newly created blocks.

**Arc End Events** The checks which must be done when a density block intersects the end of an arc are probably the most complex of all the events. First, it must be determined whether the node at the arc's end is a merge, diverge, or simply a continuation node. In all cases, it must also be determined whether the boundary of the block  $b$  is traveling downstream or upstream as this affects the nature of the checks which must be made. If the boundary is propagating forward and encounters a continuation node we must check if this node is a bottleneck or an expansion (or neither), and treat the changes as discussed in Sections 3.2.1 and 3.2.3. If the boundary is moving upstream, the treatment is the same, although there is often less work necessary. In both cases, we may be required to inactivate a block or create a block. Finally, assuming the block in question was not inactivated, we must call `get_next_event` on this block, and any newly created blocks.

In the case of merges and diverges, we treat the blocks as described in Sections 3.4.2 and 3.4.3. As mentioned in those sections, nearly all such events will result

in the creation of two or more new density blocks adjacent to the junction. Note that any density blocks which are created upstream of a junction must be denoted as “blocked” at their downstream boundary so that the event calculating functions do not try to calculate this boundary’s velocity based on that which would otherwise be determined by the equations given in Section 3.1.2. (This should not seem surprising as in a junction there is no one-to-one pairing of upstream and downstream density blocks from which to calculate a boundary velocity.) Just as these new density blocks are created in a “blocked” state, their creation will likely “release” blocks which would had previously been blocked (that is had their downstream boundary adjacent to a junction). Once we have created new blocks on either side of the junction, we call `get_next_event` for all the newly created blocks as well as any which were previously blocked.

We note that it is quite likely that a block death will exactly coincide with a block’s intersection with a node. Therefore, to ensure the correct processing, it is important that the `process_event` function recognize when this has occurred and correctly perform all work associated with the actual event. This also underscores the need to ensure that prior to processing an event, a check is performed that the event is still valid.

**Beginning of a Transient Event** The processing of the beginning of a transient event is relatively simple. One must simply identify the density block in which the event occurred and, as described in Section 3.2.2, create new density blocks if the flow which can pass the event is lower than the flow in the current density block (as determined by  $Q_j(k_b(x, t))$ , where the incident has occurred at point  $(x, t)$  on arc  $j$ ). Finally, the function should invoke `get_next_event` on both the block which contained the event as well as the block currently upstream of this block, in addition to any newly created density blocks.

**End of a Transient Event** The end of a transient event is relatively simple as well. After removing the event from the list of active events, one simply checks if the

removal of the flow restriction has created an unstable condition and, if so, introduce a block at the critical density and maximum flow for the arc, as discussed in Section 3.2.2. In most cases, if there is a queue present, the clearing of the incident will result in an unstable condition; the exception is when a downstream queue has grown such that it has passed upstream of the incident. If a block is created, it is necessary to call `get_next_event` on both it and the block now immediately upstream of it.

**Intersection with a Transient Event** As discussed above and in Section 3.2.2, the work required to process such an event is relatively minimal. If the downstream boundary of the block which intersects the transient event is propagating downstream, we simply need check if the flow in this block exceeds the capacity of the transient event. If so, we create a congested block upstream of the event and an uncongested block, with equal flow rate, downstream of the event. If the flow is less than the event’s capacity, nothing must be done. If the boundary is traveling upstream, the work is similar, although in most cases this will also imply the death of the block which was immediately downstream of the event. Because the block which is now immediately downstream of the temporary event is in most cases congested, there is no more work other than calling `get_next_event` on the block upstream of the event. Because the congested block must be allowing less flow to pass than the event, it will continue to propagate upstream of the event and “straddle” it. The one exception is when the upstream-propagating block is at the critical density for the arc in which case its treatment is slightly more complex, but identical to that described in Section 3.2.2.

**Proportion Events** Proportion events are slightly different than the other events previously discussed. They are initially generated when the proportion of the input flow on each path changes at the network boundaries. When this discontinuity in the proportions reaches the end of an arc, such an event occurs to indicate that the  $\alpha_{l,rs}^i(0, t)$  should be updated for the downstream arc  $l$ . At a continuation, there is no special processing which must occur. In fact, it is really only necessary to generate

such events when a discontinuity in the proportions reaches a junction (that is a merge or a diverge). In the case of a merge, as was discussed above in Section 3.4.3, the proportions downstream of the merge are easily calculated and have no affect on the flow through the merge. In the case of the diverge, however, a change in the path proportions will most likely result in a change in the turn percentages  $b_{d,e}(t)$  and  $b_{d,f}(t)$ . If there is a change, the flows passing through the merge, and the creation of any density blocks, are calculated as in Section 3.4.2.

While the processing of proportion events is relatively simple, as discussed above, the calculation of the actual time such events occur is not as straightforward. These discontinuities in the proportions propagate at the same rate as a vehicle which entered the network at the same time as the discontinuity. As we cannot know the arrival time of this discontinuity at the end of the arc  $a_i(t)$  when it enters at time  $t$  – as this is the output of the network loading –, we must make a series of approximations for the arrival time at the end of the arc (or junction, if we take the less intensive approach). We accomplish this as follows: when we process a proportion event, or when the change in path proportions first enters the network, we calculate the minimum travel time to the end of the arc (or next downstream junction), and set the event to occur at this time. Note that this minimum travel time is simply the length of the arc divided by the speed at which vehicles travel in free-flow conditions. When the event is processed, we first check if the discontinuity could have actually arrived at the specified node, given the current knowledge of travel times on the network. If it has, the event is processed as described above. If it could not have reached the specified node, we calculate a new estimated arrival time. This is done simply by finding the position of the discontinuity at the current time and then calculating the minimum possible travel time from this position to the destination node. The event is subsequently added to the event queue and rechecked at this new time. It is important to note that if we calculate the time at which these events should be processed in this manner, an event will never be processed before or after it occurs, but only at the precise moment it happens.

### 3.5.2 Java™ Implementation

As it is important that a DNLP solution algorithm be implementable to be of any practical use, the algorithm described in Fig. 3-6 was implemented in Java™. Java was chosen because of its object-oriented nature and extensive application programming interface (API). This API was important in allowing the development of an animation of the algorithm. The program was written and compiled against Sun® Microsystem's Java 1.3 and JAXP extensions v1.0 and 1.1.<sup>4</sup>

The implementation is object-oriented. We utilize density block objects as the primary component of the implementation as the objects encapsulate most of the abstract notions of the blocks' interactions. As the implementation is an event-based simulation, we utilize event objects which encapsulate all the relevant information of the event. This information, together with the underlying network representation, including density-flow relations, is passed to the loading engine which is responsible for the actual creation and processing of events. The loading proceeds according to the algorithm described in Fig 3-6, and terminates when the simulation time reaches the specified end time.

As the density block objects maintain an internal copy of their space-time history, and a list of the processed events is maintained, we are able to recreate the state of the network at any point in time after the loading has finished. This time-history recorded in the density block objects is used to construct the link travel time functions once the loading has completed. As travel time information is needed during the loading to correctly process the proportion events, we found it useful to maintain a secondary time-history of the network indexed by arc, rather than density block.

#### Animation Algorithm

While not strictly part of the dynamic network loading problem, one of the main objectives in the development of this DNLP implementation was the creation of a

---

<sup>4</sup>The JAXP extension provides XML capability to the standard Java libraries. While JAXP was originally distributed as an extension to standard Java, it is now incorporated as part of default JRE and SDK as of v. 1.4.

graphical viewer to display the output of the network loading. The method used is a compromise between animation computation time and storage space. To display the network state at a given time, we simply loop over all blocks which were created during the course of the loading. If the block was active, we compute its upstream and downstream boundaries at the specified time and color the arc segment (or multiple arcs, depending on the density block's coverage) according to a user-defined coloring scheme.

### **Algorithm Efficiency**

While the runtime of this algorithm was not of primary concern and thus no tests were performed, the performance on small networks is quite good, averaging well under a second, even on older test machines. The program was tested on a Pentium III running at 733 MHz running the Linux 2.2 and 2.4 series kernels and on a Pentium II operating at 400 MHz running Windows 2000. Both machines had 256MB of RAM. Note that for demonstration purposes the application was also tested over a port-forwarded X-Windows connection and the GUI displayed correctly although the redraw was somewhat slow. As mentioned previously, the results of the loading are output in such a way as to allow for arbitrarily detailed time-dependent graphical results with this modest number of "events". Therefore one can choose to animate the results in any number of ways, or examine specific instances in time.

### **Other Uses**

A promising output of the implementation has been its use as a learning tool to better understand traffic flows, particularly involving an incident. We have had the opportunity to present the tool in a number of educational settings and have found that such simple examples have deepened the participants' understanding of the dynamics of traffic flow and the behavior of queues.

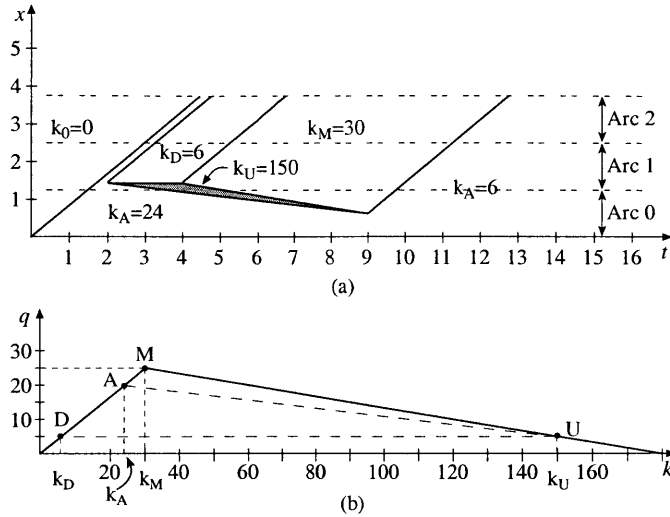


Figure 3-7: The space-time diagram resulting from the example output and the fundamental diagram for the arcs.

## 3.6 Loading Examples

To illustrate the algorithm, we describe below two examples. The first example details the workings of the algorithms on a straight stretch of highway. The second example examines the loading of a network. We note that the lack of travel time plots in these examples is not an oversight of the author, but rather due to a technical complication with the implementation which was not resolved prior to this thesis' publication.

### 3.6.1 Stretch of Highway

For this example, we consider a stretch of roadway, divided into three equal length arcs, each with a density-flow relationship as depicted in Fig. 3-7(b). This example is based on that discussed in [11]. While the units are arbitrary, the numbers used in the example roughly correspond to the characteristics of an average highway as measured in Imperial units: flow in vehicles per minute, distance in miles and density in vehicles per mile; thus the vehicles would travel at a free flow speed of 81kph (50mph).

As assumed above, we begin with an initially empty network. At time  $t = 0$ , we



set the entrance density to be 24. As the network is empty, this boundary propagates at free flow speed, as shown in Fig. 3-7(a). At time  $t = 2$ , an incident occurs 0.2 units along the second arc. This incident allows just 5 vehicles per unit time to pass through, resulting in the formation of a queue behind the accident. This queue has a density of 150, while downstream of the accident, the density is just 6. Note that the queue propagates upstream at the same rate as the slope of the line  $AU$ . At  $t = 3.68$  the queue spills back on to the first arc in the network. Finally at  $t = 4$  the incident is cleared and a block is created with critical density and maximum flow. Thus the queue begins to dissipate from the downstream end at a rate equal to the slope of the segment  $MU$ . The queue eventually dissipates at  $t = 9$ , at which point the roadway begins to return to pre-incident conditions. Finally at  $t = 12.76$  the entire network is in the steady-state condition with just one density block with density 24.

To aid in the understanding of the example, we also include the textual output from the sample implementation in Fig. 3-8. The output consists of three main types of output lines. Every time a density block is created a line beginning with `newDB` is output which includes information about the new block. When an event is added to the event queue, information about the type of event being added, the time at which it will occur is output and the block affected is output. Finally, when each event is processed, a line beginning with `Event` is output indicating the type of event and relevant information. In addition, when a density block reaches the end of the network, a comment is output; in the implementation these blocks exist in a special state: they are still active but they are no longer propagating forward.

We see that the output from the implementation is identical to that if we were to solve the problem by hand. While this example is relatively simple, the implementation has been tested on more complicated scenarios and it produces the correct result.

### 3.6.2 A Network Example

We now turn to a network example in order to illustrate the DNLN on a more complicated network based on the network used in [3]. The test network involves a 7km

```

We make initial blocks.
newDB: DensityBlock@779885 DSDB: DensityBlock@3e41ec k: 24.0
MaxTime: 21;NumTransEventBegins: 1;NumTransEventEnds: 1
Event.ARC_END: (t, A#)1.5 , 0
Added EOA event: 3.0 3 DB: DensityBlock@779885
Event.TRANSIENT_BEGIN: (t) 2.0
We enter TRANSIENT.POINT
newDB: DensityBlock@3e76c7 DSDB: DensityBlock@779885 k: 6.0
newDB: DensityBlock@682406 DSDB: DensityBlock@3e76c7 k: 150.0
newDB: DensityBlock@15126e DSDB: DensityBlock@682406 k: 24.0
Added EOA event: 3.26 3 DB: DensityBlock@3e76c7
Added EOA event: 3.68 3 DB: DensityBlock@15126e
Event.ARC_END: (t, A#)3.0 , 1
Added EOA event: 4.5 4 DB: DensityBlock@779885
Event.ARC_END: (t, A#)3.26 , 1
Added EOA event: 4.76 4 DB: DensityBlock@3e76c7
Event.ARC_END: (t, A#)3.68 , 1
Added EOA event: 14.18 14 DB: DensityBlock@15126e
Event.TRANSIENT_END: (t) 4.0
newDB: DensityBlock@6d2380 DSDB: DensityBlock@3e76c7 k: 30.0
Added EOA event: 5.26 5 DB: DensityBlock@6d2380
Added EOA event: 5.2 5 DB: DensityBlock@682406
Event.ARC_END: (t, A#)4.5 , 2
We reached end of the Network.
Event.ARC_END: (t, A#)4.76 , 2
We reached end of the Network.
Event.ARC_END: (t, A#)5.2 , 1
Added DBD Event: 9.000000000000002 DensityBlock@15126e
Event.ARC_END: (t, A#)5.26 , 1
Added EOA event: 6.76 6 DB: DensityBlock@6d2380
Event.ARC_END: (t, A#)6.76 , 2
We reached end of the Network.
Event.DB_DEATH: (x,t)0.6166666666666665 , 9.000000000000002
Added EOA event: 9.760000000000002 9 DB: DensityBlock@15126e
Event.ARC_END: (t, A#)9.760000000000002 , 0
Added EOA event: 11.260000000000002 11 DB: DensityBlock@15126e
Event.ARC_END: (t, A#)11.260000000000002 , 1
Added EOA event: 12.760000000000002 12 DB: DensityBlock@15126e
Event.ARC_END: (t, A#)12.760000000000002 , 2
We reached end of the Network.

```

Figure 3-8: Output from the example. In the lines concerning the creation of new blocks, a unique identifier of each density block is output [its address in memory]. Also note that DSDB refers to the downstream density block. In addition, x refers to the one dimensional position along the arc, t refers to the time at which the event occurs and A# refers to the arc number on which the event occurred.

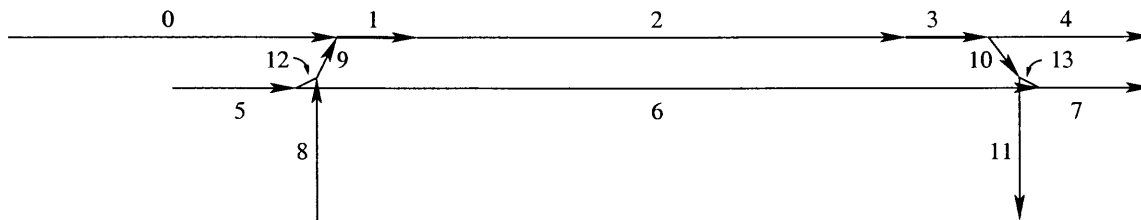


Figure 3-9: A diagram of the test network, including arc numbers. Arcs 0 through 4 represent the highway, 9 and 10 on- and off-ramps, respectively, and the remainder are arterial links.

stretch of two-lane freeway with a parallel arterial route. The network is shown in Fig. 3-9, where the two intersections have been converted into an “expanded” form via the addition of arcs 12 and 13. In order to accomate the merging and turning traffic, the freeway widens to three lanes for 0.5km as represented by arcs 1 and 3. All arterials and ramps are one lane. Freeflow speed on the highway is 110kph, 80kph on the ramps (arcs 9 and 10) and 60kph on the arterials (all other arcs). For more detailed properties of the arcs, the reader is referred to entries 2 through 4 in Table C.3.

The network was loaded from all three origin nodes. The highway had a total inflow of 3300 vph corresponding to an input density of 30 vehicles/km (or 15 veh/km/lane). This flow was divided such that 2400 vph or 72.7 percent remained on the highway; the remaining flow was split evenly between vehicles destined for arcs 7 and 11. The arterials were loaded with identical flow rates: 600vph or an input density of 10 veh/km. The destination of the flow on link 5 was evenly split between links 4 and 7, while all of the flow entering at link 8 was destined for link 4. Thirty minutes into the loading, at  $t = 0.33$ , demand on links 5 and 8 dries up and no more vehicles enter the network from these origins. The network loading is stopped after an hour of simulated time. The loading was performed on a Pentium II 400 running on Windows 2000 and took 1.37 seconds to complete.

In Fig. 3-10 through 3-13, we show the state of the network at key points in time. Because the demand from arc 1 to arc 2 is greater than the capacity of arc 2, a queue builds up at this bottleneck. This slowly propagates upstream until it is dissipated

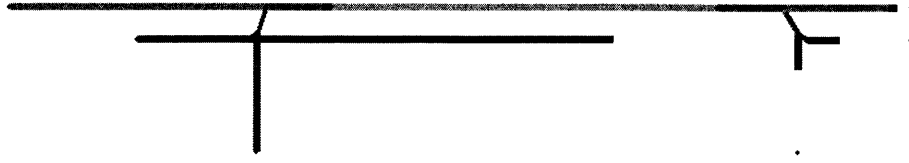


Figure 3-10: The network loading at  $t = 0.062$  hours or just under 4 minutes. The traffic has almost reached the edge of the network.

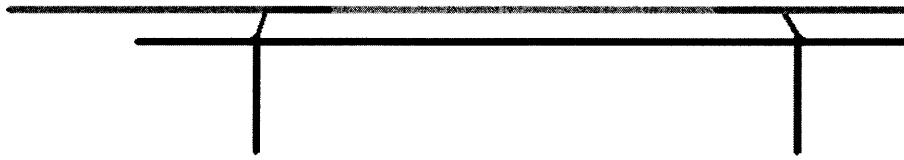


Figure 3-11: The network loading at  $t = 0.20$  hours or 12 minutes. We see that the queue due to the bottleneck has grown substantially.

by the lack of demand from the non-highway origins. Finally, the network returns to steady-state conditions after 30 mins at which point the loading terminates as no events remain in the queue.

### 3.7 Conclusions

In this chapter we have presented a new algorithm for solving the continuous time and space DNLP which generates an exact solution. Moreover, this solution is consistent

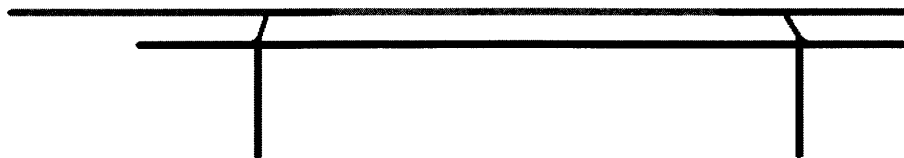


Figure 3-12: The network loading at  $t = 0.433$  hours or 26 minutes. The queue has now almost disappeared.

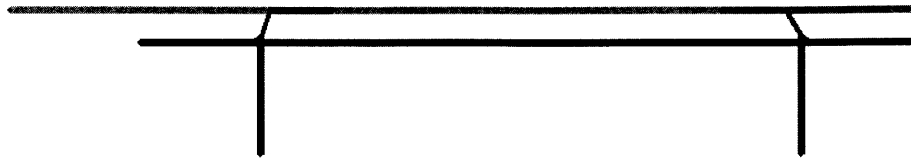


Figure 3-13: The network loading at  $t = 0.500$  hours or 30 minutes. The network has just returned to steady-state conditions.

with the original LWR hydrodynamic theory.

Such a DNLP solution method holds potential in a number of areas. By providing a continuous, exact solution it provides both detailed results and the means by which to measure other solution methods. More importantly, however, it provides the potential to better understand network effects of traffic flow and lead to the development of better traffic management tools.



# Chapter 4

## Conclusions and Future Directions of Research

Dynamic networks are important to a number of application areas in transportation, from planning to safety to real-time guidance. In each context the requirements differ. In some cases shortest paths are central. Others may focus on the Dynamic Network Loading Problem (DNLP). While still others may combine shortest paths and the DNLP to solve the Dynamic Traffic Assignment (DTA) problem.

### Summary of Contributions

We have presented a new method for solving the one-to-all dynamic shortest path problem, framing it as a series of static reoptimization problems. As background, we provided a brief history of static reoptimization, and a sample algorithm. We then showed how the dynamic shortest path problem can be viewed as a series of such problems by introducing the concept of the projection. Basic computational tests show that for networks with a small percentage of dynamic arcs, this algorithm holds much promise and with additional improvements to the implementation may prove superior to traditional methods.

In the second part of the thesis, we introduce a new framework for viewing the DNLP where density is the key state variable. Within this framework we proposed a

model in which the roadway is characterized by a set of dynamic blocks of constant density. We explain how these blocks behave at discontinuities within the network. Furthermore, we show that extension to multi-path flow is simply an overlay to the underlying model and does not materially affect the way in which the network is viewed. An algorithm using this solution method was presented and tested on a small network.

## Future Research Directions

**Dynamic Shortest Paths** The strongest point of the dynamic shortest paths algorithm is its potential savings over traditional methods of solving the one-to-all problem. In large networks where a large percentage of the data is static, the computational savings are significant. Moreover, by introducing a means to transform the dynamic problem into a series of static problems, the thesis has provided a new framework in which to view the problem which will hopefully lead to yet superior algorithms.

While we were able to test the algorithm against traditional one-to-all dynamic shortest paths algorithms, we were not able to compare it exhaustively with more recently developed algorithms, including some which utilize reoptimization in a different context. Such a comparison would allow for a better understanding of the reoptimization problem. An important analysis would be the comparison of such algorithms with the one described herein as a function of input data. As they utilize reoptimization in different ways, it is likely that each would dominate in a particular class of problems. As was also mentioned above, the static reoptimization algorithm used in the implementation was somewhat simplistic. We feel that the use of a more advanced algorithm would greatly improve the computational efficiency of the algorithm. Another potential avenue for future research would be in the underlying data structures used in the problem other than a traditional heap. Recently, for example, an “error-prone” heap has been proposed, but one in which the error rate is known and controlled [9]. It would be interesting to examine whether the runtime savings associated with such a data structure would outweigh the extra work which would



need to be done to correct errors in the order in which the changes are processed.

**Dynamic Network Loading Problem** The DNLP model presented in this thesis models continuous space and time and provides an exact solution for a network with multi-path flow. It is based on the original LWR hydrodynamic model for a link and its output is consistent with that model. All aspects of network flow are modeled, including queuing, spillback and incidents. While the model was tested on small networks, it is not clear how it will perform computationally on larger networks. In a highly dynamic network, there may be several density blocks on an arc, each continually generating events as it propagates through the network. While the work done to process some of the events is relatively minimal, many require a number of calculations and often result in the creation of one or more blocks. Without further testing we cannot know whether the rate of growth of such events and the time required to process them would become excessive for many applications.

One area of future research is the testing of the DNLP algorithm on networks in which the input data are relatively constant. Just as we suggest that highly dynamic input data might slow down our described solution method, it is likely that it would greatly benefit from data in which there are few changes to the inputs. Similarly, because the algorithm is event-driven, it is likely that it would benefit from networks with long links and fewer nodes. The events which occur on the arcs generally take less effort to process than those at the nodes. Also, the proposed algorithm currently is interrupted whenever there is a change in the input densities. It is possible that some of the penalties of highly dynamic input data could be reduced by “aligning” these data such that multiple changes occurred at the same time, thus reducing much of the overhead penalty.

Another area for future research is the full incorporation of the proposed DNLP algorithm into existing or new DTA solution methods. Such a development would allow the comparison of existing solution methods to the method described in this thesis which provides an exact solution. While this DNLP solution method could be directly incorporated into existing DTA solution methods, it holds the potential to

expand the definition of traditional analytical DTA solution approaches to something more akin to stochastic simulation models. As discussed previously, the turning proportions at a diverge are simply an overlay to the model. Thus it would be interesting to investigate a variant of this solution modeling multi-class flow; that is flow in which the vehicles have an origin and destination, but determine their path during the course of the loading based on real-time information of the network conditions. These turning decisions could either be done by a static method (i.e. always take the current shortest path) or a stochastic means. As macroscopic models traditionally do not allow for deviation from a path during the course of the loading, this could open an entirely new avenue of research.

# Appendix A

## A Static Shortest Paths Reoptimization Algorithm

As part of the dynamic reoptimization algorithm we also investigated existing algorithms for reoptimization of static shortest paths. To aid in the understanding of the dynamic shortest path reoptimization algorithm we give below a pseudocode implementation of an algorithm for reoptimizing shortest paths in a static network. While the implementation does not reflect the most recent developments in static reoptimization, we feel that it is sufficient and does not complicate the point. The procedure's arguments are the network  $G$ , the previous shortest path tree  $SPT$ , and the change in travel time  $k_{ij}$ . In the following description we use the notation  $SPT_i$  to refer to the set of nodes in the subtree of the shortest path tree rooted at node  $i$ , inclusive. In addition, we refer to the predecessor of a node  $i$  in the shortest path to node  $i$  by  $pred(i)$ .

### Static-Reoptimization (G, SPT, $k_{ij}$ )

```
 $d_{ij} \leftarrow d(B_{ij}(k))$   
 $\bar{c}_{ij} = d_{ij} + d_i - d_j$   
 $S \leftarrow SPT_j$   
 $Q \leftarrow \emptyset$   
If  $(i, j) \in SPT$   
  If  $\bar{c}_{ij} < 0$   
    For each  $e \in S$   
       $d_e = d_e + \bar{c}_{ij}$   
    While  $S \neq \emptyset$  do  
       $e \leftarrow \arg \min_{e' \in S} d_{e'}$   
       $S \leftarrow S \setminus e$   
      For each  $f \in A(e)$  do  
        If  $d_{ef} + d_e - d_f < 0$   
           $d_f \leftarrow d_{ef} + d_e$   
           $SPT \leftarrow SPT \setminus (pred(f), f) \cup (e, f)$   
           $S \leftarrow S \cup f$   
  Else If  $\bar{c}_{ij} = 0$   
    return  
  Else  
    For each  $e \in S$  do  
       $d_e = d_e + \bar{c}_{ij}$   
       $Q \leftarrow Q \cup B(e)$   
    While  $Q \neq \emptyset$  do  
       $e \leftarrow \arg \min_{e' \in Q} d_{e'}$   
       $Q \leftarrow Q \setminus e$   
      For each  $f \in A(e)$  do  
        If  $d_{ef} + d_e - d_f < 0$   
           $d_f \leftarrow d_{ef} + d_e$   
           $SPT \leftarrow SPT \setminus (pred(f), f) \cup (e, f)$   
           $Q \leftarrow Q \cup f$   
  Else  
    If  $\bar{c}_{ij} \geq 0$   
      return  
    Else  
       $SPT \leftarrow SPT \setminus (pred(j), j) \cup (i, j)$   
      For each  $e \in S$   
         $d_e = d_e + \bar{c}_{ij}$   
      While  $S \neq \emptyset$  do  
         $e \leftarrow \arg \min_{e' \in S} d_{e'}$   
         $S \leftarrow S \setminus e$   
        For each  $f \in A(e)$  do  
          If  $d_{ef} + d_e - d_f < 0$   
             $d_f \leftarrow d_{ef} + d_e$   
             $SPT \leftarrow SPT \setminus (pred(f), f) \cup (e, f)$   
             $S \leftarrow S \cup f$ 
```

# Appendix B

## Network Loading Algorithm

### Implementation Details

#### B.1 Class List

We provide the list of classes used in the implementation, as organized by package. The `ctDNLP` package consists of the classes involved in the network loading and the display of the results. The remaining packages were developed as part of a general network API. For a more detailed description of this API, including documentation and source code, the reader is referred to the research group's web site, currently located at <http://dijkstra.mit.edu>.

##### **ctDNLP Package**

This package consists of the classes used in the DNLP and those used to display the results. They are listed below alphabetically by class name.

```
ctDNLP.BadEventTimeException.java
ctDNLP.CTDensityColorPanelComponent.java
ctDNLP.CTDensityNetworkPanel.java
ctDNLP.CTNetworkDisplayPanelPicker.java
ctDNLP.CTNetworkLoader.java
ctDNLP.CTNetworkLoading.java
ctDNLP.DensityBlock.java
ctDNLP.DensityTableModel.java
```

ctDNLP.Event.java  
ctDNLP.EventGroup.java  
ctDNLP.EventTableModel.java  
ctDNLP.LoadingEngine.java  
ctDNLP.MergeDBTracker.java  
ctDNLP.NetworkViewer.java  
ctDNLP.RunLoadingDialog.java  
ctDNLP.SetDensityDialog.java  
ctDNLP.SetEventsDialog.java  
ctDNLP.UndefinedRelationshipException.java

### **edu.mit.acts.GUI Package**

This package includes classes of a general “all-purpose” nature, useful for GUI-based applications.

edu.mit.acts.GUI.AboutWindow.java  
edu.mit.acts.GUI.HelpFrame.java  
edu.mit.acts.GUI.SmartInternalFrame.java  
edu.mit.acts.GUI.TrackableWindow.java  
edu.mit.acts.GUI.WindowTracker.java  
edu.mit.acts.GUI.WindowTrackerMenu.java  
edu.mit.acts.GUI.WindowTrackingAdapter.java  
edu.mit.acts.GUI.WindowTrackingListener.java

### **edu.mit.acts.io Package**

This package includes some classes used to redirect program I/O.

edu.mit.acts.io.BucketWriter.java  
edu.mit.acts.io.PrintStreamWriter.java  
edu.mit.acts.io.TextAreaWriter.java

### **edu.mit.acts.net Package**

This package includes the core network functionality such as a Node, Arc, Path and Network.

edu.mit.acts.net.Arc.java  
edu.mit.acts.net.ConstantMergePriority.java  
edu.mit.acts.net.ConstantTravelTimeFunction.java  
edu.mit.acts.net.ContinuousTravelTimeFunction.java  
edu.mit.acts.net.DensityFlowRelation.java  
edu.mit.acts.net.Diverge.java  
edu.mit.acts.net.DivergeProportion.java  
edu.mit.acts.net.Junction.java  
edu.mit.acts.net.Merge.java

```
edu.mit.acts.net.MergePriority.java
edu.mit.acts.net.Network.java
edu.mit.acts.net.NetworkLoading.java
edu.mit.acts.net.NetworkParameters.java
edu.mit.acts.net.Node.java
edu.mit.acts.net.Path.java
edu.mit.acts.net.TravelTimeFunction.java
```

### **edu.mit.acts.netGUI Package**

This package consists of classes used for the creation of network-based GUI applications. It includes basic functionality for displaying a network and its loading as well. Note that many of the items listed below are interfaces, not classes.

```
edu.mit.acts.netGUI.AbstractNetworkLoader.java
edu.mit.acts.netGUI.ColorPanel.java
edu.mit.acts.netGUI.DrawingPrefs.java
edu.mit.acts.netGUI.FileReaderDirector.java
edu.mit.acts.netGUI.FileWriterDirector.java
edu.mit.acts.netGUI.InternalNetworkDisplayFrame.java
edu.mit.acts.netGUI.NetLoadingListener.java
edu.mit.acts.netGUI.NetworkDisplayPanel.java
edu.mit.acts.netGUI.NetworkDisplayPanelPicker.java
edu.mit.acts.netGUI.NetworkFileReader.java
edu.mit.acts.netGUI.NetworkFileWriter.java
edu.mit.acts.netGUI.NetworkLoader.java
edu.mit.acts.netGUI.NetworkMonitor.java
edu.mit.acts.netGUI.NewWindowDialog.java
edu.mit.acts.netGUI.SaveNetworkDialog.java
edu.mit.acts.netGUI.SetDisplayParamsDialog.java
```

### **edu.mit.acts.netGUI.ColorPanels Package**

Default coloring models for network and network loadings.

```
edu.mit.acts.netGUI.ColorPanels.ArcNumberColorPanelComponent.java
edu.mit.acts.netGUI.ColorPanels.ColorPanelComponent.java
edu.mit.acts.netGUI.ColorPanels.FlowColorPanelComponent.java
```

### **edu.mit.acts.netGUI.FileFilters Package**

Classes used to filter files based on the underlying data format.

```
edu.mit.acts.netGUI.FileFilters.NFFFFileFilter.java
edu.mit.acts.netGUI.FileFilters.XMLFileFilter.java
edu.mit.acts.netGUI.FileFilters.YiyiFileFilter.java
```

## **edu.mit.acts.netGUI.NetworkDisplayPanels Package**

Default network display panels.

```
edu.mit.acts.netGUI.NetworkDisplayPanels.ArcNumberNetworkPanel.java
edu.mit.acts.netGUI.NetworkDisplayPanels.BareNetworkPanel.java
```

## **edu.mit.acts.netGUI.NetworkFileReaders Package**

Classes used to parse a file describing a network and its attributes.

```
edu.mit.acts.netGUI.NetworkFileReaders.NFFNetworkFileReader.java
edu.mit.acts.netGUI.NetworkFileReaders.XMLNetworkFileReader.java
edu.mit.acts.netGUI.NetworkFileReaders.YiyiNetworkFileReader.java
```

## **edu.mit.acts.netGUI.NetworkFileWriters Package**

Classes used to write out a network representation and the results of a network loading to a file.

```
edu.mit.acts.netGUI.NetworkFileWriters.XMLNetworkFileWriter.java
```

## **edu.mit.acts.util Package**

A package consisting of various utility classes, most of which are data structures. Many of these lie at the heart of the ability to model the DNLP in continuous space and time.

```
edu.mit.acts.util.KeyedObject.java
edu.mit.acts.util.LinearFunction.java
edu.mit.acts.util.Math2.java
edu.mit.acts.util.NoRemoveIterator.java
edu.mit.acts.util.NoRemoveListIterator.java
edu.mit.acts.util.PolynomialFunction.java
edu.mit.acts.util.RangedArrayList.java
edu.mit.acts.util.StepFunctionLinear.java
edu.mit.acts.util.TimeHistoryList.java
```

## **edu.mit.acts.xml Package**

A package consisting of helper classes used for reading and writing objects to XML.

```
edu.mit.acts.xml.ObjectCreationListener.java
edu.mit.acts.xml.XMLCompatible.java
edu.mit.acts.xml.XMLFactory.java
edu.mit.acts.xml.XMLObjectCreationWrapper.java
```



## B.2 Class Hierarchy

In this subsection we give the class hierarchy, with respect to the Java API and the classes which were developed as part of this implementation. Note that we only give the hierarchies for the three primary components of the implementation, the ctDNLP package the edu.mit.acts.net package and edu.mit.acts.util package. Also, several layers of nesting are omitted for the GUI classes; this is indicated by ∴.

### ctDNLP Package

#### *Class Hierarchy*

- class java.lang.Object
  - class edu.mit.acts.netGUI.AbstractNetworkLoader (implements edu.mit.acts.netGUI.NetworkLoader)
    - class ctDNLP.CTNetworkLoader (implements java.lang.Runnable)
  - class javax.swing.table.AbstractTableModel (implements java.io.Serializable, javax.swing.table.TableModel)
    - class ctDNLP.DensityTableModel
    - class ctDNLP.EventTableModel
  - class java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - ∴
    - class javax.swing.JPanel (implements javax.accessibility.Accessible)
      - \* class edu.mit.acts.netGUI.ColorPanels.ColorPanelComponent
        - class ctDNLP.CTDensityColorPanelComponent
      - \* class edu.mit.acts.netGUI.NetworkDisplayPanel (implements javax.swing.Scrollable)
        - class ctDNLP.CTDensityNetworkPanel
      - \* class ctDNLP.NetworkViewer.AboutPanel
    - class javax.swing.JDialog (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
      - \* class ctDNLP.RunLoadingDialog
      - \* class ctDNLP.SetDensityDialog
      - \* class ctDNLP.SetEventsDialog

- class ctDNLP.DensityBlock
- class ctDNLP.Event (implements java.lang.Comparable)
- class ctDNLP.Event.Transient
- class ctDNLP.EventGroup
- class ctDNLP.EventGroup.EventGroupItem
- class ctDNLP.EventGroup.TransientEventGroup
- class ctDNLP.EventGroup.TransientEventGroup.EventItem
- class ctDNLP.LoadingEngine (implements edu.mit.acts.net.DivergeProportion)
- class ctDNLP.MergeDBTracker
- class edu.mit.acts.netGUI.NetworkDisplayPanelPicker
  - class ctDNLP.CTNetworkDisplayPanelPicker
- class edu.mit.acts.net.NetworkLoading (implements edu.mit.acts.xml.XMLCompatible)
  - class ctDNLP.CTNetworkLoading
- class ctDNLP.NetworkViewer
- class java.lang.Throwable (implements java.io.Serializable)
  - class java.lang.Exception
    - \* class ctDNLP.BadEventTimeException
    - \* class ctDNLP.UndefinedRelationshipException

## **edu.mit.acts.net Package**

### *Class Hierarchy*

- class java.lang.Object
  - class edu.mit.acts.net.Arc (implements edu.mit.acts.xml.XMLCompatible)
  - class edu.mit.acts.net.ConstantMergePriority (implements edu.mit.acts.net.MergePriority)
  - class edu.mit.acts.net.DensityFlowRelation (implements edu.mit.acts.xml.XMLCompatible)
  - class edu.mit.acts.net.Junction
    - \* class edu.mit.acts.net.Diverge
    - \* class edu.mit.acts.net.Merge

- class edu.mit.acts.net.Network (implements edu.mit.acts.xml.XMLCompatible)
- class edu.mit.acts.net.Network.ArcIterator (implements java.util.Iterator)
- class edu.mit.acts.net.Network.NodeIterator (implements java.util.Iterator)
- class edu.mit.acts.net.NetworkLoading (implements edu.mit.acts.xml.XMLCompatible)
- class edu.mit.acts.net.NetworkParameters (implements edu.mit.acts.xml.XMLCompatible)
- class edu.mit.acts.net.Node (implements edu.mit.acts.xml.XMLCompatible)
- class edu.mit.acts.net.Path (implements java.lang.Cloneable, edu.mit.acts.xml.XMLCompatible)
- class edu.mit.acts.net.TravelTimeFunction (implements edu.mit.acts.xml.XMLCompatible)
  - \* class edu.mit.acts.net.ConstantTravelTimeFunction
  - \* class edu.mit.acts.net.ContinuousTravelTimeFunction

### *Interface Hierarchy*

- interface edu.mit.acts.net.DivergeProportion
- interface edu.mit.acts.net.MergePriority

## **edu.mit.acts.util Package**

### *Class Hierarchy*

- class java.lang.Object
  - class java.util.AbstractCollection (implements java.util.Collection)
    - \* class java.util.ArrayList (implements java.util.List, java.lang.Cloneable, java.io.Serializable)
      - o class edu.mit.acts.util.RangedArrayList
  - class edu.mit.acts.util.KeyedObject (implements java.lang.Comparable)
    - \* class edu.mit.acts.util.KeyedObject.Double
    - \* class edu.mit.acts.util.KeyedObject.Integer
  - class edu.mit.acts.util.Math2
  - class edu.mit.acts.util.NoRemoveIterator (implements java.util.Iterator)
    - \* class edu.mit.acts.util.NoRemoveListIterator (implements java.util.ListIterator)

- class edu.mit.acts.util.PolynomialFunction (implements java.lang.Cloneable)
  - \* class edu.mit.acts.util.LinearFunction
- class edu.mit.acts.util.StepFunctionLinear (implements java.lang.Cloneable)
- class edu.mit.acts.util.TimeHistoryList

# Appendix C

## Examples Tested with DNLP

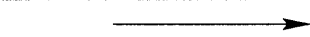
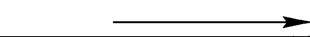
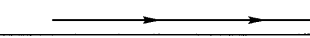
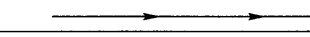
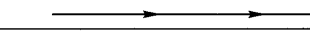
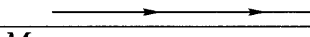
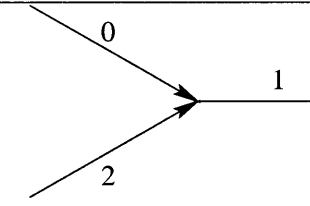
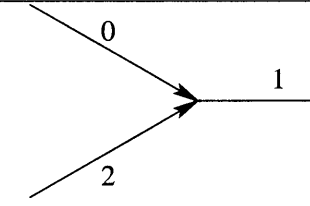
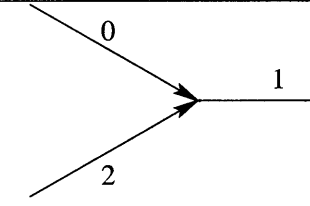
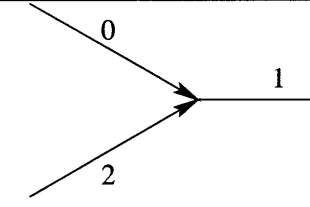
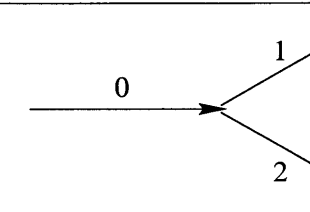
### Solution Implementation

The process of testing and debugging the DNLP implementation (described in Chapter 3) was long and involved the use of many small networks, with contrived conditions. While these test cases are too numerous to warrant insertion in the algorithm description, they may be of interest to future researchers or implementors.

#### C.1 Example Networks

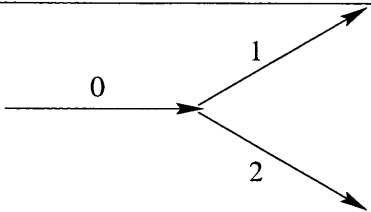
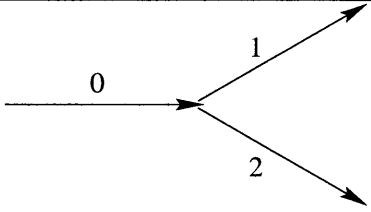
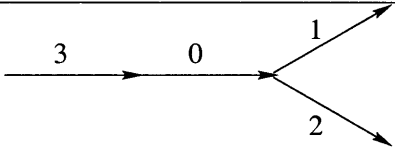
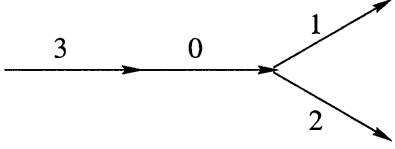
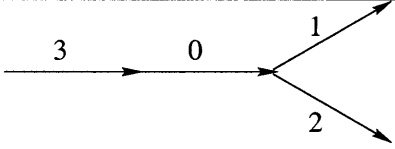
Table C.1 summarizes each test, providing information on the network shape, and any temporary events which occurred. As most of the arcs used in the tests were similar, if not identical, we summarize the arc types used in Table C.2. The only exception is for the test network used in the example; the characteristics of these arcs are described in the text, in Section 3.6; the arc types of this network is also denoted by a \* in Table C.1. The summary of the arc types is followed by a summary of the density-flow relations used in the examples in Table C.3. While we only describe triangular and trapezoidal fundamental diagrams, the implementation was designed for general convex, piecewise linear diagrams. Finally, the types of events used are summarized in Table C.4. As mentioned in Section 3.6, the values given below are largely based on those appearing in other works, including [11].

Table C.1: A summary of the examples used in testing the DNLP implementation.

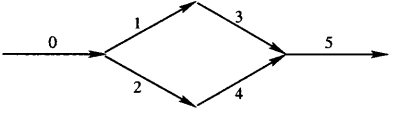
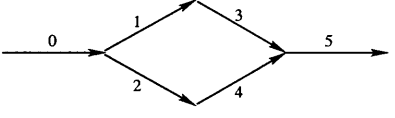
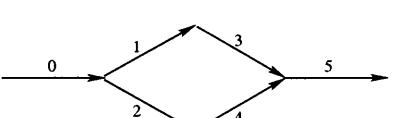
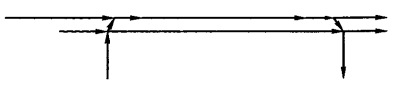
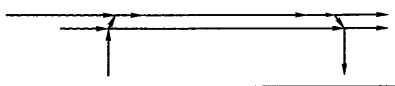
| Ex. Num                   | Graphic   | Arc Types | Entrance Densities | Event 1 | Event 2 |
|---------------------------|---|-----------|--------------------|---------|---------|
| <i>Stretch of Highway</i> |   |           |                    |         |         |
| 1                         |    | 1         | 24                 |         |         |
| 2                         |    | 1         | 24                 | 1       |         |
| 3                         |    | 1         | 24                 |         |         |
| 4                         |    | 1         | 24                 | 2       |         |
| 5                         |    | 1         | 24                 | 3       |         |
| 6                         |    | 1         | 24                 | 6       |         |
| <i>Basic Merges</i>       |   |           |                    |         |         |
| 7                         |    | 1         | 10,10              |         |         |
| 8                         |   | 1         | 10,10              | 1       |         |
| 9                         |  | 1         | 10,10              | 2       |         |
| 10                        |  | 1         | 10,10              | 1       | 5       |
| <i>Basic Diverges</i>     |   |           |                    |         |         |
| 11                        |  | 1         | 20                 |         |         |

*continued on next page*

continued from previous page

| Ex. Num | Graphic   | Arc Types | Entrance Densities     | Event 1 | Event 2 |
|---------|---|-----------|------------------------|---------|---------|
| 12      |    | 1         | 20                     | 1       |         |
| 13      |    | 1         | 20                     | 2       |         |
| 14      |    | 1         | 20                     |         |         |
| 15      |   | 1         | 20                     | 4       |         |
| 16      |  | 1         | [0,5):20;<br>[5,20):15 |         |         |

Network Examples

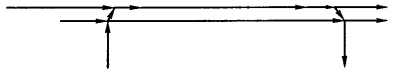
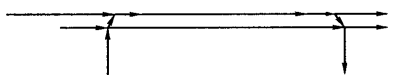
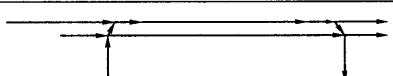
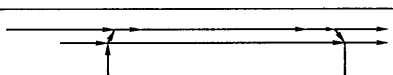
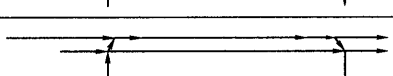
|    |   |   |   |  |  |
|----|---|---|---|--|--|
| 17 |  | 1 | 20: $p_0=0.5,$<br>$p_1=0.5$   |  |  |
| 18 |  | 1 | 20: $p_0=0.7,$<br>$p_1=0.3$   |  |  |
| 19 |  | 1 | 20: [0,5):<br>$p_0=0.7, p_1=0.3;$<br>[10,20):<br>$p_0=0.5, p_1=0.5$ |  |  |
| 20 |  | 1 | 20  |  |  |
| 21 |  | 1 | 20  |  |  |

continued on next page

| Num. | Length | Density-Flow Relationship |
|------|--------|---------------------------|
| 1    | 1.25   | 1                         |
| 2    | 1.0    | 1                         |
| *    |        | See Section 3.6           |

Table C.2: A summary of the arc types used in the examples, including all relevant characteristics.

*continued from previous page*

| Ex. Num | Graphic   | Arc Types | Entrance Densities        | Event 1 | Event 2 |
|---------|---|-----------|---------------------------|---------|---------|
| 22      |    | 1         | 20                        |         |         |
| 23      |    | 1         | [0,10):20;<br>[10,20):0   |         |         |
| 24      |    | 1         | [0,10):20;<br>[10,20):0   |         |         |
| 25      |   | *         | 20                        |         |         |
| 26      |  | *         | $k_0 = 30,$<br>$k_5 = 10$ |         |         |

## C.2 Example Network File

As was alluded to in the discussion of the DNLP algorithm implementation, XML was used to store the network files.<sup>1</sup> This allowed for a flexible file format and the potential to include results in the same file, if desired, thus allowing easy distribution of example files. We include below the sample input of the basic diverge network, consisting of just three arcs, all with identical properties. Note that Cost tags are included below for completeness only; they are not used by this algorithm.

<sup>1</sup>Note that the dynamic shortest paths implementation relies on a different file format which is not presented here.



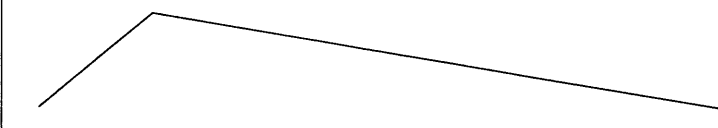
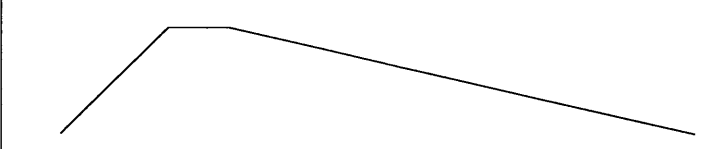
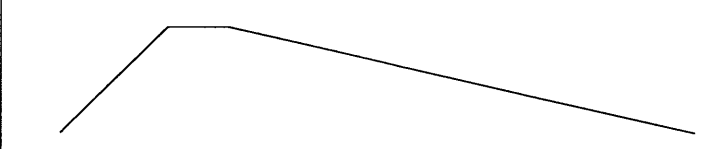
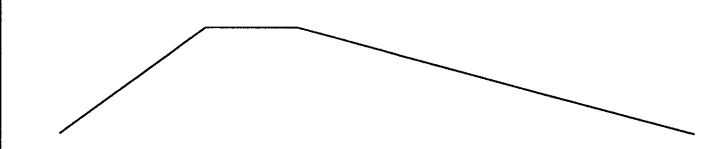
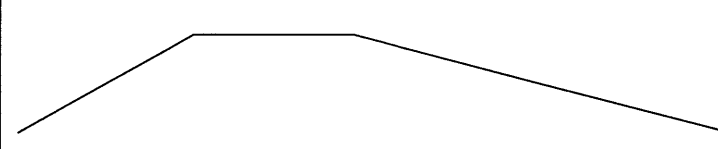
| Num. | Graphic   | Breakpoints  |
|------|---|--|
| 1    |   | (0, 0),<br>(30, 25),<br>(180, 0)                         |
| 2    |   | (0, 0),<br>(36.36, 4000),<br>(57.754, 4000),<br>(220, 0) |
| 3    |   | (0, 0),<br>(54.54, 6000),<br>(86.631, 6000),<br>(330, 0) |
| 4    |   | (0, 0),<br>(25, 2000),<br>(41.666, 2000),<br>(110, 0)    |
| 5    |  | (0, 0),<br>(30, 1800),<br>(60, 1800),<br>(125, 0)        |

Table C.3: A summary of the density-flow relations used in the examples, including all relevant characteristics. Note that the underlying units in the first example differ from the remaining relations.

| Num. | Arc | $x$ | $Q_{\max}$ | $t_0$ | $t_f$ | Duration | Delay |
|------|-----|-----|------------|-------|-------|----------|-------|
| 1    | 0   | 0.8 | 5          | 2     | 4     |          |       |
| 2    | 1   | 0.8 | 5          | 2     | 4     |          |       |
| 3    | 1   | 0.2 | 5          | 2     | 4     |          |       |
| 4    | 0   | 0.2 | 5          | 6     | 7.5   |          |       |
| 5    | 2   | 0.8 | 5          | 2     | 4     |          |       |
| 6    | 1   | 0.2 | 5          | 2     | 4     | 0.5      | 0.5   |

Table C.4: A summary of the events used in the testing of the networks. Note that the Duration and Delay columns apply to cyclic events: duration refers to the duration of the blockage and delay to the amount of time which passes after the blockage is removed before it reappears (therefore the sum of the two is equal to the period or cycle time of the event). As these only apply to cyclic events, these columns are blank in non-cyclic events.

```

<Network nodes="4" arcs="3">
  <Node ID="0" x="50" y="50"></Node>
  <Node ID="1" x="250" y="50"></Node>
  <Node ID="2" x="450" y="25"></Node>
  <Node ID="3" x="450" y="75"></Node>
  <Arc head="1" tail="0">
    <Cost>5</Cost>
    <Length>1.25</Length>
    <DensityFlowRelation number="3">
      <Breakpoint>0,0</Breakpoint>
      <Breakpoint>30,25</Breakpoint>
      <Breakpoint>180,0</Breakpoint>
    </DensityFlowRelation>
  </Arc>
  <Arc head="2" tail="1">
    <DensityFlowRelation number="3">
      <Breakpoint>0,0</Breakpoint>
      <Breakpoint>30,25</Breakpoint>
      <Breakpoint>180,0</Breakpoint>
    </DensityFlowRelation>
    <Cost>4</Cost>
    <Length>1.25</Length>
  </Arc>
  <Arc head="3" tail="1">
    <DensityFlowRelation number="3">
      <Breakpoint>0,0</Breakpoint>
      <Breakpoint>30,25</Breakpoint>
      <Breakpoint>180,0</Breakpoint>
    </DensityFlowRelation>
    <Cost>10</Cost>
    <Length>1.25</Length>
  </Arc>
  <Path>
    0,1
  </Path>
  <Path>
    0,2
  </Path>
</Network>

```

# References

- [1] B. H. Ahn and J. Y. Shin. Vehicle routing with time windows and time-varying congestion. *J. Opl. Res. Soc.*, 42:393–400, 1991.
- [2] R. Ahuja, T. Magnanti, and J. Orlin. *Network Flows*. Prentice Hall, 1993.
- [3] A Astarita, K. Er-Rafia, M. Florian, M Mahut, and S. Velan. A comparison of three methods for dynamic network loading. *Transportation Research Record*, 2001.
- [4] A. Ceder, editor. *Transportation and Traffic Theory*, Jerusalem, Israel, July 1999. Pergamon. Proceedings of the 14th ISTTT.
- [5] I. Chabini. Discrete dynamic shortest path problems in transportation applications: Complexity and algorithms with optimal run time. *Transportation Research Record*, pages 170–175, 1998.
- [6] I. Chabini. Personal Communication, January 2001.
- [7] I. Chabini, A. Glenn, S. Pallottino, and M. G. Scutellà. Reoptimization of dynamic shortest paths with respect to leaving times: algorithms and computational results. presented at the Annual Meeting of the Transportation Research Board, Washington D.C., January 2002.
- [8] I. Chabini and V. Yadappanavar. Algorithms for single-origin minimum time path problems in networks with piece-wise linear time-dependent link travel time functions. *Transp. Res. Rec.*, 2002. Accepted for publication.

- [9] B. Chazelle. The soft heap: An approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, November 2000.
- [10] M. Cremer, D. Stäcker, and P. Unbehauen. Macroscopic modelling of traffic flow by an approach of moving segments. In Ceder [4], pages 517–532. Proceedings of the 14th ISTTT.
- [11] C. Daganzo. The cell-transmission model. Part I: A simple dynamic representation of highway traffic. Technical Report UCB-ITS-PRR-93-7, University of California, Berkeley, 1993.
- [12] C. Daganzo. The cell-transmission model: A dynamic representation of highway traffic consistent with hydrodynamic theory. *Transp. Res.*, 28B(4):269–287, 1994.
- [13] C. Daganzo. The cell-transmission model, part II: Network traffic. *Transp. Res.*, 29B(2):79–93, 1995.
- [14] C. Daganzo. *Fundamentals of Transportation and Traffic Operations*. Pergamon, 1997.
- [15] C. Daganzo. The lagged cell-transmission model. In Ceder [4], pages 81–104. Proceedings of the 14th ISTTT.
- [16] C. Daganzo. Personal communications with I. Chabini subsequently relayed to the author, 2000-2001.
- [17] S. E. Dreyfus. An appraisal of some shortest path algorithms. *Journal of Mathematical Analysis and Applications*, 14:492–498, 1969.
- [18] J. Farver. Continuous time algorithms for a variant of the dynamic traffic assignment problem. Master’s thesis, Massachusetts Institute of Technology, 2001.
- [19] S. Fujishige. A note on the problem of updating shortest paths. *Networks*, 11:317–319, 1981.
- [20] G. Gallo. Reoptimization procedures in shortest path problems. *Rivista di Matematica per le Scienze Economiche e Sociali*, 3:3–13, 1980.

- [21] A. Glenn. Algorithms for the shortest path problem with time windows and shortest path reoptimization in time-dependent networks. Master's thesis, Massachusetts Institute of Technology, 2001.
- [22] D. E. Kaufman and R. L. Smith. Fastest paths in time-depenent networks for intelligent-vehicle-highway systems applications. *IVHS Journal*, 1:1–11, 1993.
- [23] M. J. Lighthill and J. B. Whitham. On kinematic waves. I: Flow movement in long rivers; II: A theory of traffic flow on long, crowded roads. *Proc. of the Royal Soc. of London*, 229A:281–345, 1955.
- [24] G. Newell. A simplified theory of kinematic waves in highway traffic. *Transp. Res.*, 27B(4):281–287, August 1993.
- [25] G. Newell. A simplified theory of kinematic waves in highway traffic, Part II: Queueing at freeway bottlenecks. *Transp. Res.*, 27B(4):289–303, August 1993.
- [26] G. Newell. A simplified theory of kinematic waves in highway traffic, Part III: Multi-destination flows. *Transp. Res.*, 27B(4):305–313, August 1993.
- [27] S. Nguyen, S. Pallottino, and M. G. Scutellà. A new dual algorithm for shortest path reoptimization. In *Transportation and Network Analysis - Current Trends*. Kluwer, 2001.
- [28] S. Pallottino and M. G. Scutellà. Dual algorithms for the shortest path tree problem. *Networks*, 29:125–133, 1997.
- [29] S. Pallottino and M. G. Scutellà. A new algorithm for reoptimizing shortest paths when the arc costs change. Draft paper, 2001.
- [30] P. I. Richards. Shock waves on the highway. *Operations Res.*, 4:42–51, 1956.