DYNAMIC SHORTEST PATH ALGORITHMS FOR IVHS APPLICATIONS

by

Andras Farkas

Diploma Engineer
Technical University of Budapest, Hungary
(1989)

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
May 1993

© Massachusetts Institute of Technology 1993
All rights reserved

Signature of Author ______________________________
Department of Civil and Environmental Engineering
May 7, 1993

Certified by ______________________________
Professor Haris N. Koutsopoulos
Thesis Supervisor

Accepted by ______________________________
Professor Eduardo Kausel
Chairman, Departmental Committee on Graduate Studies

ARCHIVES

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

JUN 08 1993

Dynamic Shortest Path Algorithms for IVHS Applications

by

Andras Farkas

Submitted to the Department of Civil and Environmental Engineering

on May 7, 1993, in partial fulfillment of the

requirements for the degree of

Master of Science in Transportation

Abstract

Time-dependent dynamic shortest path algorithms are one of the building blocks in developing routing and control strategies in Advanced Vehicle- Highway Systems (IVHS). We examine modifications of existing shortest path algorithms and examine their suitability for dynamic networks. We conclude that such algorithms do not offer the computational efficiency required for real-time applications. Consequently we develop a Dynamic Distributed Shortest Path algorithm, which is based on network decomposition. The algorithm successfully utilizes characteristics of urban transportation networks and performs well with large networks decomposed into subnetworks with few cutset nodes. The performance of the algorithm also depends on the number of destination nodes. We also examine the construction of dynamic routing tables (which facilitate transmission of time-dependent path information to users). Finally, in order to provide the information required for routing recommendations efficiently we suggest and analyze different strategies that combine routing table generation and dynamic shortest path calculations.

Thesis Supervisor: H.N. Koutsopoulos

Title: Assistant Professor in Civil Engineering

Acknowledgments

I would like to express my deepest gratitude to Professor Haris N. Koutsopoulos for his advise, supervision and patience during the course of the project. His experience and support contributed in large measure to the success of this research. Besides my parents, I have learned from him the most in my life. Most of all, however, I would like to thank him for his unforgettable friendship.

I express my deep sense of gratitude to Professor Moshe Ben-Akiva, Professor David Bernstein, and Professor Nigel H.M. Wilson and other professors at M.I.T. for their guidance and encouragement.

I am also indebted to my fellow researchers; Anil Mukundan, Qi Yang and Antulio Richetta for their help, experience and friendship.

I would be failing in my duty if I did not mention my appreciation to NTT Data Communication Systems Corporation for their partial financial support of this work.

I would wish to thank all my friends, Amalya Polydoropoulou, Ramon Tarrech, Patrick Ky, Adriana Bernardino, Rabi Mishalani, Rajesh Shenoi, Gábor Hetyei, Zoltán Szabó, Tamás Gáti, Sabine Vermeersch, Erika Kiss, Allan Brik, Dinesh Gopinath, the Shiftan and Lotan families, and all my other fellow students and friends, too numerous to list, whose friendship made my time more enjoyable and the tough times more bearable at M.I.T.

Finally, I would like to thank my beloved parents Lidia and Endre, whom I missed a lot and whom I have to thank for really everything.

Ezt a munkát Nektem ajánlom és egyben köszönöm, hogy felneveltetek, útnak indítottatok, két even át nélküloztetek és mindenben támogattatok.

Köszönök Nektek mindent Drága Szüleim.

Table of Contents

Chapter 1 Introduction ... 9
1.1 Research Objectives ... 10
1.2 Literature Review ... 12
1.3 Thesis Organization ... 17
Chapter 2 Dynamic Shortest Path Algorithms ... 18
2.1 Time Dependent Transportation Networks ... 18
2.1.1 Dynamic Travel Times Representation ... 18
2.2 Time-Dependent Shortest Path Algorithms ... 21
2.2.1 Dynamic Label Setting Algorithm ... 21
2.2.2 Dynamic Label Correcting Algorithm ... 23
2.2.3 Computational Results ... 24
2.3 Strategies for improvement ... 26
2.3.1 Strategy 1 ... 26
2.3.2 Strategy 2 ... 27
2.3.2.1 Network Decomposition ... 28
2.3.2.2 The Distributed Dynamic Shortest Path Algorithm ... 31
2.3.2.3 Implementation ... 40
2.3.2.4 Issue of Correctness ... 41
Chapter 3 Distributed Dynamic Shortest Path Algorithm Experimental Results ... 45
3.1 Distributed Algorithm Experimental Results ... 45
3.1.1 Single Processor Implementation ... 45
3.1.2 Distributed Implementation ... 46
3.2 Effect of Cutset Nodes on DDSP Algorithms Performance ... 48
3.3 Heuristics ... 51
3.3.1 Experimental Results ... 52
3.3.2 Discussion ... 54
Chapter 4 Routing Tables ... 59
4.1 Dynamic Routing Tables ... 59
4.2 Construction of Routing Tables ... 61
4.2.1 A Special Characteristic of Dynamic Routing Tables ... 61
4.3 Routing with Postorder Numbering ... 63
4.3.1 Discussion ... 67

4.4 The Rolling Horizon Approach 69
4.4.1 Implementation 71
4.4.2 Experimental Results with the RHA 75
4.5 Computational Considerations 81
Chapter 5 Conclusion 87
5.1 Conclusions 87
5.2 Future Research 90
Bibliography 92

List of Figures

Figure 1.1: Sample Network for Circulation 14
Figure 2.1: Aggregation of travel times 19
Figure 2.2: Demonstration of the FIFO assumption violation under the constant travel time aggregation strategy 20
Figure 2.3: Interpolation based travel time aggregation scheme 20
Figure 2.4: Typical shortest paths in decomposed networks 27
Figure 2.5: Decomposition of a network in four subnetworks with associated cutset nodes 28
Figure 2.6: Types of virtual links 29
Figure 2.7: Calculating costs for virtual links 30
Figure 2.8: Representation of the two level network structure 31
Figure 2.9: Example of application of Step 2 of the DDSP algorithm 32
Figure 2.10: Different steps of the DDSP algorithm in a sample network 34
Figure 2.11: Cutset-to-cutset virtual link network in Step 2 35
Figure 2.12: Update of travel times in cutset-to-cutset virtual links 35
Figure 2.13: In Step 3 only marked cutset-to-cutset virtual links are included 37
Figure 2.14: Two implementations of Step 4 39
Figure 2.15: Forward star representation of the network in Step 4 41
Figure 2.16: Three networks for demonstrating the correctness issue 42
Figure 3.1: Number of cutset nodes and the decomposition 49
Figure 3.2: Two 70x35 grid networks connected by bridges 50
Figure 3.3: Representation of one-way streets in networks 54
Figure 3.4: Network consfigurations for each step in case of one way streets 56
Figure 3.5: Urban network decomposed along main and side streets 57
Figure 4.1: Time-dependent dynamic routing table 60
Figure 4.2: Sample network - static case 61
Figure 4.3: Sample network - dynamic case 62
Figure 4.4: Shortest path tree with postorder numbers and intervals 64
Figure 4.5: Postorder intervals 65
Figure 4.6: Implementation of Posorder Numbering 66
Figure 4.7: Shortest path tree with associated label values 69
Figure 4.8: Identical subtrees in different shortest path trees 70
Figure 4.9: Construction of RHA routing tables 71
Figure 4.10: RHA Routing Tables after insertions 72

Figure 4.11: Example for Lemma 2 73
Figure 4.12: Coverage values of dt time intervals...76
Figure 4.13: Overall coverage (black bar) and coverage for periods other than t_0
as a function of the length of update interval...77
Figure 4.14 Distribution of nodes with different coverage values 78
Figure 4.15: Average coverage as a function of the distance from the center 79
Figure 4.16a: Central node surrounded by nodes with high coverage 80
Figure 4.16b: Inadequacy of the RHA with nodes in the periphery 81
Figure 4.17: Comparison of coverage for Strategy-1 and Strategy-2.............................82
Figure 4.18: Node coverage resulting from Strategy-3...83
Figure 4.19: Selected node coverage for Strategy-', Strategy-2 and Strategy-3 84

List of Tables

Table 2.1: Computational results of dynamic binary heap and dequeue algorithms 25
Table 2.2: Calculated travel times and their differences for three network types 43
Table 3.1: Computational results for the serial and the DDSP algorithm 46
Table 3.2: Computational results for each step of the DDSP algorithm 47
Table 3.3: Detailed results of DDSP algorithm in networks connected by bridges 51
Table 3.4: Running time as a function of the number of cutset nodes 52
Table 3.5: Detailed running times for different networks...53
Table 4.1: Node coverage for Strategy 1 and 2..82
Table 4.2: Coverage values for Strategies 1, 2, and 3..84
Table 4.3: Computational results for different strategies 85

Chapter 1 Introduction

Ever since the beginning of the automobile age, the number of vehicles on our roads has been increasing dramatically. In the world in general and in the United States and Europe in particular existing facilities are at or soon approaching capacity. The concept of Intelligent Vehicle-Highway Systems or IVHS has been suggested for improving the operations of the transportation system.

IVHS was introduced in the late 1980's as a collection of advanced technologies to provide a solution to the congestion problem [Muku92, Stra92, BBHK92]. The basic goal is to incorporate a wide range of technologies to improve the transportation system by better use of existing facilities in a safe, more energy efficient and environmentally friendly manner.

A wide array of technologies makes up IVHS [Stra92]. Among those are traffic engineering, computer science, computer hardware and software, control, electronics and communications. Five main functional areas were defined [Stra92, Muku92]:

- Advanced Traffic Management Systems (ATMS) which deal with traffic management problems such as traffic control, incident detection, route guidance, congestion pricing, etc.

- Advanced Traveler Information Systems (ATIS) which provide various types of information for drivers, using different road-side and on-board technologies. The provided information can be about shortest or alternate paths to destination, actual traffic conditions, nearest hotels, etc.

- Advanced Vehicle Control Systems (AVCS) which apply on-board advanced technologies to make the travel safer, more comfortable, and energy efficient. Examples of such technologies include cruise control, automatic breaking, night-vision sensors, etc. Eventually they lead to automated highways.

- Commercial Vehicle Operations (CVO) mainly apply ATIS and ATMS technologies and services to commercial vehicles, assisting in optimal routing, delivering and more efficient and safe operation.

- Advanced Public Transportation Systems (APTS) use ATMS, ATIS and CVO technologies to improve the operation of transit transportation.

The main building blocks of IVHS are ATMS and ATIS, which provide area-wide, real-time, adaptive traffic control and route guidance reflecting dynamic traffic conditions.

ATMS and ATIS operate in real-time. The route guidance and traffic control measures that are part of the ATMS/ATIS should be based on predicted traffic conditions [Muku92]. Hence ATMS/ATIS needs traffic models to predict future travel times in the network. One of the approaches used for traffic prediction is based on the Dynamic Traffic Assignment (DTA) Problem. Central in the algorithm for the solution of the DTA problem is the dynamic shortest path problem.

Since ATMS and ATIS operate in real time, efficient algorithms for the solution of the dynamic shortest path problem are required. Generation of shortest paths is indeed the bottleneck operation in generating optimal traffic control and routing strategies. This implies that if the shortest path problem can not be solved efficiently, then the effectiveness of the overall system is also jeopardized. It is hence important to understand the key issues associated with the time-dependent shortest path problem, and develop approaches that solve the problem efficiently.

1.1 Research Objectives

The main objective of this research is to develop and evaluate different time-dependent shortest path algorithms for real-time IVHS applications (time-dependent and dynamic will be used interchangeably from now on in the thesis). The research also suggests approaches for efficient construction and storage of routing tables (which facilitate transmission of information to users).

The shortest path problem has been the subject of extensive research over the years. However, the vast majority of the research deals with static networks. Research on dynamic shortest path problems for transportation applications is limited (e.g., [KaSm92], [KoXu], [ZiMa92]). These publications mainly focus on the theoretical issues and assumptions of the problem and only one paper [ZiMa92] provides a new algorithm for transportation applications. The performance of the algorithm though, is still not good enough for real-time applications.

Hence, the main part of the thesis focuses on different dynamic shortest path algorithms and the modifications that are required so that they can be used with dynamic networks. In order to achieve running time improvements different strategies, such as distributed algorithms, are examined. Since ATMS and ATIS systems may

operate under different strategies and different network types, it is important to understand the performance of the various algorithms under different circumstances.

The thesis also examines the construction of dynamic routing tables. Routing tables provide the information which node (or link) should the driver visit (or traverse) next in order to travel on the shortest path towards its destination. The type and amount of information vary depending on the different architecture and implementations. The basis for the construction of routing tables is the shortest path trees. The research is based on the assumption that shortest path calculations are obtained centrally (or in distributed computational structure but not on-board) and only the routing information is transmitted to the vehicles. The system under consideration can be described as following:

- Information is updated in the system every *dt* minute.
- Routing information is provided to the vehicle at each node. The routing information can be transmitted to the vehicle through communications between the node computer (beacon) and the vehicle on-board computer.
- The information that is transmitted to the vehicle holds routing instructions for all possible destination nodes. This feature is assumed because of privacy issues. The relevant information is selected by the on-board computer.
- Information is provided in the form of advising the vehicle which node to visit (or link to traverse) next in order to travel on the shortest path to the desired destination.

Once shortest path results are obtained the information has to be stored and in an efficient format and transmitted to the vehicle on-board computer. Because of the time-dependent environment, the information that is provided has to be time-dependent as well. This implies the use of dynamic routing tables. Dynamic Routing Tables are complex because of their time dimension. Shortest paths may change over time, thus routing tables should have the capability to provide time-dependent information. In other words routing tables should be able to give routing information from each node to every destination for all instances in time. There is a need to design and to analyze approaches to store efficiently the large amount of information that is generated by the shortest path calculations for different time intervals.

Finally once the shortest path algorithms and routing table generation approaches are developed, it is important to understand how combinations of these two approaches

may be used to further enhance the performance and efficiency of shortest path calculations.

1.2 Literature Review

The shortest path problem has been the subject of extensive research since the late 50's. Because shortest path problems are the major analytical components of numerous network flow models, a number of algorithms were developed to find the shortest paths between the nodes of the network. Even though there are many different algorithms in the literature, the algorithms can be categorized by a handful of general methods [DGKK79]. The literature typically classifies the algorithms into two groups: label setting and label correcting algorithms. The approaches vary in the method they use to update and converge to the final node labels, (labels represent the shortest path distances from the origin node). Label setting algorithms are more efficient with respect to worst-case complexity, while label correcting algorithms apply to more general cases and offer much more algebraic flexibility [AMOr93].

The first label setting algorithm was suggested by Dijkstra in 1959 [Dijk59]. Dial suggested two better implementations of the Dijkstra's algorithm [Dial69], [DGKK79], storing distance labels in so-called buckets. The algorithm is based on the property that the distance labels that the label setting algorithms assign as permanent, are non-decreasing. Dial's algorithm solves the problem in pseudo-polynomial time $O(m+nC)$, (where n is the number of nodes, m is the number of arcs, and C is the maximum arc cost in the network) and performs well in practice [DGKK79]. Later development and experience showed that another label setting algorithm, the binary-heap heap implementation of the Dijkstra's algorithm, solves the shortest path problem in sparse networks very efficiently. The binary-heap algorithm uses binary heap data structures to maintain the temporary labelled nodes. The binary-heap implementation of the Dijkstra's algorithm runs in $O(m \log n)$ time [AMOr93]. In sparse networks this implementation solves the shortest path problem faster than other label setting algorithms. A detailed description of the algorithm is provided in Chapter 2.

Label correcting algorithms iteratively update the labels associated with each node until the distance labels satisfy the shortest path optimality conditions. The first label correcting algorithm was presented by Ford in 1956 [Ford56]. Bellman presented his FIFO algorithm in 1958 [Bell58]. This algorithm is still the best known polynomial time

algorithm in the literature with running time *O(nm)*. Pape, based on an idea due to D'Esopo, developed a new algorithm that uses a dequeue as a sequence list for updated nodes [Pape74], [Pape80]. The interesting characteristic of this algorithm is that its worst-case complexity is exponential, however in practice, is very efficient [GaPa88]. This is especially true in the case of sparse networks. Dial et. al. showed that in sparse networks the dequeue implementation is the best among all available algorithms [DGKK79]. According to Ahuja et. al. the dequeue implementation has possibly linear running time in practice [AMOr93]. In chapter 2 we discuss the dequeue implementation in detail.

Because transportation networks are usually sparse networks, the binary-heap implementation of Dijkstra's label setting algorithm and Pape's dequeue implementation of the label correcting algorithm receive serious attention in this research.

Shortest path problems with time-dependent costs were first considered by Cooke and Halsey in 1966 [CoHa66]. They have developed a function that uses the enhanced version of Bellman's principle of optimality [Bell57] to obtain time-dependent shortest paths from source nodes to a destination node. The algorithm assumes that the arc costs, that are discretized multiples of dt, are known for each T=$\{t_0, t_0+dt, t_0+2*dt, \ldots, t_0+K*dt\}$ discrete time intervals. The time complexity of the algorithm is $O(n^3K)$.

Dreyfus [Drey69] suggested a modification of Dijkstra's algorithm [Dijk59] eliminating the restrictions of discretized time intervals and proving that the time-dependent shortest path problem can be solved by the same worst case complexity, $O(n^2)$, as that of the Dijkstra's algorithm in static networks. Hall [Hall87] considers networks with randomly generated time-dependent networks and shows that in general static algorithms cannot be applied for the time-dependent cases.

The introduction of dynamic travel times generates two questions:

a) Under what conditions can static shortest path algorithms be used for the dynamic case?

b) What modifications to the standard algorithm are required?

To demonstrate potential problems regarding the first question, let us consider the following example:

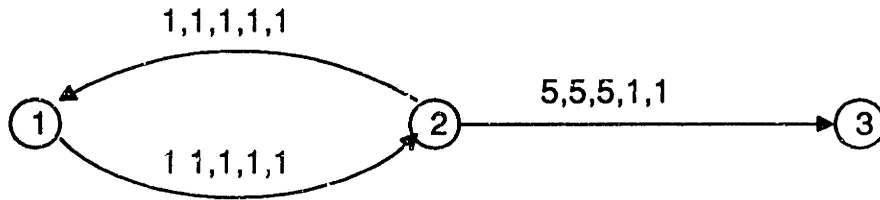


Figure 1.1: Sample Network for Circulation

where the numbers indicate link travel times starting at t=0.

Clearly the shortest path from *1* to *3* in this case is the following sequence of nodes *1-2-1-2-3*. The motorist is better off either waiting at node *2* for conditions to improve or cycling around node *1*. Recently Kaufman and Smith [KaSm92] and independently Koutsopoulos and Xu [KoXu93] identified conditions under which the above situation does not occur.

Let the departure time of a motorist from node i be t_i and the travel time on link (i,j), $c_{ij}(t_i)$. Then the arrival time at j is given by:

$$t_j = t_i + c_{ij}(t_i)$$

It is required that the arrival time t_j be a monotone increasing function of the departure time t_i from node i so that there is no incentive for motorists to wait at node i. This condition can be expressed as:

$$\frac{dt_j}{dt_i} = 1 + \frac{d}{dt_i} c_{ij}(t_i) \geq 0$$

which is equivalent to:

$$\frac{d}{dt_i} c_{ij}(t_i) \geq -1$$

If all travel times on links in the network satisfy the above condition then no motorist departing later from the origin can arrive earlier at the destination following the

same path (we will refer to it as the FIFO condition), i.e., there is no waiting at any node of the network (for the traffic conditions to change). Under these conditions there is no benefit to wait in any node in the network and the standard shortest path algorithms (with modifications) can be used to determine the correct time dependent shortest path.

Kaufman and Smith [KaSm92] proved that if the FIFO assumption holds, then both label correcting and label setting algorithms for static networks, after modification, can be used for solving the time-dependent problem correctly. They also proved that the worst-case running time bound of the dynamic algorithm is identical to that of the static one. The required modifications are introduced in Chapter 2 of the thesis.

Halpern discusses a time-dependent shortest path problem for a general case where waiting at nodes is possible [Halp77]. Orda and Rom in [OrRo90] address the problem of time-dependent shortest path considering three types of networks. In the first type, delay is possible at any node; in the second, delays are permissible only at the origin node; and in the third, vehicles are not permitted to delay anywhere. In [OrRo91] they suggest improved algorithms and formulate the criterion that guarantees the existence of finite optimal path. For IVHS applications only the second and third network types are useful. The second type of network condition may be applicable in advanced ATMS systems, where information may be provided, to commuters at their work place, to wait for better conditions. The third type of network represents the general problem of time-dependent shortest path calculations in urban transportation networks.

For IVHS applications, to the best of our knowledge, only one paper has been published. Ziliaskopoulos and Mahmassani [ZiMA92] introduce a time-dependent algorithm that calculates the time-dependent shortest paths from all nodes to a destination node. The algorithm is based on the Bellman's principle of optimality and, starting from the destination nodes, it calculates the shortest paths operating backwards. The algorithm assumes that travel times on each arc are given in the form $S=\{t_0,t_0+dt,t_0+2*dt,\ldots,t_0+K*dt\}$, where dt is very small; t_0 is the earliest departure time from any origin node in the network and K is a large integer such that the time interval covers the period of interest (the whole peak period for instance). The algorithm, using dequeue data structures, iteratively expands the network backwards from the destination node, while calculating shortest paths from all nodes to the destination node, for all time intervals in S. The article claims that one of the advantages of the algorithm is that it calculates the shortest paths not only for one but for all time intervals. However,

for IVHS applications, the system only needs shortest paths for the time periods between information updates (this interval is much smaller than the entire time horizon, which is required by the algorithm in order to operate correctly). If K is large (as long as the longest expected travel time in the network), then the number of shortest paths to be solved from all nodes to the destination node is prohibitively large. For instant the article reports that to calcualte shortest paths from all nodes to a destination node on a Cray Y/MP-8 supercomputer, in a random network with 1000 nodes and 2500 arcs and K=240, takes 73.28 milliseconds, while in a random network of 2500 nodes, 8000 arcs and K=240 it takes 235.04 milliseconds. The shortest paths that the [ZiMA92] algorithm calculates between the new update time and t_0+K*dt have no use and only consume computer resources. An additional disadvantage of the algorithm is that if the number of origin nodes is small it needs to calculate almost from all nodes the shortest paths for all K time intervals. The advantage of the algorithm is that it calculates shortest paths only to destination nodes, however if the number of destination nodes is large it may loose its advantage.

In the field of distributed algorithms the literature mentions two types of decomposition based algorithms for solving the shortest path problem: linear decomposition [Hu1969] and tree decomposition [BlHu77] algorithms. For IVHS applications Habbal [AlHA91] developed a network decomposition algorithm that solves that static shortest path problem on a SIMD fine grained, massively parallel computer, the Connection Machine. He compares the running times on different architectures and draws conclusions on appropriate network decomposition strategies.

Though there are relatively many distributed parallel shortest path algorithms developed, according to our knowledge, there is no time-dependent shortest path algorithm developed.

The literature on routing tables deals exclusively with communication networks. Most of them deal with the trade-offs between space and efficiency [PeUp89],[Sega77]. Many applications use different hierarchical routing techniques [KlKa77],[PeUp89]. In general the algorithms and assumptions of routing strategies developed for communication networks can be used for IVHS applications only to a very limited extend. A main difference is that in communication networks routing instructions are transmitted over the same network as the messages. Furthermore, in transportation networks, the trade-off between routing and space efficiency is more limited. Drivers expect correct travel

information and if we want them to comply with the instructions, the provided information should be reliable. Rerouting and longer paths due to storage optimization considerations are not acceptable.

1.3 Thesis Organization

The remaining of the thesis is organized as follows. Chapter 2 introduces the time-dependent transportation network environment and discuses the basic assumptions of the problem. It compares different dynamic serial algorithms and introduces a new Distributed Dynamic Shortest Path algorithm (DDSP). Chapter 3 analyses the effects of the number of cutset nodes in the network and introduces some heuristics. It also discusses the advantage of the DDSP algorithm in networks with special characteristics. Chapter 4 deals with routing tables and the associated special issues due to the dynamic nature of the problem. Finally Chapter 5 concludes the thesis by summarizing the results and making suggestions for further research.

Chapter 2 Dynamic Shortest Path Algorithms

2.1 Time Dependent Transportation Networks

Following the literature review and discussion of the dynamic shortest path problem in Chapter 1 in this section we introduce the required modifications of the static shortest path algorithms for the solution of the dynamic shortest path problem. Since transportation networks are sparse the algorithms of interest are the dequeue implementation of the label setting algorithm and the binary-heap implementation of the label setting algorithm.

2.1.1 Dynamic Travel Times Representation

Dynamic travel times, generated by a traffic prediction algorithm, are inputs to the problem. These inputs are presented in a discrete form, i.e., for each time interval there is an associated travel time. The form and representation of the data are important issues. Ideally travel times should be represented by appropriate units so that all values are integers and the discretization of time intervals should have the same resolution as the travel time units. Then travel times can be stored in an array and the travel time on arc (i,j) at time t_k would be easily found by looking up the t_k-th entry of the cost-array of arc (i,j). Representing travel times in this form however has some drawbacks. In urban networks link travel times may be represented in units of seconds and the size of the time horizon should at least be equal to the travel time of an average longest-path in the network. In large networks the time horizon could be in the order of 30 minutes. The representation of 30 minutes in the network with seconds as time-units would require an array of size 1800 elements for each arc. If we take into account that urban networks contain many thousands of arcs and each integer value requires at least two bytes, then the memory requirements (36 Mbytes) become very high.

To reduce the memory requirements we need to compress the data. A simple approach is through aggregation. Time for example is discretized in half-minute, minute or n-minute time intervals with the corresponding average travel time predictions assigned in each interval. This aggregation however, may generate some complications and inconsistency in the shortest path calculations, which should be resolved.

If data is aggregated, i.e., time intervals are longer than the time unit of the system, the FIFO property, which the original data satisfies, may be violated. Let us assume that travel times are given in units of seconds and stored for each one minute interval, i.e., travel time is given at $\{t_0, t_0 + 1\,\text{min}, t_0 + 2 * 1\,\text{min}, \ldots\}$. Since travel times are in seconds it is likely that during the iterations of the shortest path algorithm we would need the travel on a link at time *t*, with $t_0 + k < t < t_0 + (k+1)$ (see Figure 2.1.)

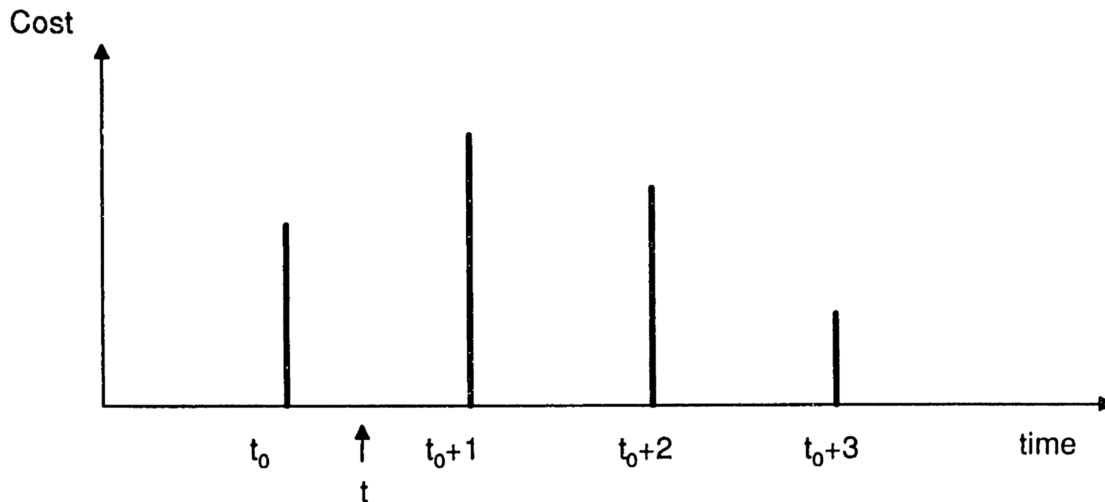


Figure 2.1: Aggregation of travel times

One (naive) way to proceed in this case is to assume that travel time does not change during a time interval, i.e., travel time remains constant for the entire interval. As Figure 2.2 shows, this approach would easily create instances where the FIFO assumption is violated, thus producing the incorrect shortest path results. The slope of the change in travel times between times t_2 and t_1 for example, is smaller than -1. Thus, the FIFO assumption is violated.

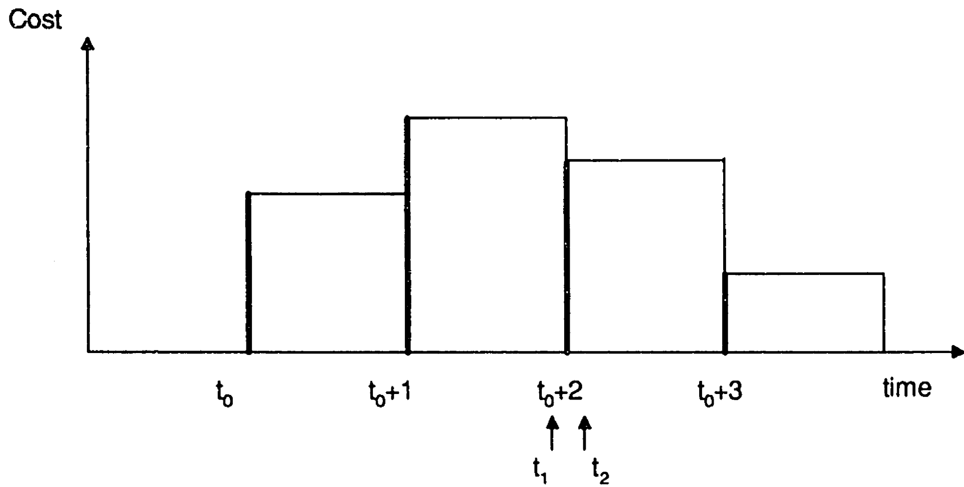


Figure 2.2: Demonstration of the FIFO assumption violation under the constant travel time aggregation strategy

One way to overcome the problem is to interpolate between the given cost values (see Figure 2.3). It can be easily proved that if discrete travel costs are consistent and the FIFO assumption holds, i.e., the change in travel times between two intervals is not grater than the interval size, then, with linear interpolation, the FIFO assumption always holds for the interpolated values.

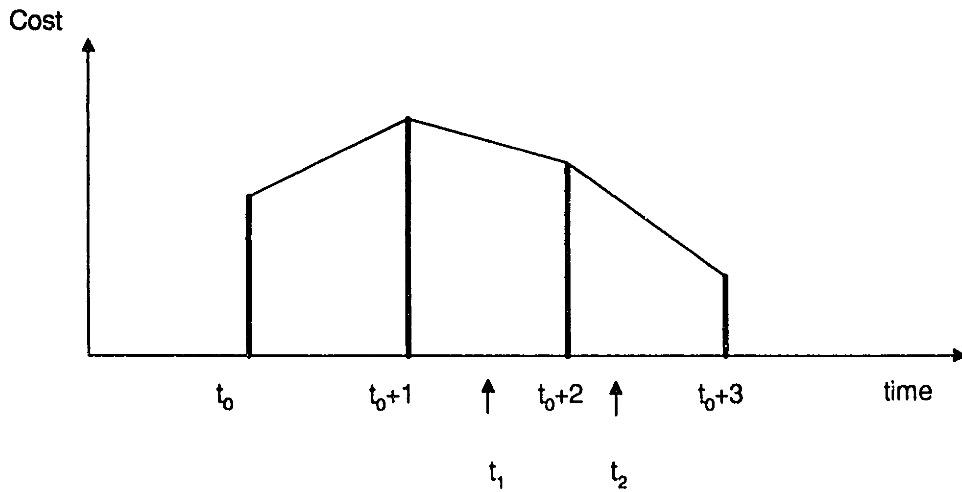


Figure 2.3: Interpolation based travel time aggregation scheme

2.2 Time-Dependent Shortest Path Algorithms

Assuming that the FIFO conditions are satisfied we now present two algorithms (label-correcting and binary-heap label setting) and the needed modifications for the solution of the dynamic one-to-all shortest path problem.

The general label updating function in the case of dynamic networks becomes:

$$d_j = \min_i(d_i + c_{ij}(d_i))$$

where, d_j is the shortest path travel time from the origin node to node j (leaving at time t_0), and $c_{ij}()$ is the cost function of link (i,j). The argument in the cost function $c_{ij}()$ is the arrival time at node i, which is given by d_i. The optimality condition for the Dynamic Shortest Path problem then becomes:

$$d(s,j,t_0) \leq d(s,k,t_0) + d(k,j,t_k), \quad \text{for } \forall j,k \in N$$

where s is the source node and $d(k,l,t)$ represents the distance label (travel time) from node k to node l leaving k at time t.

2.2.1 Dynamic Label Setting Algorithm

Due to its efficiency for sparse networks, a binary-heap implementation of Dijkstra's algorithm was chosen. The binary heap data structure requires $O(\log n)$ time to perform every heap operation. Consequently the binary heap version of Dijkstra's algorithm runs in $O(m \log n)$ time (where n is the number of nodes and m the number of arcs). The binary heap implementation is slower than the original Dijkstra's algorithm for completely dense networks, but it is faster for sparse networks [AMOr93]. Since transportation networks are usually sparse the use of the binary heap implementation provides a good alternative for testing.

The heap is a data structure that allows the following operations to be performed efficiently on a collection H of objects, each with an associated number called its key [AMOr93].

create-heap(H): Creates an empty heap.

find-mini(H): Finds and returns an object i of minimum key.

insert(i,H): Inserts as new object i with a predefined key.

decrease-key(value,i,H): Reduces the key of an object i from its current value to "value", which must be smaller than the key it is replacing.

delete-mini(i,H): Deletes object i with minimum key.

In the implementation of Dijkstra's algorithm the heap H is the set of nodes that currently have finite temporary distance labels and the key of a node is its distance label. With this modification the binary heap implementation of Dijkstra's algorithm is the following:

```
algorithm Dynamic binary-heap ;
begin
  create-heap(H);
  d(j,t0):=INF for all j ∈ N;
  d(s,t0):=t0 and pred(s):=-1
  insert(s,H);
  while H ≠ ∅ do
  begin
    find-mini(H);       /*find node i in H with minimum distance label */
    delete-mini(i,H);  /*delete that node from the heap */
    for each i,j ∈ A(i) do
    begin
       value:= d(i,t0) + cij(d(i,t0))
       if d(j,t0) > value then
          if d(j,t0)=INF then
               d(j,t0):=value, pred(j):=i and insert(j,H)
          else  set d(j,t0):= value, pred(j):=i, and decrease-key(value,j,H);
    end;
  end;
end;
```

A(i) is the vector of arcs emanating from node i, node s is the starting node, N is the set of nodes in the network, $d(i,t_0)$ is the arrival time at i starting from node s at t_0; $pred(i)$ holds the predecessor node to node i on the shortest path.

After termination of the dynamic binary-heap shortest path algorithm the labels $d(j,t_0)$ indicate the arrival time at node j and $d(j,t_0)-t_0$ is the travel time on the shortest path from

node s to j. By tracing back the predecessor list *pred(j)* it is easy to identify the shortest path between s and each node j in the network.

2.2.2 Dynamic Label Correcting Algorithm

The Label Correcting Algorithm we tested, is a modified version of the general Label Correcting Algorithm using a sequence list. The Generic Label Correcting Algorithm's has worse case running time $O(n^2mC)$ while the Dequeue implementation has a $O(\min(nmC, m2^n))$ running time (where C represents the maximum cost value on links) [AMOr93]. In practice, especially with sparse transportation networks the Dequeue implementation has superior performance [DGKK79].

In the Dequeue implementation (Pape's Algorithm) the sequence list is a dequeue. The dequeue is a data structure that permits to store a list so that one can add or delete elements from the front as well as the rear of the list [Pape74]. The dequeue implementation always selects nodes from the front of the dequeue, but adds nodes either in the front or in the rear. If the node has been in the List earlier, then the algorithm adds it in the front; otherwise it adds it to the rear. This heuristic has the following intuitive justification: if node i has previously appeared in the list, then some nodes, say $1,2,..,k$, may have node i as its predecessor. Suppose further that the List contains the nodes $1,2,..,k$ when algorithm updates $d(s,i,t)$ again. It is then advantageous to update the distance labels of nodes $1, 2,..,k$ from node i as soon as possible. Adding node i to the front of the List tends to correct the distance labels of nodes $1,2,..,k$ quickly and reduces the need to re-examine nodes.

The modified dequeue implementation of the label correcting algorithm is the following:

```
begin
  d(s,t0):= t0; pred(s)= -1;
  d(j,t0):= INF for each j ∈ N - {S};
  LIST := {s};
  while LIST ≠ ∅ do
  begin
    remove element i from beginning of LIST;
    for each (i,j) ∈ A(i) do
    if d(j,t0) > d(i,t0)+c_{i,j}{d(i,t0)} then
    begin
      d(j,t0):= d(i,t0)+c_{i,j}{d(i,t0)};
      pred(j):= i;
      if j ∉ LIST then
        if j was on the LIST before then
          add j to the front of LIST;
        else
          add j to the rear of LIST;
    end;
  end;
end;
```

2.2.3 Computational Results

In this section we evaluate the computational results of the two algorithms calculating shortest paths from all nodes to all nodes for time t_0. We run the dynamic one-to-all algorithms sequentially from all nodes. The algorithms were tested on two real networks (the Sudbury network with 205 nodes and 578 links, and the Boston network with 4922 nodes and 11647 links) and three randomly generated grid networks with 20x20, 40x40, and 70x70 nodes. Each node has arcs adjacent to its neighbor nodes. For every arc the dynamic travel costs are stored in an array of size 40. The time horizon was longer than the longest expected travel time between any two nodes of the network. In the Sudbury network each time interval represented 20 seconds, while in the Boston and the grid networks each time interval corresponded to a 1 minute period. The travel costs of Sudbury network were based on the output of a traffic simulation model under development at MIT [Muku92]. The link travel times for the Boston network were based on the free flow travel times and were randomly increased to incorporate dynamic characteristics. The link travel times for the grid networks were randomly assigned and they satisfy the FIFO assumption. Table 2.1 shows the computational results of the

algorithms on a SUN-4 workstation. The code was written in C++ and was compiled using the AT&T C++ compiler version 2.1.

	average # of links/node	Binary Heap	Dequeue
Sudbury Network (205 nodes, 578 links)	2.81	3.8 sec	2.3 sec
Grid Network 20x20 (400 nodes, 1520 links)	3.80	14.3 sec	8.8 sec
Grid Network 40x40 (1600 nodes, 6240 links)	3.90	244.0 sec	160.0 sec
Grid Network 70x70 (4900 nodes, 19320 links)	3.94	1982.0 sec	1194.3 sec
Boston Network (4922 nodes, 11647 links)	2.36	1866.0 sec	1086.4 sec

Table 2.1: Computational results of dynamic binary heap and dequeue algorithms

The empirical results show that:

- Both dynamic shortest path algorithms solve the time-dependent shortest path problem correctly.
- The label correcting dequeue implementation of the shortest path algorithm for all network sizes outperforms the binary heap implementation of Dijkstra's algorithm.
- The computational time for large networks exceeds the acceptable time for real time implementation.

2.3 Strategies for improvement

In order to improve the performance of the algorithms so that they satisfy the real-time requirements the following strategies for running time improvements were examined:

Strategy 1 Since only shortest paths to destination nodes are of interest the Label Setting Algorithm may terminate when all destination nodes become permanent.

Strategy 2 The algorithms are implemented in a multiprocessor/distributed environment.

2.3.1 Strategy 1

In transportation networks the number of destination nodes is much less than the number of total nodes in the network, and both algorithms give shortest paths to all nodes. However, the label setting algorithm may terminate when the labels of nodes of interest become permanent, but this is not true with the label correcting algorithm. Therefore, to achieve running time improvements, we tested the idea of terminating the label setting algorithm when all destination nodes are added to the list. Since in label setting algorithms when a node enters the list it becomes a permanent node, it is possible to stop the one-to-all shortest path calculation when all the destination nodes become part of the list. Naturally, the time when all destination nodes are entered the list depends on the distribution of destination nodes in the network, the actual dynamic characteristics of the shortest path tree, and other characteristics of the network.

To test the method we run this algorithm on the Sudbury and Boston networks. The experimental results on the modified binary heap implementation of the Dijkstra's algorithm showed that a 10% running time improvement can be achieved. However, even with this reduction the dequeue implementation of the label correcting algorithm still outperforms the binary heap implementation.

2.3.2 Strategy 2

The basic approach in strategy 2 is based on a distributed dynamic shortest path algorithm, which consists of the following steps (see also [AlHa91]):

1. decompose the network into a few subnetworks;

2. solve the time-dependent shortest path problem parallel in each subnetwork (using a few processors);

3. organize partial results to obtain global solution;

Given this general approach the correctness of the algorithm requites that the shortest paths are correctly determined in the following cases:

- both the origins and destinations are in the same subnetwork and the shortest path belongs in its entity in the subnetwork (Figure 2.4 path a);

- both the origins and destinations are in the same subnetwork but the actual shortest path loops out of the borders of the subnetwork (Figure 2.4 path b);

- the origin and destination nodes are in different subnetworks (Figure 2.4 path c).

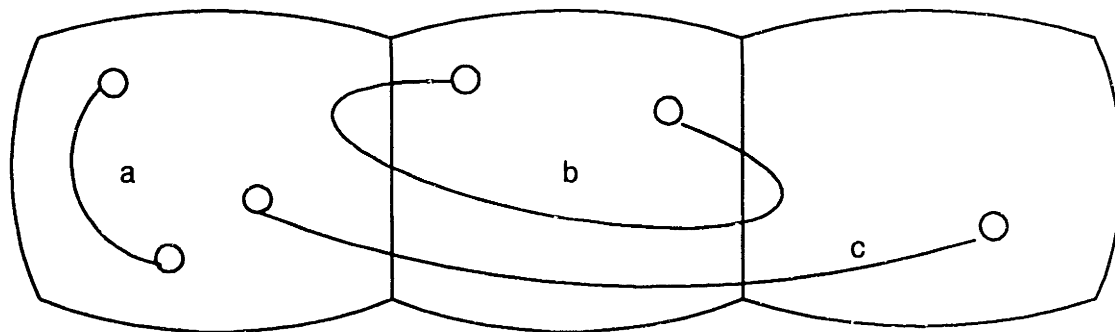


Figure 2.4: Typical shortest paths in decomposed networks

The dynamic nature of the problem provides additional difficulty when combining the partial results to obtain the global solution. Additional calculations are required to obtain the correct solution.

2.3.2.1 Network Decomposition

The set of nodes C is called a cutset, if by eliminating those C nodes from the network, the remaining network becomes disconnected (see Figure 2.5).

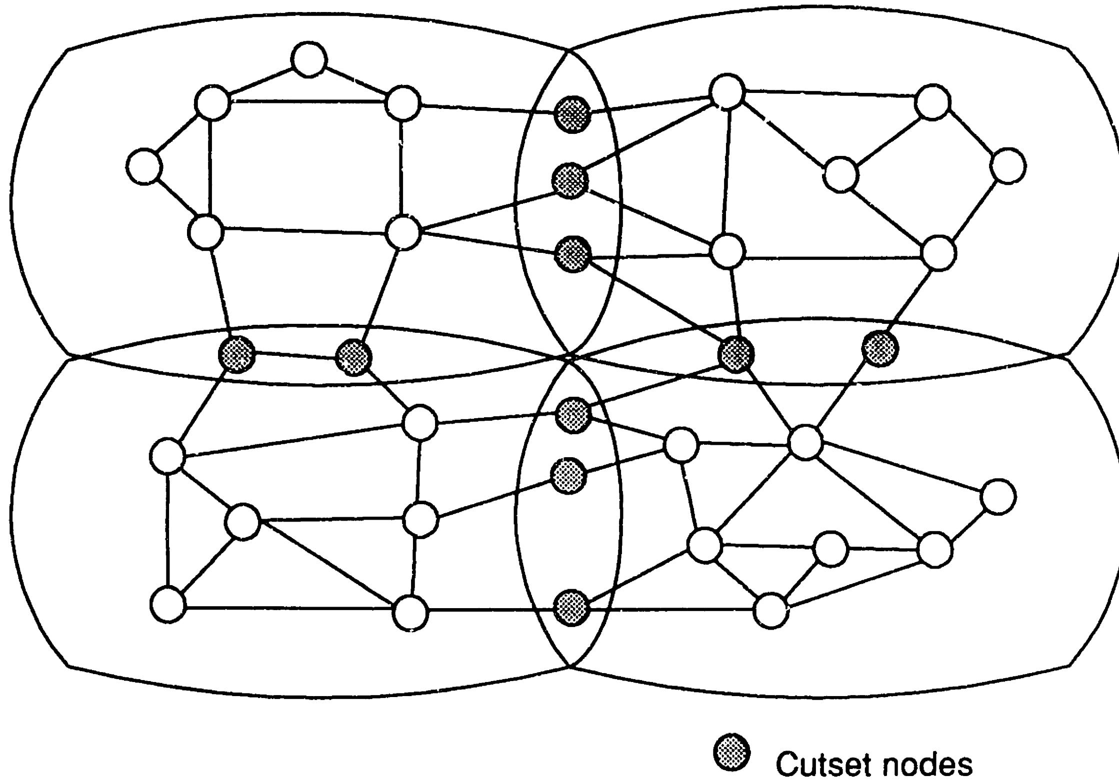


Figure 2.5: Decomposition of a network in four subnetworks with associated cutset nodes

After decomposition we introduce virtual links to the original network. These virtual links are of three types:

- node-to-cutset virtual links: link between each node of a subnetwork to all cutset nodes of the same subnetwork (Figure 2.6a).
- cutset-to-cutset virtual links: a virtual link between each pair of cutset nodes in the same subnetwork (Figure 2.6b). Note, that because of the double representation of cutset nodes such virtual links will be defined in more than one subnetworks.
- cutset-to-node virtual links: virtual links between each cutset node to all destination nodes of the same subnetwork (Figure 2.6c). Note that the number of destination nodes is less than the total number of nodes in the network. Creating virtual links only to destination nodes, as we will see later, plays a very significant role in improving the efficiency of the algorithm.

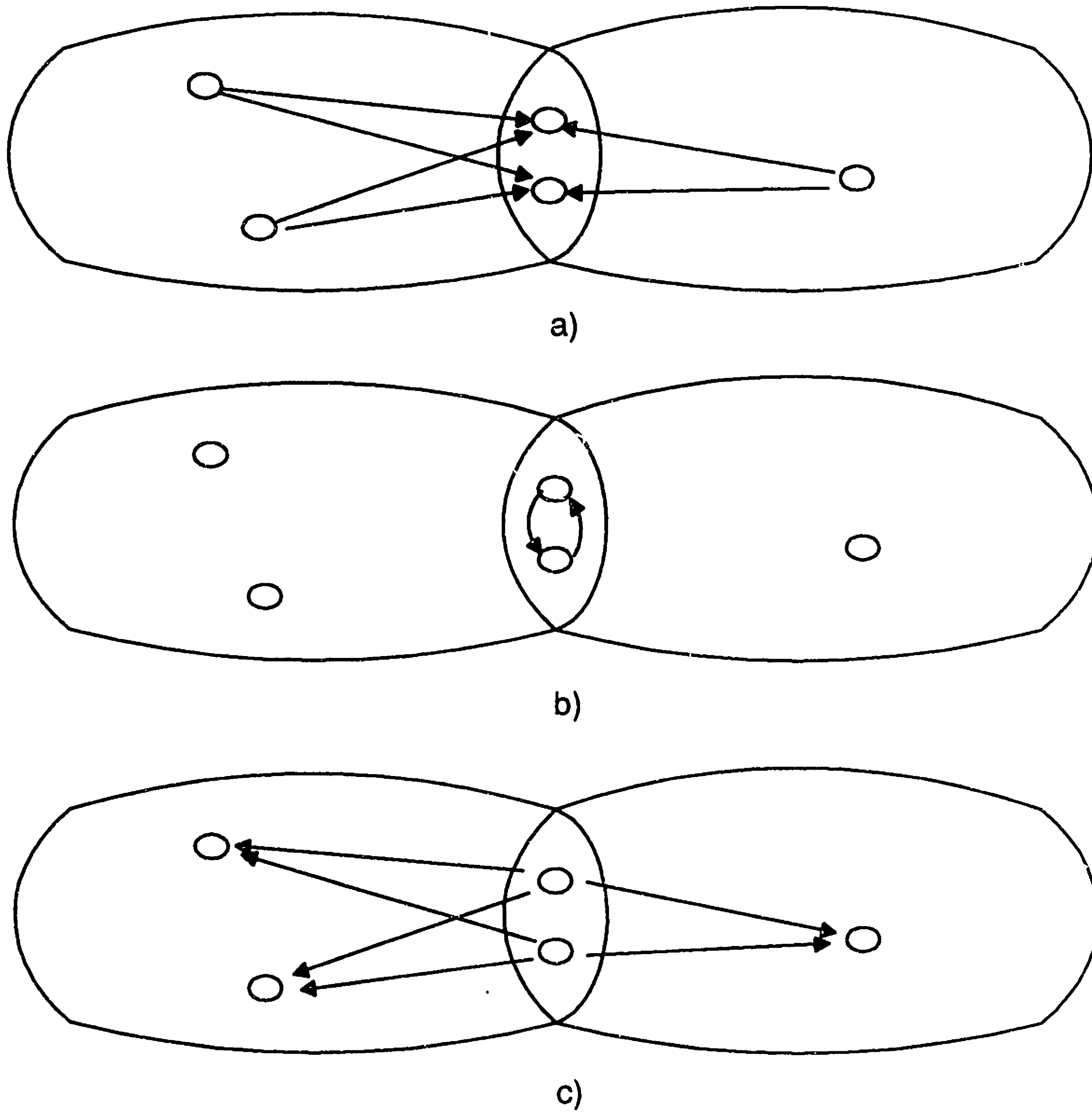


Figure 2.6: Types of virtual links

Because of the dynamic characteristics of the problem, virtual links have to be time-dependent. To assign costs to a virtual link, for example between nodes c and i, (see Figure 2.7), we have to solve the time-dependent shortest path problem between these two nodes. The cost corresponding to t_0 time interval on the virtual link (c,i), is equal to the travel time on the shortest path from node c to node i leaving at t_0. Similarly, the cost corresponding to the t_{10} time interval on virtual link (c,i) equals to the travel time on the shortest path leaving c at t_{10}. Note, that travel times on shortest paths are represented directly in virtual links even though, as Figure 2.7 shows, the actual shortest paths may be different.

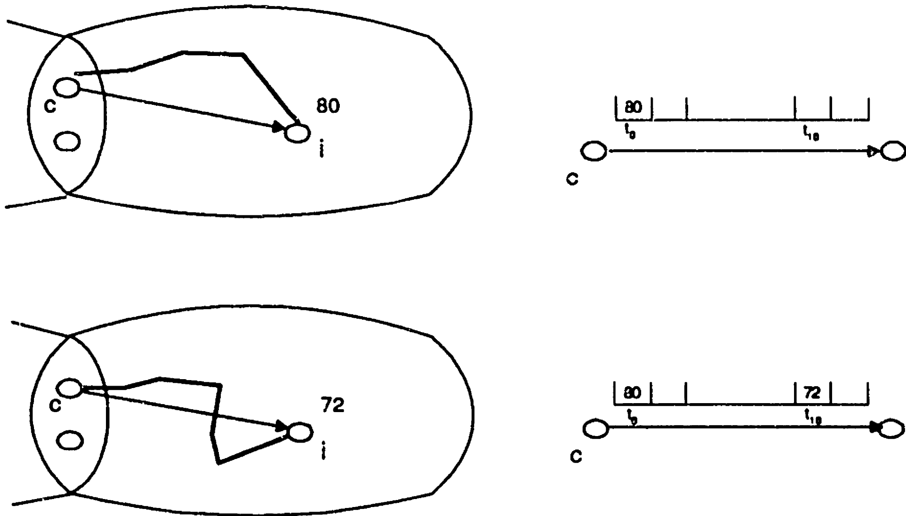


Figure 2.7: Calculating costs for virtual links

With the introduction of virtual links we can create a two-level network structure, where each individual subnetwork with its corresponding nodes and original links is at the lower level, and all nodes with the virtual links are at the higher level. However, the cutset-to-cutset virtual links are represented in both the low and high levels.

As Figure 2.8 shows the higher level network is connected with the low level subnetworks through its virtual links. Note also, that at each level the network components have to be dynamically changed and updated. This means that through the different steps of the algorithm different sets of network components are actively examined. Specifically, as we will see, at one step only the cutset-to-cutset virtual links are represented, while at another step the node-to-cutset with the cutset-to-destination node virtual links.

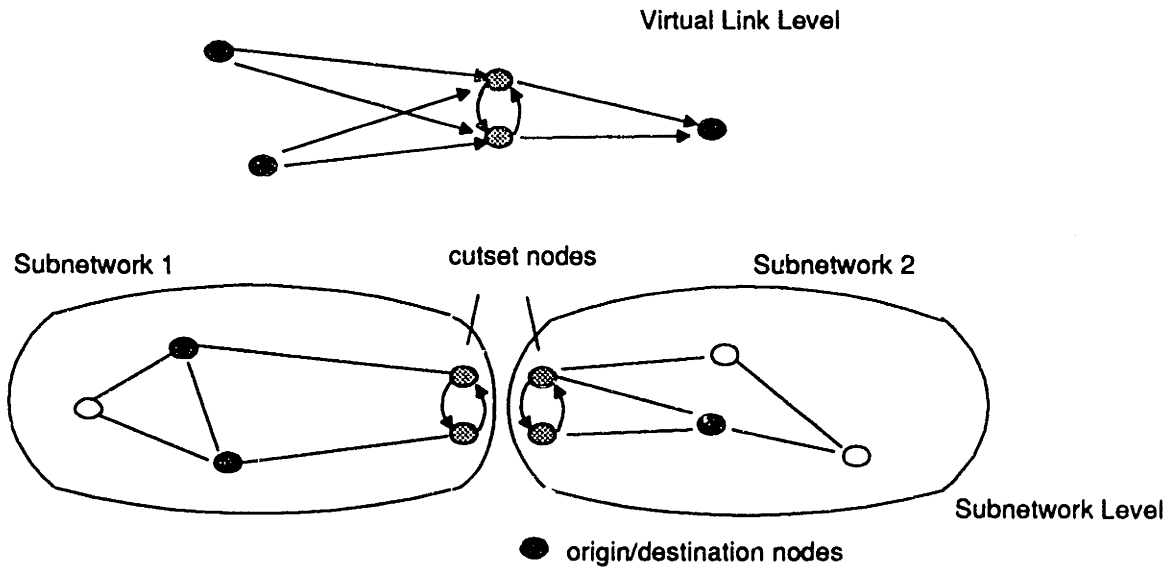


Figure 2.8: Representation of the two level network structure

This two level network structure is an essential part of our approach to compose the information obtained at the lower levels via the high level network representation. It is important to mention that at the virtual link level the links from cutset nodes are pointing only towards to the destination nodes. This characteristic reduces the required size of the virtual network and improves its performance.

2.3.2.2 The Distributed Dynamic Shortest Path Algorithm

We now present the Distributed Dynamic Shortest Path Algorithm (DDSP) and its implementation assuming that there are $p \geq 1$ processors available. We assume that the network is already divided into exactly p (balanced) subnetworks and the virtual links, discussed earlier, have been generated. Each processor is responsible for one subnetwork. Each processor has its own memory and the information is exchanged by fast communication among the processor units, (processor modules). We assume that p is small number, i.e., the number of processors (computers) remains within the limits of the transportation project's needs and budget constraints.

Distributed Dynamic Shortest Path Algorithm (DDSP)

STEP 1

Calculate shortest paths from cutset nodes to all nodes (including other cutsets) in each subnetwork s_i for all time intervals $t \in T$, (where $T=\{t_0, t_0+dt, t_0+2*dt, \ldots, t_0+r*dt\}$, and r is the number of time intervals in the time horizon). The network in this step consists only of those original links and nodes that belong to the actual subnetwork s_i, (i.e., virtual links are not part of the network.) However, the costs are calculated for all cutset-to-destination and cutset-to-cutset virtual links.

STEP 2

Find the shortest paths among all cutset nodes in the network for all $t \in T$, using only cutset-to-cutset virtual links (see Figure 2.9). Update the new costs on cutset-to-cutset virtual links. After this step the real shortest paths between cutset nodes will be available for all subnetworks. Each processor module is responsible for calculating the shortest paths from |c|/p cutset nodes, where |c| is the cardinality of the cutset. A virtual link $(i,j) \in s_i$ is marked, if for at least one time interval, the shortest path does not use exclusively links in s_i. All marked arcs then represent shortest paths that connect cutset nodes through nodes in a subnetwork other than s_i

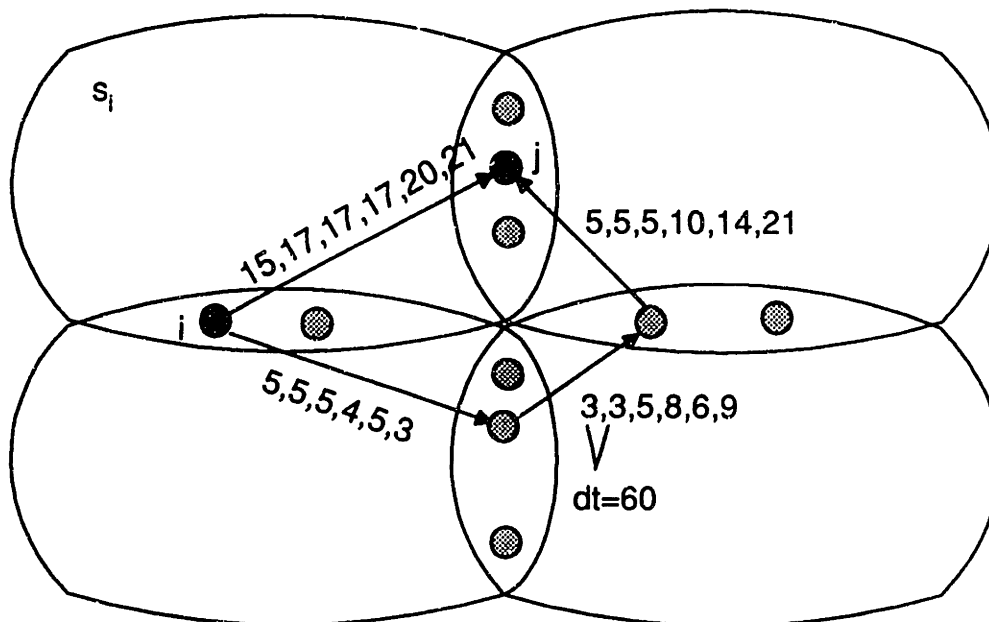


Figure 2.9: Example of application of Step 2 of the DDSP algorithm

STEP 3

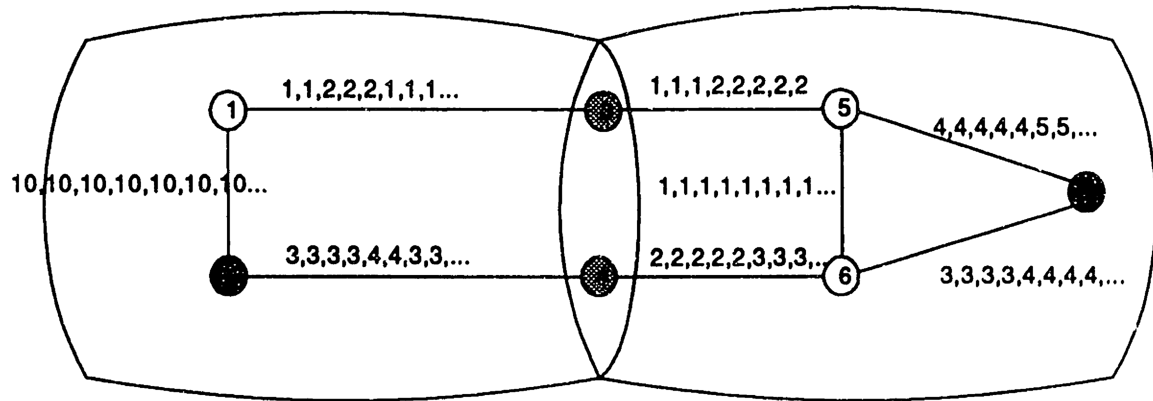
Find the shortest paths from all nodes to all other nodes in each subnetwork s_i, for $t=t_0$. Each subnetwork consists of the original links and the cutset-to-cutset virtual links marked in step 3. Update the costs on node-to-cutset virtual links.

STEP 4

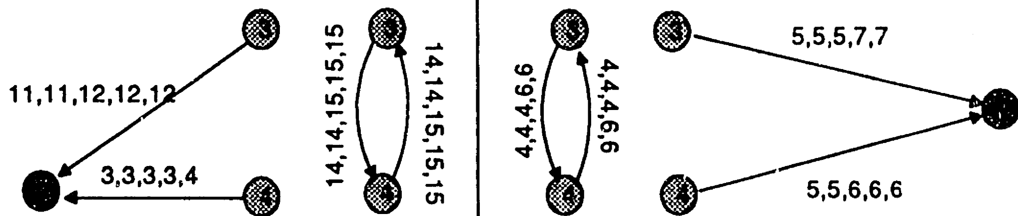
On the virtual link network find the shortest paths from all nodes k in each subnetwork to all destination nodes l such that $k \in s_i$ and $l \notin s_i$. Solve only for $t=t_0$.

In order to understand the algorithm better, Figure 2.10 shows the results of each step on a small sample network; nodes 2 and 7 are the only destination nodes (dark gray on Figure). In step 1 we calculate the shortest paths for time $t \in T$ from node *3* and *4* in both networks. The calculation of costs for the virtual links (*3*,*4*) and (4,3) in both subnetworks is shown on the figure. As a result of the calculations the costs for the cutset-to-destination virtual links, (*3*,*2*), (*4*,*2*), (*3*,*7*), and (*4*,*7*) are also obtained. Note, that the costs on the virtual links are equal to the travel times on shortest paths in the subnetwork for departures at different time intervals.

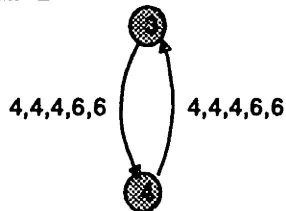
In step 2 we calculate the shortest paths between cutset nodes based only on the cutset-to-cutset virtual links. In the case of Figure 2.10 this results in assigning costs to virtual links (*3*,*4*) and (*4*,*3*) calculated in the right subnetwork, since for all time intervals the shortest path between nodes *3* and *4* were smaller there. We also mark the virtual links (*3*,*4*) and (*4*,*3*) in the left subnetwork implying that the shortest path between these nodes went through the other subnetwork for at least one time period. For large networks this step can be complicated. Suppose that we have a network divided into four subnetworks. Figure 2.11 shows the cutset-to-cutset virtual links of this network (each link represents two actual links in both directions.) The number of cutset-to-cutset virtual links in each subnetwork grows rapidly with the number of cutset nodes in of subnetwork. If c_i is the number of cutset nodes in s_i, then the total number of cutset-to-cutset virtual links equals to c_i^2. To exploit parallelism we can assign *1*/*p* of the cutset nodes to each processor module and calculate shortest paths from that set of nodes only. In each processor module the underlying network is the same.



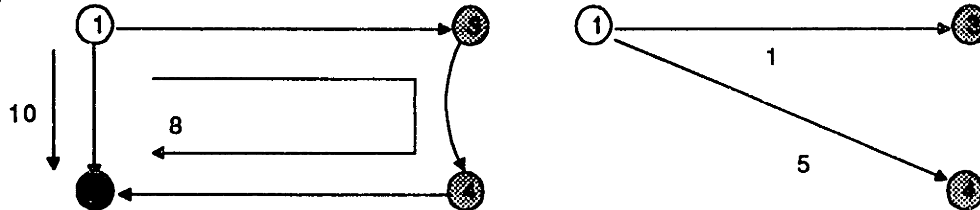
STEP 1



STEP 2



STEP 3



STEP 4

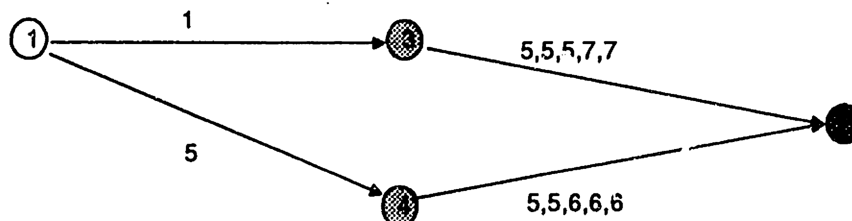


Figure 2.10: Different steps of the DDSP algorithm in a sample network

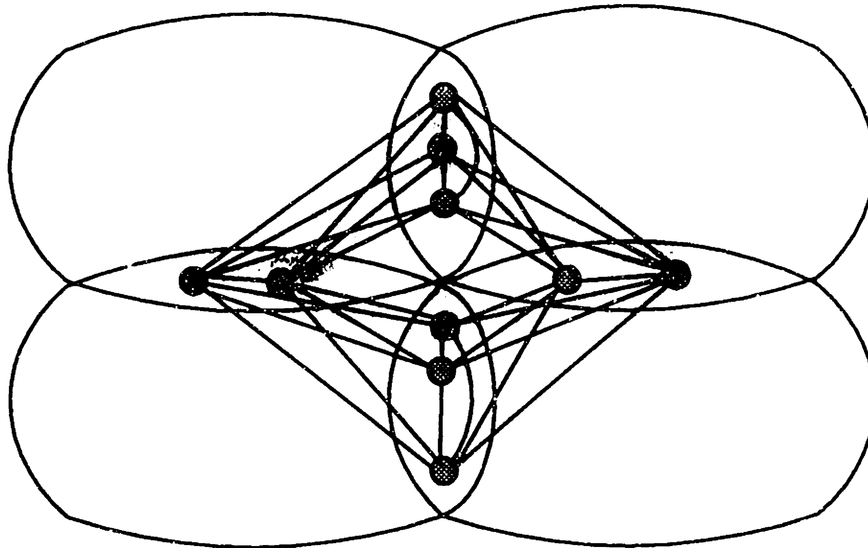


Figure 2.11: Cutset-to-cutset virtual link network in Step 2

In this step we mark all links, whose costs were updated. If the cost of a virtual link is updated, it means that the shortest path between the two nodes goes through another subnetwork. Figure 2.12 explains this situation. As it is shown on the figure the travel time on the shortest path (sp_1) in subnetwork s_1, between cutset nodes c_1 and c_2, at time t_0, is 15 minutes. This travel time was calculated in step 1 and the t_0 entry of virtual link (c_1, c_2) was set to 15. In step 2, however, we found that the travel time on the shortest path (sp_2) is 13 minutes and loops out of s_1. Hence, the link (c_1, c_2) becomes marked and the new value of 13 is updated on the t_0 entry of the virtual link. Now the real shortest path is embedded in the subnetwork s_1.

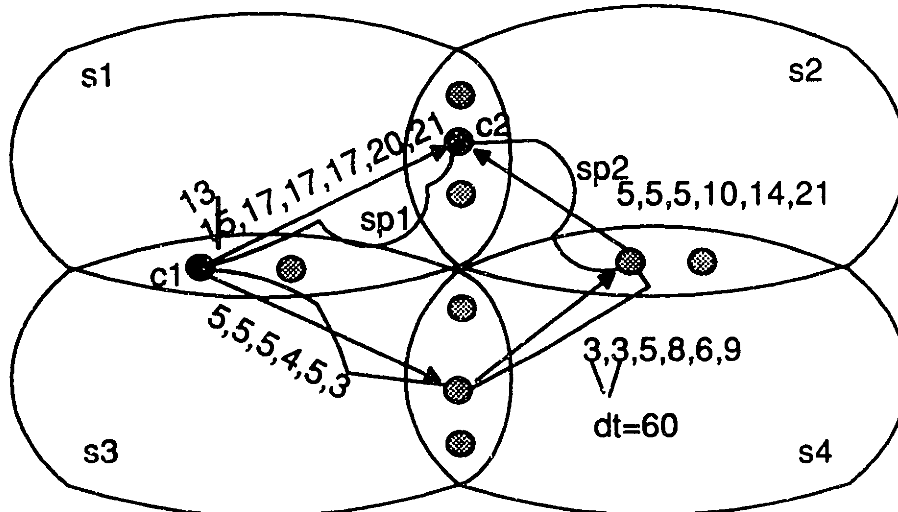


Figure 2.12: Update of travel times in cutset-to-cutset virtual links

In step 3 we calculate the shortest paths between all nodes of the same subnetwork. Each subnetwork consists of the original nodes and links of that subnetwork and all corresponding cutset-to-cutset virtual links that were marked in step 2 (see for example Figure 2.13). The virtual links are added in order to represent that part of the shortest path that loops around to another subnetwork. Note, that if the shortest path between node i and j does not loop out, then we do not need to add virtual link (i,j) to the network. This observation reduces the average network size and improves the overall performance of the algorithm.

In our sample network in Figure 2.10, the shortest path from node *1* to node *2* leaving at t_0, goes through nodes *3* and *4*. The shortest path between the cutset nodes *3* and *4* goes through the other subnetwork, but since the virtual link (*3*,*4*) is part of the left-hand-side network, that loop is represented in the virtual link and embedded in the left-hand-side network. Therefore, the shortest path calculation between nodes *1* and *2* will result in the correct path (*1*-*3*-*4*-*2* as opposed to *1*-*2*). Note, that at this point the algorithm suggests that a vehicle leaving node *1* at t_0 should travel to node *3* in order to be on the shortest path. It is the responsibility of the right-hand-side calculations to guide the vehicle to the shortest path between nodes *3* and *4*. As soon as the vehicle arrives at node *4* the shortest path results suggest that the vehicle should take the path *4*-*2*.

In this step, as a result of the shortest path calculations, we obtain the costs associated with the node-to-cutset virtual links. Note, that these links have only costs corresponding to the t_0 time interval.

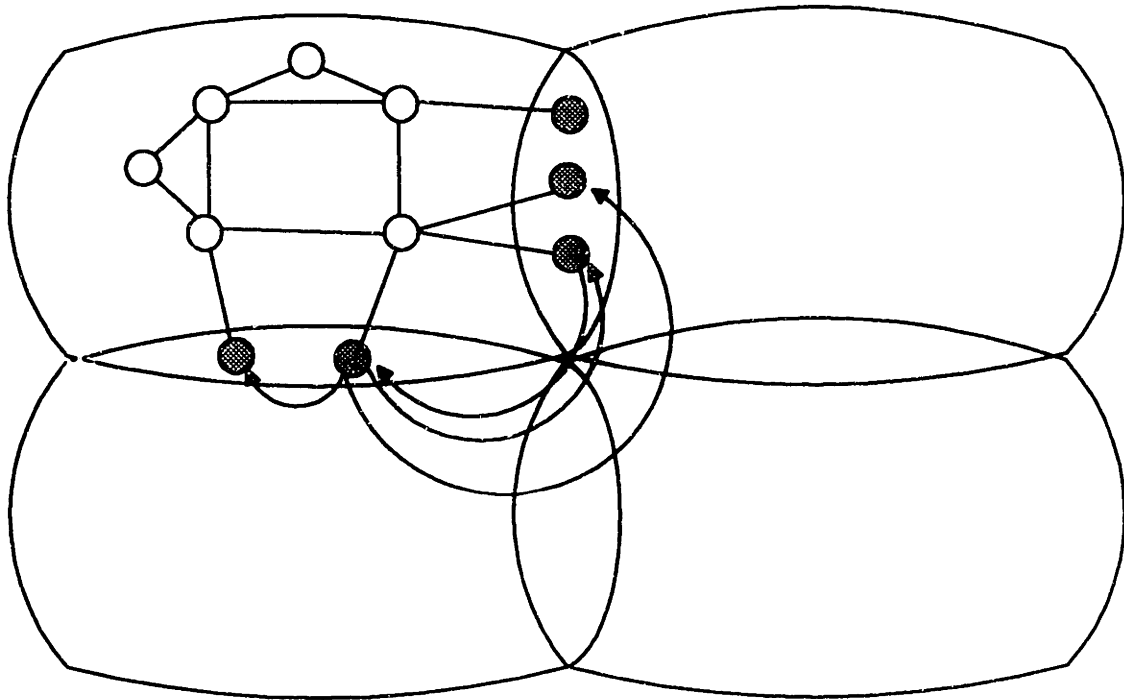


Figure 2.13: In Step 3 only marked cutset-to-cutset virtual links are included

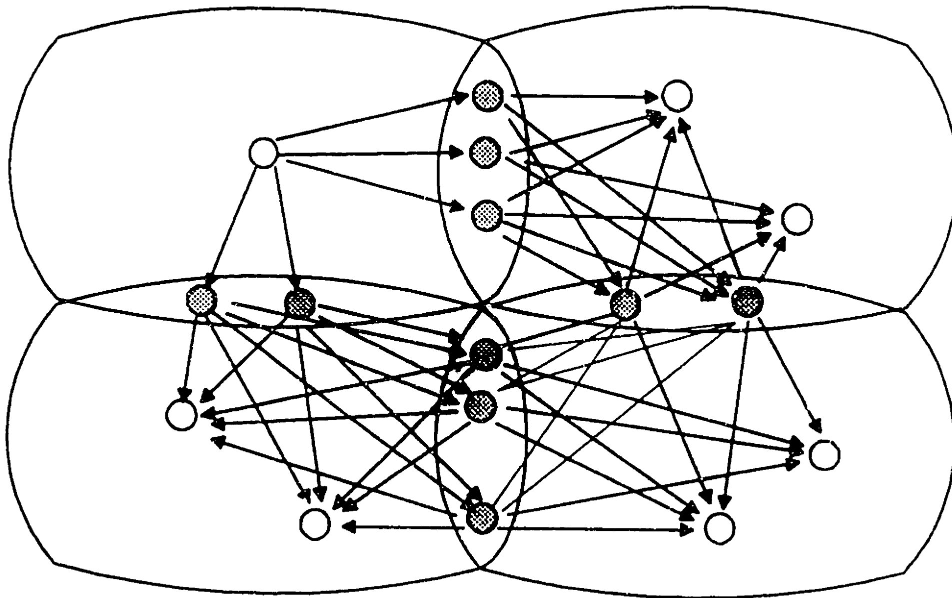
In step 4 we calculate the shortest paths from all nodes to all destination nodes that are not part of the same subnetwork. The calculation is based only on virtual links. See for example Figure 2.10. The shortest path from node *1* to node *7* is the virtual path *1-3-7*. This is a high level routing, which indicates the cutset nodes that the shortest path has to go through. How to travel on the shortest path between the nodes and the cutset nodes is the responsibility of the lower level routing based on the shortest path calculations in the previous steps.

There are two ways to implement this step for a network divided into S subnetworks. We will demonstrate the ideas with the grid network in Figure 2.14, which is divided into 4 subnetworks.

a) At each iteration we calculate the shortest path from a source node, s, to all destination nodes, d, such that s and d are not in the same subnetwork. The network consists of the node-to-cutset virtual links connecting s with its cutset nodes, the cutset-to-cutset virtual links plus the cutset-to-destination virtual links that are part of the other subnetworks (see Figure 2.14a).

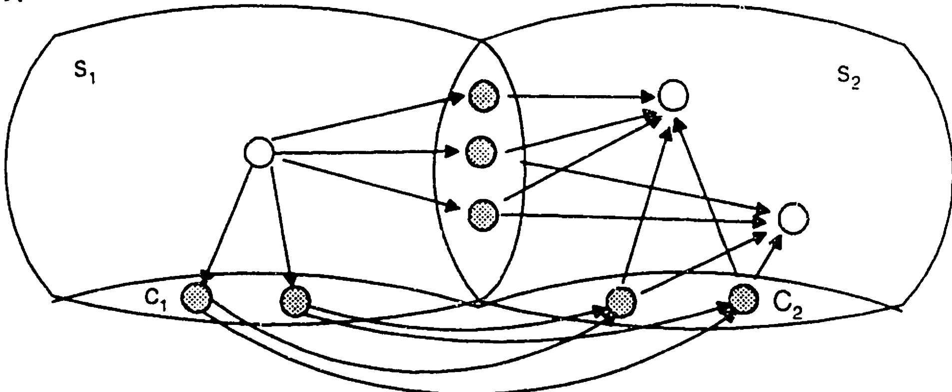
b) Each subnetwork pair is examined separately. Depending on the relative position of the subnetworks the virtual link representation may differ (type1 and type2 in Figure 4.14b). There are virtual links between all cutsets of the origin subnetwork to all cutsets of the destination subnetwork. The costs of these virtual links have been

obtained in step 2. For instance, link (c_1, c_2) should not included in the type 1 network if the shortest path between these cutset nodes goes entirely through subnetworks s_1 and s_2. In this case the node-to-cutset virtual links $\in s_i$ and the cutset-to-destination virtual links $\in s_2$ will have all the information required. These links can be easily identified and marked in step 2. Using this strategy the network size can be reduced substantially. This also may reduce the computational time required for this step (this strategy was not yet implemented in the algorithm whose experiments will be discussed later). Note that the network in this case is acyclic and therefore by topological ordering of nodes shortest paths can be calculated very efficiently, (for detailed discussion of the algorithm refer to [AMOr93]).

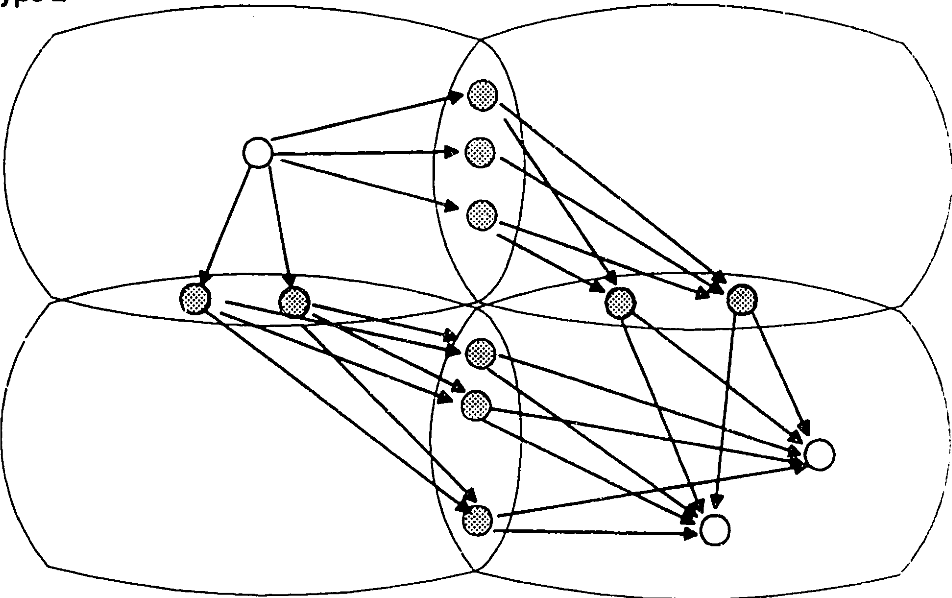


a)

type 1



type 2



b)

Figure 2.14: Two Implementations of Step 4

2.3.2.3 Implementation

In the different steps of the DDSP algorithm we use the dequeue implementation of the label correcting algorithm and the binary heap implementation of Dijkstra's algorithm. These algorithms and their data structures were discussed in section 2.2. In this section we discuss the implementation of the overall algorithm.

In order to reduce the storage requirements and improve the efficiency of the various operations we store the networks in a forward star representation [AMOr93]. In this representation we use two arrays; the "first" and the "end" arrays, where *first[i]* shows where the first link emanating from node i is stored in the end array, *end[k]* shows the node number at the end of link k. The corresponding costs (travel times) are stored in array *costs[][]*, which is a matrix of size *number_of_links*number_of_cost_intervals*. Figure 2.15 illustrates the data organization. In our network representation we have four types of links (original, cutset-to-cutset, node-to-cutset, and cutset-to-cutset). For the different steps of the algorithm we build the actual *first[]* and *end[]* arrays and link them to the corresponding link costs.

In step 3 we add the cutset-to-cutset links to the network only if that link is marked as we discussed earlier under step 2. We build the required network by maintaining an additional array corresponding to each cutset-to-cutset virtual link and mark it if it is part of the network. When the shortest path algorithm is used we need to check whether the link under consideration is marked. If it is not marked we ignore it, otherwise we use it in the calculations. Alternatively we can build dynamically a new network after step 2, based on the marked links, and solve the shortest path problem on the newly constructed network.

In step 4, at each step we solve sequentially the shortest path from each node in the subnetwork to all destination nodes that are not part of the same subnetwork. Note that, in this case, after each step only the source node and the cost of the links that link the source to its cutsets change. Note also, that for all source nodes the number of these links is constant. Therefore, for step 4, we build a generic network. To keep the network size small, the source node is represented by a dummy node and its corresponding node-to-cutset virtual links. At each iteration of the algorithm, i.e., calculation of the shortest paths from different source nodes s, we set the *end[]* pointers of the dummy node to the cost-arrays of the node-to-cutset virtual links that corresponds to the actual s source node.

This is shown in Figure 2.15. This implementation results in small network size and leads to better running time.

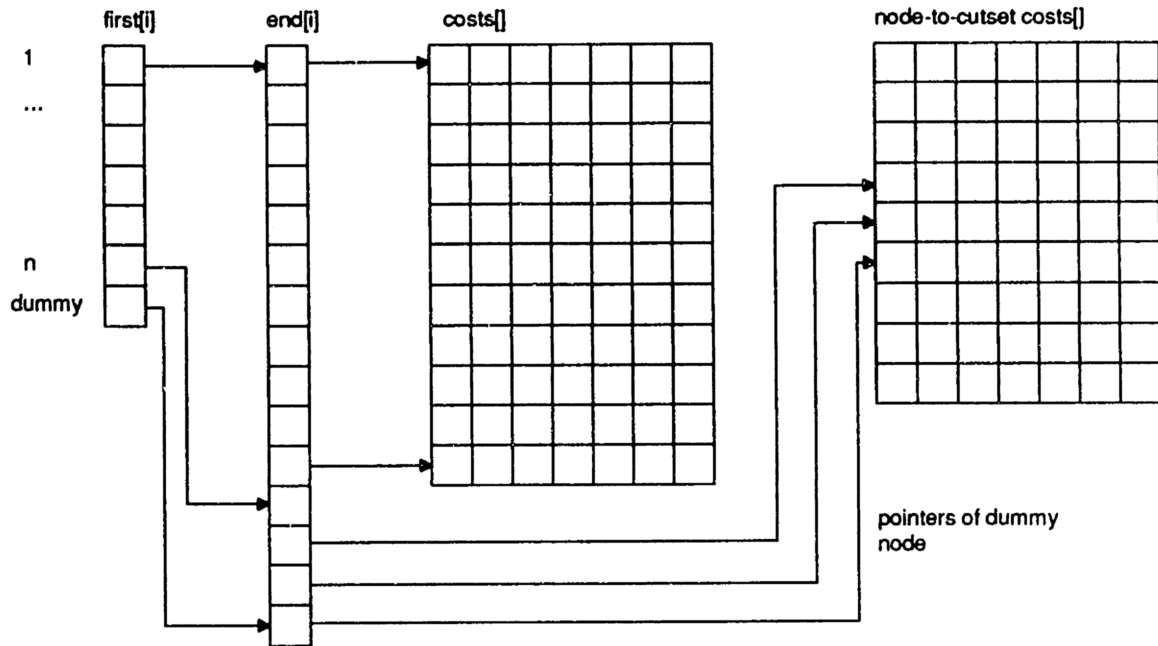


Figure 2.15: Forward star representation of the network in Step 4

The various elements of the decomposition approach were coded using the object oriented programming paradigm and the C++ programming language. The different subnetwork modules and the associated tasks performed fit naturally into the philosophy of object oriented languages. Nevertheless, for the core part of the algorithm, i.e., the dynamic shortest part calculation, we used the low level features of the C language.

2.3.2.4 Issue of Correctness

Comparing the results obtained by the serial and the distributed algorithms we observed that it is possible that some of the shortest paths obtained from the DDSP algorithm differ to a certain degree from the results calculated by the standard algorithms. In our experimental work we found that these inconsistencies are very few.

The problem is caused by the aggregate characteristics of the virtual links. Let us take a look at Figure 2.16. Let us assume that travel times on the links are given every minute and that we want to calculate the shortest paths from node *1* every 5 seconds. The figure shows for each link and for the virtual link (*1*,*4*) the corresponding cost values. Note, that

the cost values of the virtual link correspond to the travel times from node *1* to node *4* departing at t=0 and t=60. In order to get cost values for time instances between the minute interval we linearly interpolate between the discrete values. We examine three cases: in case a) and case b) the costs on links are monotone decreasing and monotone nondecreasing respectively, and in case three the costs are moving in different directions in each link segment. Tabie 2.2 shows the calculated travel times for the three cases. The first column represents the departure time from node *1*. In the columns that follow for each case we report: *label-a* the travel time calculations based on actual links (i.e., finding travel times by interpolating on each link), *label-b* the travel time calculations based on the virtual link (i.e., interpolating between cost values associated with times t=0 and t=60, i.e., interpolating between shortest path values), and *diff* the difference of these travel times .

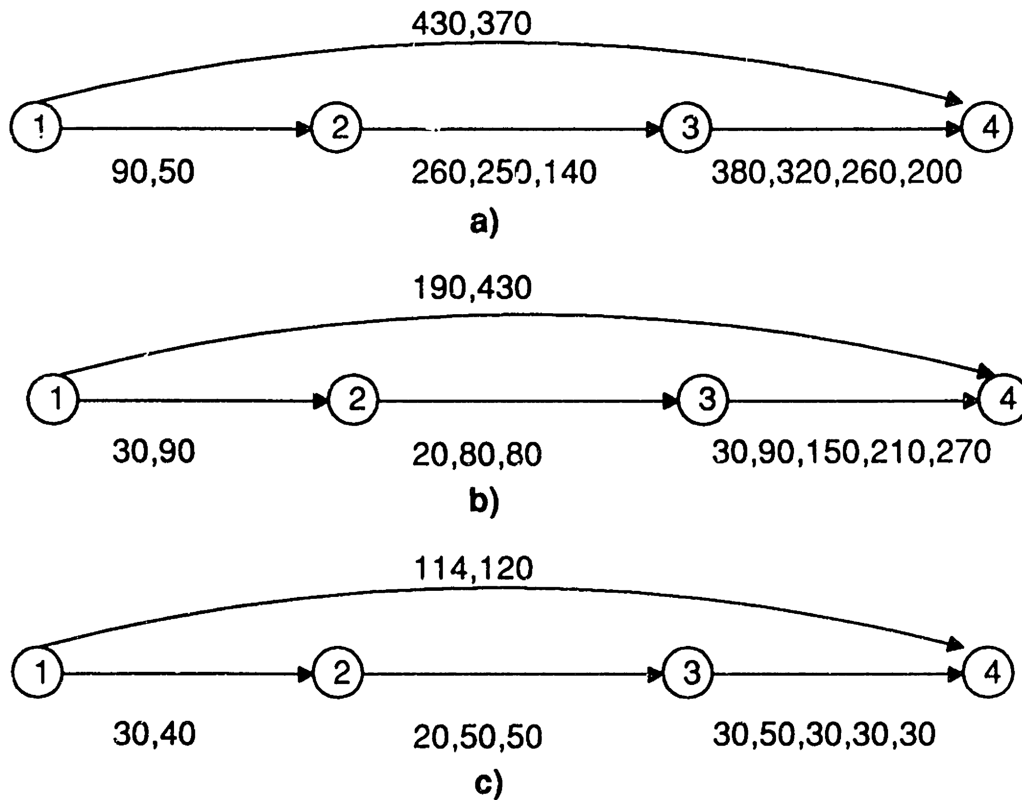


Figure 2.16: Three networks for demonstrating the correctness issue

The results show that in cases of increasing costs and costs moving in different directions the travel time calculation on virtual links may give different results. There is no difference when the costs are monotone nonincreasing.

	a)			b)			c)		
time	label-a	label-b	diff.	label-a	label-b	diff.	label-a	label-b	diff.
0	430	430	0	190	190	0	114	114	0
5	425	425	0	225	210	-15	113	114	1
10	420	420	0	260	230	-30	114	115	1
15	415	415	0	295	250	-45	115	115	0
20	410	410	0	310	270	-40	116	116	0
25	405	405	0	325	290	-35	117	116	-1
30	400	400	0	340	310	-30	117	117	0
35	395	395	0	355	330	-25	115	117	2
40	390	390	0	370	350	-20	116	118	2
45	385	385	0	385	370	-15	117	118	1
50	380	380	0	400	390	-10	118	119	1
55	375	375	0	415	410	-5	119	119	0
60	370	370	0	430	430	0	120	120	0

Table 2.2: Calculated travel times and their differences for three network types

Results with real networks also indicate the differences in travel times are small. Experiments with the Sudbury network showed that the DDSP algorithm solves the shortest path problem with a 99.1% correctness (i.e., 99.1% of the shortest paths are correctly determined). Furthermore, there was only a few seconds difference in the travel times along the different paths. The average difference was 2.59 seconds (compared with a 209.49 seconds average travel time in the network). The maximum difference was 10 seconds. It appeared only once and corresponded to an origin destination pair with 186 seconds travel time. As we mentioned already the running time of the algorithm improves if, in step 1 and 2, we do not calculate shortest paths for all time intervals but only for every second or third one and we obtain the missing values by linear interpolation. We analyzed the correctness of the algorithm for these cases as well. The maximum difference when shortest paths are calculated every second time interval was 17 seconds. This appeared only for one OD pair. The shortest paths were correct 98.5% of the time.

When virtual link costs were calculated every third time interval, the maximum difference was 20 seconds and it occurred with two OD pairs. The correctness of the DDSP algorithm in this case was 98%.

The observed small differences are due to the fact that in transportation networks travel times do not change dramatically in short time periods (as in case b).) If there are no accidents in the network costs change with small slopes resulting in little or no differences. Note also that if the differences are small then it is unlikely that the physical paths that correspond to the two cases differ (only the corresponding travel time).

Chapter 3 Distributed Dynamic Shortest Path Algorithm Experimental Results

In this chapter we report the experimental results with the Distributed Dynamic Shortest Path (DDSP) algorithm, and introduce some of its features. We also show that if the number of cutset nodes is small, then the running time of the algorithm improves significantly. Eliminating some cutset nodes from the network may be an acceptable practice given the nature of urban networks. In many large urban networks the decomposition in subnetworks is driven by natural boundaries, like rivers, and the number of cutset nodes is automatically reduced to the number of bridges along the river. The DDSP algorithm is especially suited for such cases.

3.1 Distributed Algorithm Experimental Results

In this section we present the computational results obtained using the DDSP algorithm for different networks. For each case we run the algorithm to find the shortest path from all nodes to all destination nodes leaving the source nodes at time t_0. The algorithms were tested on a real network (Sudbury network, 205 nodes and 578 links) and three randomly generated standard square grid networks with 40x40, 70x70 (simulating a urban network with size similar to Boston: 4922 nodes) and 100x100 nodes.

3.1.1 Single Processor Implementation

The serial implementation of the algorithm is faster than the traditional implementation only when the number of destination nodes is small (≤1% of total nodes). On a 70x70 square grid network for example, the running time using the dequeue algorithm was 1194.3 seconds. Using the DDSP algorithm (on one processor) the running time dropped to 875 seconds (4 destination nodes). However, when the destination nodes represent a larger percentage of the nodes the traditional modified shortest path algorithm runs faster than the distributed one (with the current implementation).

3.1.2 Distributed Implementation

We tested the algorithm on the Sudbury network, and on various grid networks using 4 workstations. The networks were divided into 4 subnetworks and each subnetwork was assigned to a processor. As was noted before, and the results here support it, the number of destination nodes strongly influences the performance of the algorithm. The computational results, run on SUN-4 workstations, are shown in Table 3.1.

	destination nodes	**Serial Dequeue Algorithm**	**DDSP Algorithm**
Sudbury Network (205 nodes 578 links)	100%	2.3 sec	3.9 sec
Grid Network 40x40 (1600 nodes 6240 links)	100%	157.0 sec	94.0 sec
Grid Network 70x70 (4900 nodes 19320 links)	8.3% 5.0%	1194.3 sec 1194.3 sec	482.6 sec 387.8 sec
Grid Network 100x100 (10000 nodes 39600 links)	8.3% 5.0%	4876.1 sec 4876.1 sec	2305.8 sec 1799.4 sec

Table 3.1: Computational results for the serial and the DDSP algorithm

We should point out that the DDSP produces better results only with large network sizes. This was expected, since the running time of the shortest path algorithm increases more than linearly with n, the number of nodes in the network. The serial algorithm is superior to the distributed one for small networks (Sudbury network) because of the relatively longer set-up times. This changes however, as we move towards larger networks. For large networks the DDSP algorithm outperforms the serial algorithms (utilizing the computational power of the four processors). The results also show that the running time depends strongly on the number of destination nodes in the network. As we can see in Table 3.2, for the 70x70 grid network, if 8.3% of the nodes are destination nodes, the running time of step 4 is 268.8 seconds, while if 5% of nodes are destinations the running time becomes 174.0 seconds (for comparison the Boston network (4922 nodes) has 239 destination nodes, or 4.9% of total nodes). These running times result in a total of 482.6 seconds computational time for the former case and 387.8 seconds for the

later, as opposed to the 1194.3 seconds running time of the serial algorithm. We can observe similar trends for the 100x100 network as well.

	Grid Network 70x70		Grid Network 100x100	
	network size in each proc. module	running time	network size in each proc. module	running time
step 1	1225 nodes 4760 links	45.0 sec	2500 nodes 9800 links	147.0 sec
step 2	139 nodes 13024 links	44.0 sec	199 nodes 26854 links	114.0 sec
step 3	1225 nodes 4979 links	124.8 sec	2500 nodes 10244 links	547.8 sec
step 4 (8.3%dest. node)	446 nodes 25908 links	268.8 sec	825 nodes 71488 links	1497.0 sec
step 4 (5.0%dest. node)	323 nodes 17518 links	174.0 sec	575 nodes 46852 links	990.6 sec

Table 3.2: Computational results for each step of the DDSP algorithm

Table 3.2 also shows that steps 3 and 4 are the most time-consuming parts of the algorithm. Steps 1 and 2 are relatively fast because of the small number of shortest paths to be solved (step 1) and the good utilization of the distributed computation efforts. For instance in the 70x70 grid network the number of nodes in step 1 is 1225 and the number of links is 4760. At this step we find the shortest paths from the 70 cutset nodes for all *dt* time intervals. In step 2 we obtain shortest paths in the cutset-to-cutset virtual link network. The number of nodes in the network is 139 and the number of links is 13024. In step 3 we solve the shortest path trees from all the 1225 nodes in each subnetwork. The computational time increases due to the high arc degrees at cutset nodes. Note that, if we did not use the virtual link marking technique discussed in 2.4.2 then from each cutset node 70 links would emanate. Using the marking technique the number of links was reduced to about 4979 (instead of 9184 links). Since many of the cutset nodes are of high degree the label setting implementation of the dynamic shortest path algorithm was used. In step 4 the network becomes dense resulting in long computation time. In this case, surprisingly, the label correcting algorithm performs better. Note, that step 4 was implemented using the alternative shown in Figure 2.14a. We believe though, that the other alternative (Figure 2.14b) may result in running time improvements (if combined with topological sorting).

In our results the costs on each virtual link were calculated for ten time intervals. Note, that the number of time intervals used affects the running times as well. For instance, if we calculate the costs for every second time intervals only, and we derive the missing results by interpolation, then the computational time for step 1 and 2 will be almost halved. This gain in computational time, however, might result in erroneous identification of certain shortest paths.

Note, that if we use the four processors in a simple distributed architecture (i.e., where in each processor module the whole original network is represented and each module calculates shortest paths from one fourth of the nodes), then the resulting running time is better than the running times of the DDSP architecture. The running time of this approach results in almost linear speed-up, hence it is approximately equal to one fourth of the running time of the serial method (i.e., 1194.3/4=298.6 seconds). However, as we will show in the next sections, in certain types of transportation networks or by using reasonable heuristics the DDSP outperforms this simple distributed architecture.

3.2 Effect of Cutset Nodes on DDSP Algorithms Performance

The number of cutset nodes in the network significantly influences the running time of the DDSP algorithm. In the previous section we have already discussed that the number of destination nodes influences the running time. The question addressed here is whether the number of cutset nodes has similar effects.

In three out of the four steps of the DDSP algorithm the cutset nodes have direct influence on the running time. In step 1 we calculate shortest paths from cutset nodes to all other nodes in each subnetwork. If the number of cutset nodes is reduced, it results in linear speed up in the calculations. In step 2 the algorithm calculates the shortest paths from all cutset nodes to all other cutset nodes in the network. The network in this step is based on the cutset-to-cutset virtual links (see Figure 2.11). Therefore, the number of nodes and the number of virtual links in the network at this step depend on the number of cutset nodes. In step 3 we calculate shortest paths from all nodes to all nodes of the subnetwork. The number of cutset nodes has indirect effect on the network size, since all cutset-to-cutset virtual links that were marked in step 2 are included in the network. In step 4 we calculate shortest paths based on cutset-to-destination virtual links only. Therefore, in step 4 as well, the size of the network depends on the number of cutset nodes.

The above discussion shows that the number of cutset nodes in many ways affects the performance of the algorithm. Network decomposition then becomes a key issue, since determines the number of cutset nodes. To demonstrate the above argument, take a look at a trivial example shown in Figure 3.1. It is possible to divide the network into two balanced subnetworks in two different ways, (along axes *p1* and *p2*). However, dividing the network along the *p1* line defines less cutset nodes than the division along the *p2* line. Therefore, this decomposition is more advantageous for the DDSP algorithm.

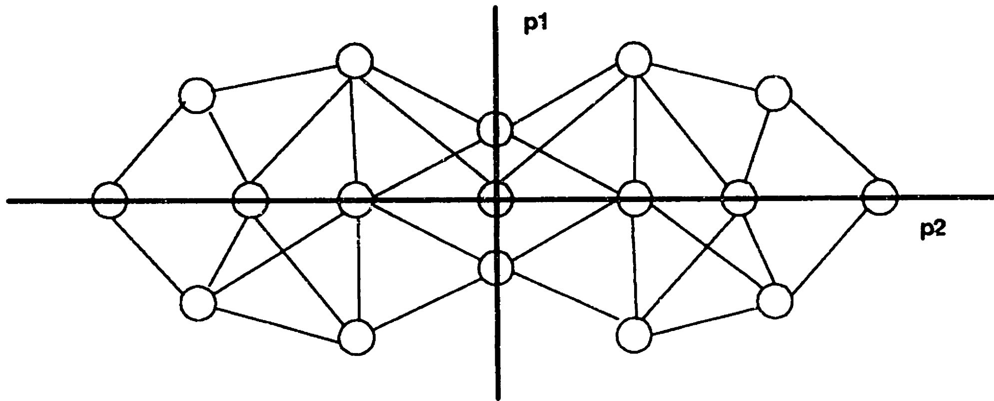


Figure 3.1: Number of cutset nodes and the decomposition

Many of the large cities in the world are divided naturally into subnetworks by rivers and other barriers. This is especially true for the vast majority of European cities (e.g., London, Paris, Budapest, Moscow, Berlin, etc.) but also for many U.S. (e.g., Boston, N.Y. City etc.) and Asian cities (e.g., Tokyo, Bangkok and Sanghai). Consequently natural barriers can be used to guide the network decomposition. In such cases the number of cutset nodes is small since it becomes equal for example, to the number of bridges along the river.

We tested the DDSP algorithm in a grid network, which consists of two 70x35 subnetworks connected by a number of arcs (bridges). Figure 3.2 illustrates the network, which is naturally decomposed into two subnetworks.

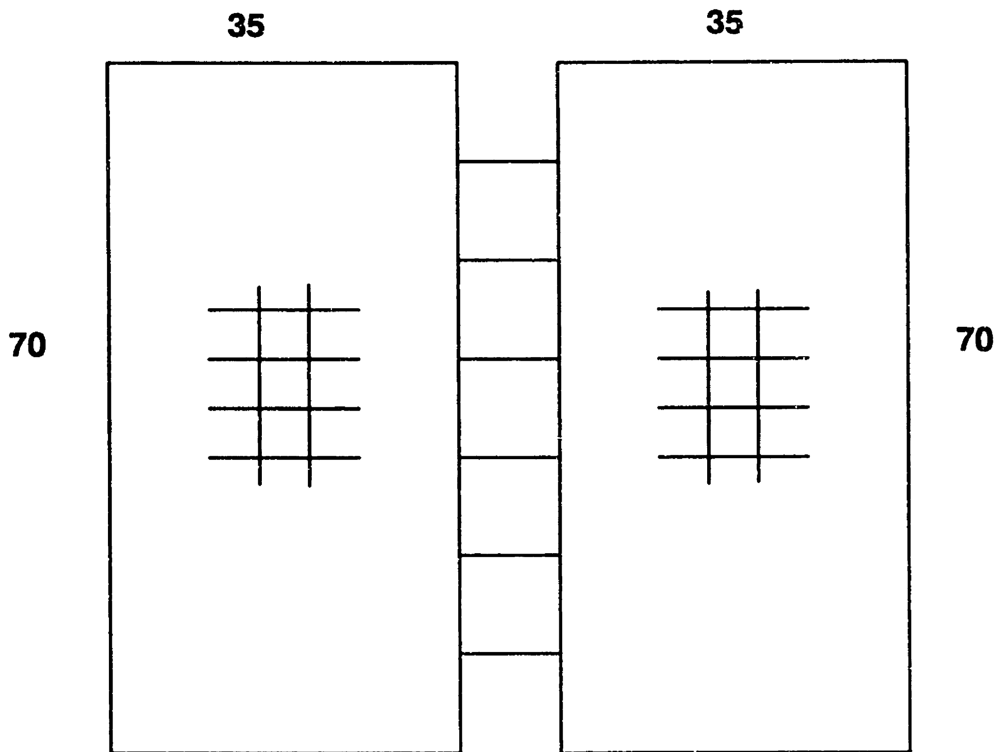


Figure 3.2: Two 70x35 grid networks connected by bridges

We used four processors such that in each subnetwork the computational efforts were shared by two processors (i.e., if n shortest paths were to be solved $n/2$ shortest path trees were calculated by one processor). We tested the algorithm with 18 and 10 bridges and 5% destination nodes. Table 3.3 shows the running time for the two cases. The problem in step 2 becomes very simple to solve. In this case we just pairwise compare the cutset-to-cutset virtual links and mark the nodes for step 3. For this reason the running time of this step is incorporated in the running time of step 1. Note, that the running time of the serial algorithm is not very sensitive to the number of bridges and it is 1142 seconds in the 18 bridge case and 1135 seconds in the 10 bridge case.

	DDSP	DDSP
Number of Bridges	18	10
step 1	12.25 sec	7.28 sec
step 3	161.66 sec	153.96 sec
step 4	34.26 sec	21.67 sec
Total	208.42 sec	182.91 sec

Table 3.3: Detailed results of DDSP algorithm in networks connected by bridges

The results show the DDSP algorithm performs well with networks exhibiting the above structure (natural boundaries with limited access links). The running time improvements are substantial. Calculating shortest paths by the simple distributed architecture in the 18 bridge network, the running time is 285 seconds, while calculating by the DDSP algorithm is 208.42 seconds. The improvement is even better in the 10 bridge network, where the simple distributed architecture runs in 283 seconds, while the DDSP runs in 182.91 seconds. The results show that in networks with limited access links the DDSP algorithm is superior.

The results also suggest that the decomposition strategy in general, should try to identify subnetworks with the minimum number of cutset nodes.

3.3 Heuristics

Given the importance of the number of cutset nodes in determining the running time of the DDSP algorithm we will examine heuristics that ignore some of the cutset nodes in the network. Furthermore we claim that in transportation networks it is possible to ignore cutset nodes and still obtain acceptable results. By ignoring a cutset node c we mean that no virtual link will be incident to or from node c. In practice ignoring node c from the cutset will result in no inter-subnetwork routing through node c. Hence no shortest path between node i and j, such that $i \in s_k$ and $j \notin s_k$, where s_k is a subnetworks of the network, will go through node c (however, for intra-subnetwork routings node c

can be part of shortest paths). As a result the DDSP algorithm may not find the correct shortest path. It finds shortest paths given the reduced set of cutset nodes.

3.3.1 Experimental Results

We tested the DDSP algorithm with reduced set of cutset nodes, on the three grid networks of size 40x40, 70x70 and 100x100 nodes, each divided in 4 subnetworks. For all networks we assumed that 5% of the nodes are destination nodes, as is the case with the Boston network. For each network we tested the algorithm with various size subsets of cutset nodes. For example when 87.5% of the cutset nodes were included every 8th cutset node was ignored (i.e., treated as a normal node in the network). Table 3.4 summarizes the results for different scenarios.

% cutsets included	70x70 Grid	100x100 Grid
100%	387.8 sec	1799.4 sec
87.5%	351.5 sec	1585.4 sec
75%	309.0 sec	1420.0 sec
50%	243.1 sec	1097.2 sec
33%	205.0 sec	926.13 sec
25%	187.3 sec	833.86 sec
12.5%	161.9 sec	716.86 sec

Table 3.4: Running time as a function of the number of cutset nodes

These results clearly show that as the number of the cutset nodes decreases the running time of the algorithm also decreases. Hence, using these heuristics the computational requirements have been reduced. The running time with 50% of the cutset nodes included, for the 70x70 grid network, is 243.1 seconds. This is a 37% improvement compared to the case when all nodes are included (100% case.) The running times for the 50% and 100% cases in the 100x100 network are 1097.2 and 1799.4 seconds respectively, which corresponds to a 39% improvement. For the cases where only every 3rd and 4th cutset node is included (33% and 25%) the running times of the DDSP algorithm, in the 70x70 grid network, are 205.0 and 187.3 seconds respectively. This

corresponds to a 47% and a 51% running time improvement with respect to the case where 100% of the cutset nodes are included in the network.

Table 3.5 shows the running times in detail for each step of the DDSP algorithms. As we argued, the running times in the different steps dropped significantly. The largest reduction was achieved in step 4 where, for example in the 50% case, the reduction is 84.2 seconds in the 70x70 network and 525 seconds in the 100x100 grid network. This is due to the reduced network size in this step. The number of links for example in the 70x70 network when 25% of the cutset nodes are included, is 3618, while in the case when 100% are included, it is 17518. In step 2 for both the 70x70 and 100x100 networks, the new running time represents an 84% improvement.

	70x70 Grid			**100x100 Grid**		
%cutsets included	100%	50%	25%	100%	50%	25%
step 1	45.0 sec	21.3 sec	11.5 sec	47.0 sec	65.1 sec	34.6 sec
step 2	44.0 sec	7.15 sec	1.6 sec	114.0 sec	19.2 sec	4.4 sec
step 3	124.8 sec	124.8 sec	124.8 sec	547.0 sec	547.0 sec	547.0 sec
step 4	174.0 sec	89.8 sec	49.3 sec	990.6 sec	465.6 sec	247.7 sec

Table 3.5: Detailed running times for different networks

The running time for the serial implementation (serial) for the two networks are 1194.3 and 4876.1 seconds respectively. When the 4 processors are used to solve 1/4 of the shortest paths the times become 298.6 and 1219 seconds. The DDSP algorithm outperforms this implementation when 50% or less cutset nodes are included. Note, however, that the grid network of the type generated for this analysis present a worst case scenario in terms of number of cutset nodes. Actual transportation networks will have much less cutset nodes (as will be discussed in the following section).

Summarizing, the DDSP algorithm performs very well in the case of low number of cutset nodes. It takes advantage of the network structure by solving shortest paths only towards the destination nodes and utilizing a small number of cutset nodes.

3.3.2 Discussion

Eliminating some cutset nodes from the network will result in no inter-subnetwork routing through those nodes. At first glance this seems to be a negative consequence. We claim however, that in urban networks this can be acceptable and may even be considered to be a positive aspect.

Assume that the urban network under consideration is a grid network. We have chosen this network type because this is the worst network structure regarding the number of cutset nodes after decomposition. Nevertheless, many cities, especially in the United States, have grid network structures. Note, however that in most cases the streets in such networks are one way streets (e.g., Manhattan). In our network terminology it would mean that most of the cutset nodes will have virtual links only in one direction. The situation is shown in Figure 3.3. Note that if we assume that all streets are one way streets then, from the perspective of a subnetwork, this is equivalent to the case where every second cutset node is ignored. Half of the boundary nodes have links leading out and half of the nodes have links leading in the subnetwork.

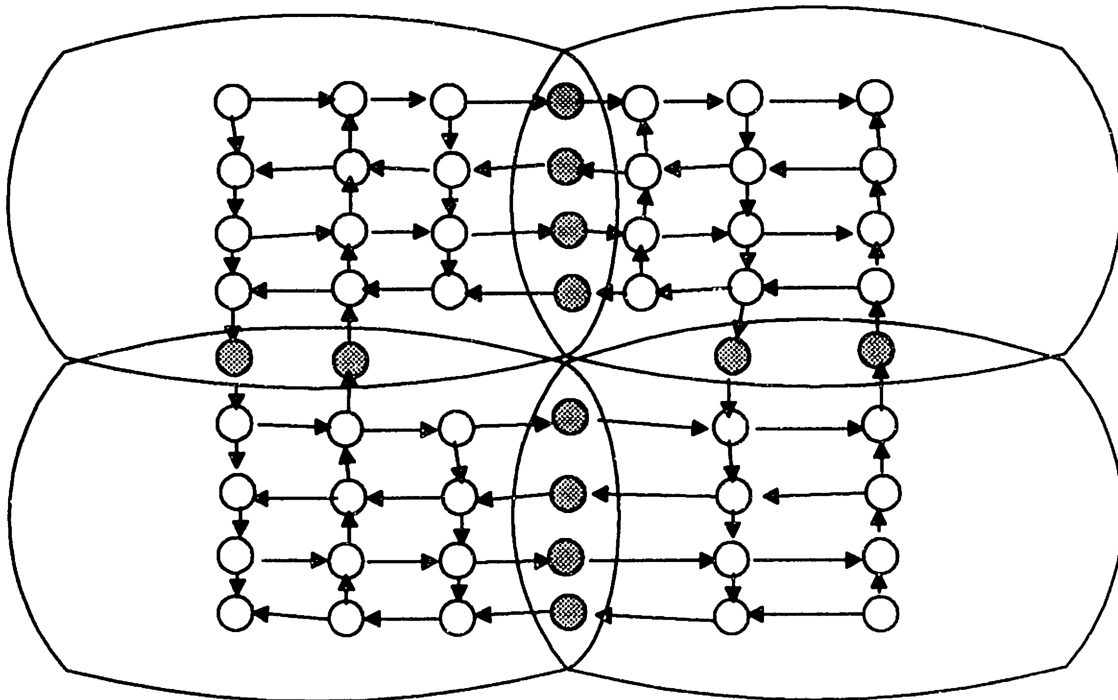
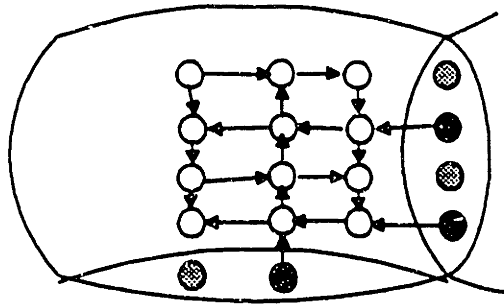


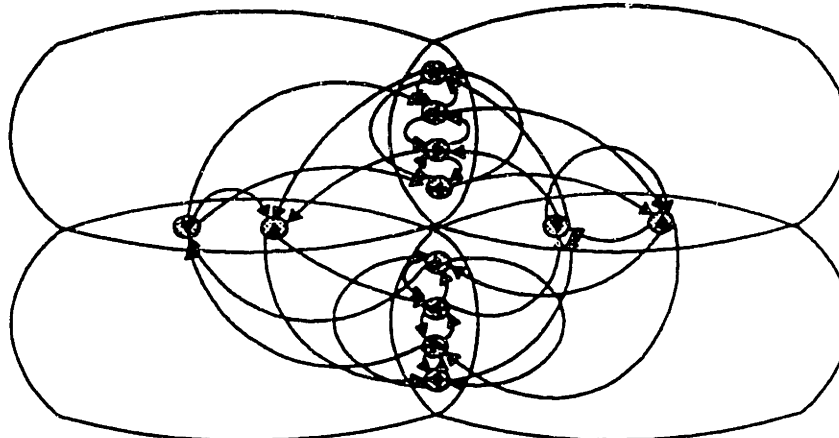
Figure 3.3: Representation of one-way streets in networks

Figure 3.4 shows that with networks exhibiting the structure of Figure 3.3. there are computational benefits at all steps of the DDSP algorithm. At step 1 we need to solve the shortest paths only from the cutset nodes that have links pointing inward the subnetwork. In step 2 the number of cutset-to-cutset virtual links is reduced also. We connect a cutset node i with a cutset node j of a subnetwork by a virtual link only if i is an outbound node and j is an inbound. This constraint reduces the number of links in the network by a factor of 2. Consequently in step 3 the maximum number of the marked and included cutset-to-cutset virtual links in each subnetwork is also halved. We include node-to-cutset virtual links only between the original nodes and the cutset nodes that have outward arcs. Similarly cutset-to-destination virtual links are only defined between the cutset nodes having inward arcs and the destination nodes. Hence, in step 4, when we calculate shortest paths from each node of a subnetwork to all destination nodes in other subnetwork, only half of the nodes and arcs are represented.

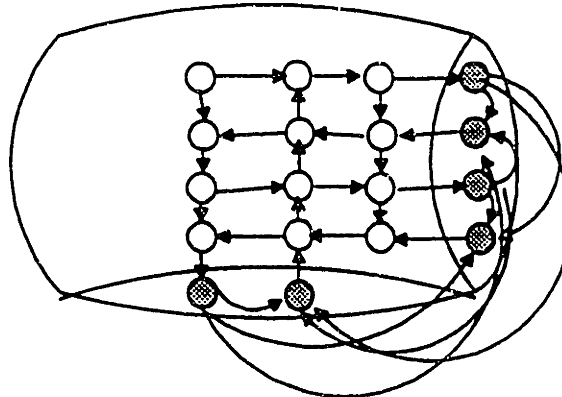
Step 1



Step 2



Step 3



Step 4

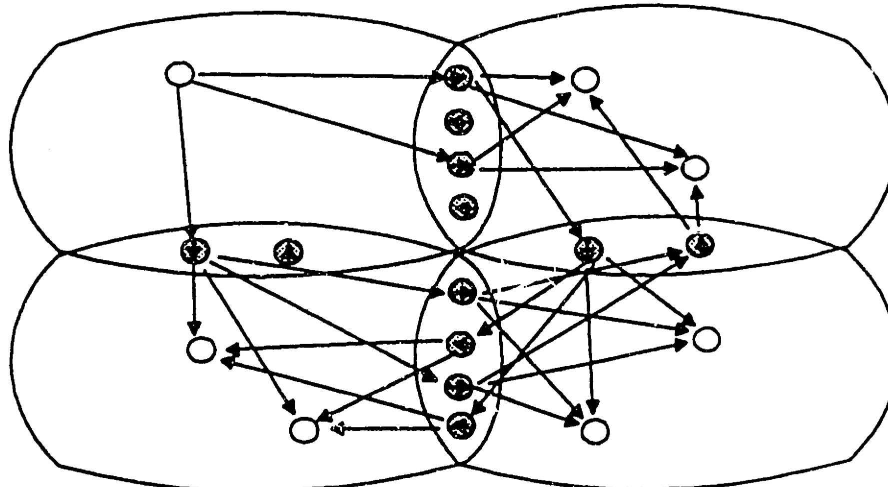


Figure 3.4: Network configurations for each step in case of one way streets

However, even if the grid network does not have a large number of one-way streets, social, political and other considerations, may dictate that only a small subset of the actual cutset can be used effectively. Consider for example the network in Figure 3.5, which consists of main roads, arteries side streets and residential roads with low capacities. In many cases the small streets only serve local traffic and they cannot be used for routing decisions. The goal of a fair transportation management and information system can not be routing inter-subnetwork traffic flows through these streets. The inter-subnetwork traffic (traffic between different parts of the city) should be routed through the main roads and the main arteries of the urban network. This suggests that in urban networks it is acceptable (and desirable) to ignore cutset nodes that correspond to side streets or residential area streets.

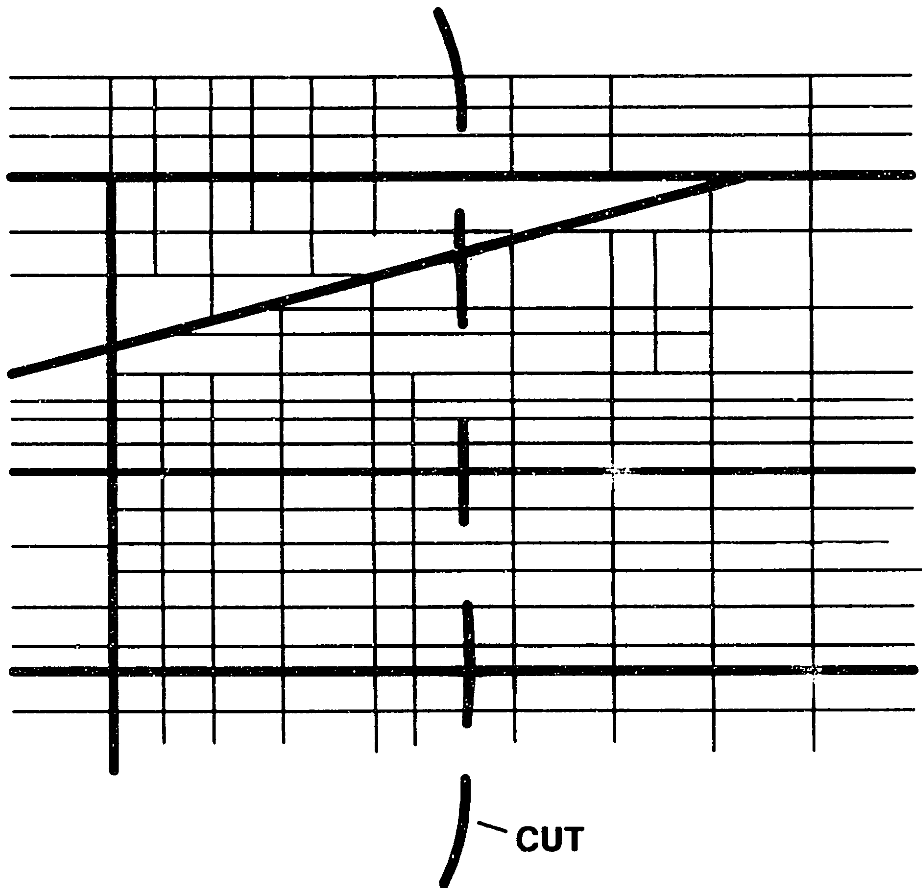


Figure 3.5: Urban network decomposed along main and side streets

In some ATIS demonstration projects (like LISB in Berlin, Germany [Hoff91]) this problem was solved by not representing all streets in the network at all. The heuristic that

we assume here however is better, since it does not ignore any link segments or nodes from the subnetwork calculations (hence shortest path from all those nodes to all other nodes is possible), and, at the same time prohibits inter-subnetwork traffic through these nodes.

The above discussion implies that for urban transportation networks because of sociopolitical, environmental and traffic engineering reasons it is acceptable and sometimes even advisable to prevent some streets (cutset nodes) from being used. Therefore the assumptions behind the DDSP heuristics are reasonable.

Chapter 4 Routing Tables

Once time-dependent shortest paths have been determined and proper control strategies have been generated this information should be properly stored and transmitted to the drivers. The basis for the construction of routing tables is the shortest path trees generated in the shortest path calculations. In general routing tables provide the information which node (or link) should the driver visit (or traverse) next in order to travel on the shortest path towards its destination. The type and amount of information vary depending on the different architecture and implementations. In this chapter we analyze the special characteristics and problems of constructing the time dependent routing tables and suggest different algorithms and approaches to construct these tables. For the purpose of this discussion we assume that the system operates in a way similar to the one described in Chapter 1.

4.1 Dynamic Routing Tables

Dynamic Routing Tables are complex because of their time dimension. Shortest paths may change over time, thus routing tables should have the capability to provide time-dependent information. In other words routing tables should be able to give routing information from each node to every destination for all instances in time. Hence (between two information-updates), at each node the dynamic routing table should be represented by a matrix, where each column corresponds to a destination and each row to a time interval. Each entry holds information on which node should be visited next in order to travel along the shortest path. The information is thus dependent on the **destination node** (column) and the **time** the vehicle arrives at the node. The size of the matrix is specified by the number of destination nodes in the network and the number of time intervals included in the matrix. Figure 4.1 shows a time dependent dynamic routing table.

time	1	2	3	4	5	6	7	8	9	10	11	n
t_0	2	1	2	2	2	4	1	1	4	4	4	4
t_0+dt	2	1	2	1	1	4	1	4	4	4	1	4
t_0+2dt	2	1	2	1	1	4	1	4	4	4	1	4
...												
t_h	2	1	2	1	1	4	1	4	4	4	1	4

(Columns 1–n: destinations)

Figure 4.1: Time-dependent dynamic routing table

The information required to complete a routing table may be obtained by solving the shortest path problem for each *dt* time interval, where *dt* corresponds to the size of the time interval. However, the computation time needed to solve the SP problem for each *dt* interval, especially for large networks, may exceed the length of the planed update time. Information is available at t_0 and $t_h = t_0 + t_{update}$ time intervals. Therefore, in order to satisfy the real-time requirements of the systems, different approaches have to be examined and developed in order to reduce the computational requirements (in addition to developing fast dynamic shortest path algorithms).

One possible approach consists of calculating the time dependent shortest paths only for each t_{update} interval, assuming that dynamic shortest paths do not change significantly during this time. Hence, it would be enough to calculate shortest paths only at $l*(t_{update})$ instances, where $l=0,1,2,...$. This approach overcomes the problems of long computation times but it ignores any variability in travel times that may exist between updates.

In what follows we will discuss approaches that assume that shortest path calculations take place every t_{update} intervals only. We will describe the problems arising and introduce two approaches, the Postorder Numbering Approach and the Rolling Horizon Approach, that we have developed for the construction of routing tables.

4.2 Construction of Routing Tables

4.2.1 A Special Characteristic of Dynamic Routing Tables

Dynamic networks differ from the static ones by having one more dimension in their representation. This extra dimension makes not only the shortest path problem but also the routing table construction problem more complex than it is for the static case.

On the basis of the observations mentioned above, the simplest and most straightforward approach would be, if at each node we would provide an array of routing information updated at each t_{update} interval. This approach leads to the correct solution of the problem in the case of static networks. However, as we will show, this simple approach cannot be used in the case of time-dependent networks.

Let *next_node[i]* indicate the next node on the shortest path to destination *i*. See for example Figure 4.2. At the origin the value of *next_node[6]* is *2* indicating that the next node to visit is *2*. At node *2* we obtain the value of *next_node[6]* stored there and we proceed following the recommendations until we arrive, on the shortest path, to node *6*.

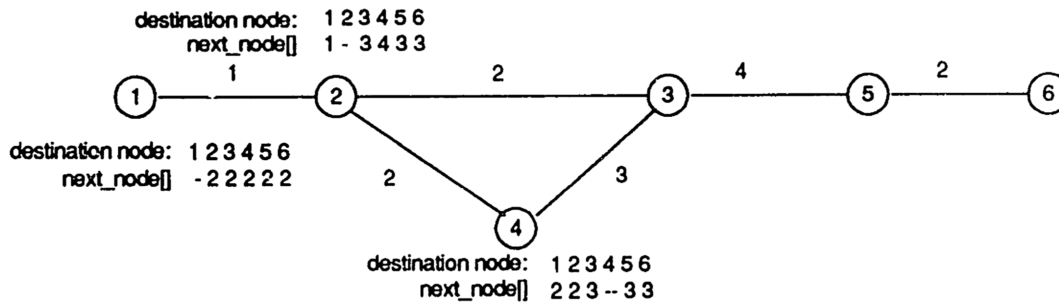


Figure 4.2: Sample network - static case

The above approach is correct for static networks. Let us now examine what happens if the network is dynamic, which is the case in most IVHS applications. Let us consider the same network but with time-dependent link costs (see Figure 4.3). Following the approach discussed earlier we solve the SP problem from every node to all destinations for time t_0. Then for every node we have the *next_node[]* vector based on its shortest path solution. The information contained in each entry though, is correct only if the vehicle departs the corresponding node at time t_0. However, the vehicle departing node *1* at time t_0 will arrive at node *i* at time t_0+tt, where *tt* is the travel time from *1* to *i*.

Consequently, the information provided at node i is not consistent with the arrival time of the vehicle (whose origin is another node).

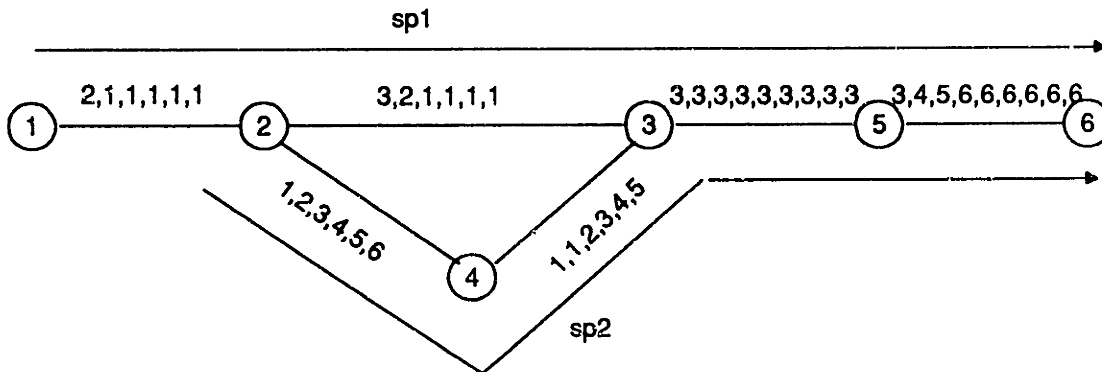


Figure 4.3: Sample network - dynamic case

If for example we calculate the shortest path from node *1* to node *6* at t_0, the path *sp1*, goes through nodes *1-2-3-5-6*. However, if we calculate the shortest path from node 2 at t_0 the path *sp2*, goes through nodes *2-4-3-5-6* (note that if we had calculated the shortest path from node 2 at a time equal to the sum of t_0 and the travel time between nodes *1* and *2*, this path would be the same as *sp1*). Consequently, if a vehicle departs from node *1* at t_0, when it arrives at node 2 the following routing information will be provided (based on *sp2*, the shortest path from 2 to 6 at time t_0)

1	2	3	4	5	6	7
			4	5	5	5

instead of the correct (based on *sp1*)

1	2	3	4	5	6	7
			4	5	4	4

Correct treatment of the problem, under the assumption that SPs are calculated every t_{update} interval, would require that at each node we store as many arrays as origin nodes. This naive approach would require at each node a matrix of size *number_of_origin*number_of_destination*. In case of large networks this naive method would require a large amount of information to be stored. Furthermore, this method

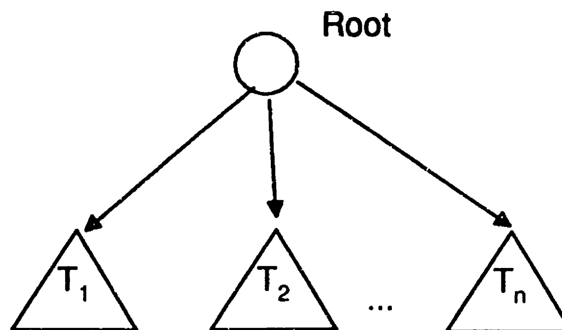
stores information in a redundant and inefficient way that, as we will show, can be avoided.

One way to overcome this problem would be to store the shortest path tree only at the origin node and transmit it to the vehicle at departure time. This method however prevents the vehicle from obtaining updated information while it is en-route. The way to overcome this inadequacy is again to store and to transmit all shortest path trees to the vehicles at every node (since we do not know its origin node). But this is what we want to avoid. Note, that the amount of information to be transmitted becomes very large.

4.3 Routing with Postorder Numbering

In this section we present an implementation of routing tables that minimize the amount of storage and information transmission that is required. It is based on an approach that uses postorder numbering.

Postorder numbering results from numbering the nodes according to the order in which they are visited in postorder traversal. For the general tree shown below,



Postorder traversal is defined by the following recursive function:

```
Postorder(Root)
{
 Postorder(T1);
 Postorder(T2);
 ...
 Postorder(Tn);
 visit Root;
}
```

Before introducing the Postorder Numbering Approach (PNA), let us take a look at Figure 4.4. This figure shows a SP tree with its root as origin node and each node with its node-number. Each node in the shortest path tree has been also assigned a number, besides its original number, that corresponds to the order in which the node is visited in postorder traversal of the SP tree.

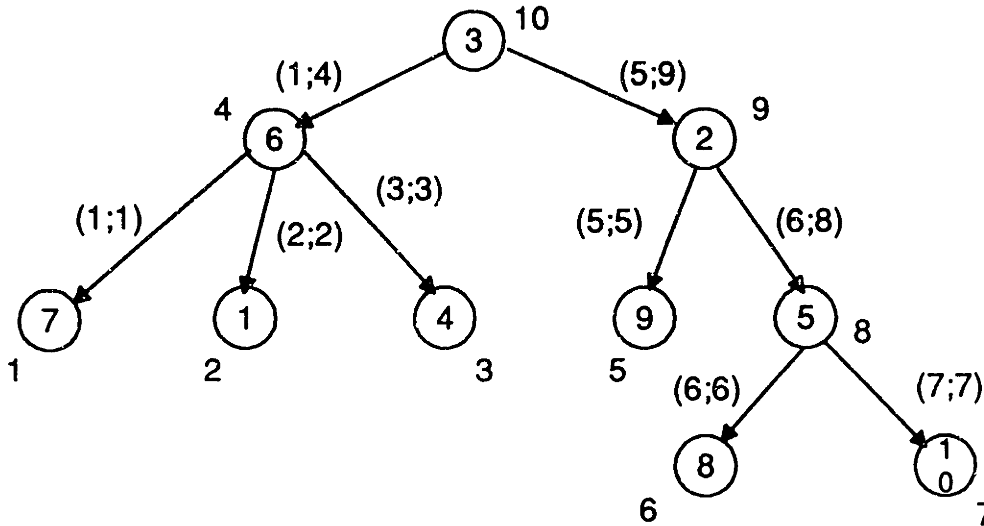


Figure 4.4: Shortest path tree with postorder numbers and intervals

Note now that it is possible to assign an interval to each link a emanating from node i so that the postorder numbers of the nodes that are downstream (descendants) of node i and can be reached through link a all belong in this interval. Note also that because of the characteristics of the postorder visiting method, all postorder numbers that fall between the bounds of the interval are all assigned to the nodes that are descendants of the particular link. Consequently, the link that leads to a particular destination can be easily identified. Assuming for example that our destination is a node with postorder number k, then we just have to look up among the intervals $\{[i,j]_1,[i,j]_2,\ldots[i,j]_{links}\}$, associated with each link emanating from that node, and choose link m such that $k \in [i,j]_m$, namely the link m for which $i_m \leq k \leq j_m$. i_m and j_m are the lower and upper bounds respectively of the interval assigned to link m (see for example, Figure 4.5)

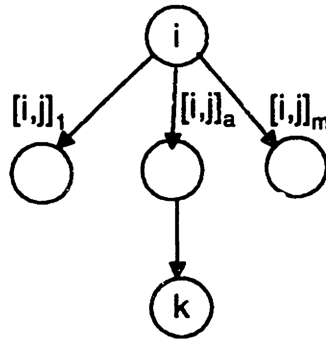


Figure 4.5: Postorder intervals

Let us now consider the example on Figure 4.4. If the postorder number of the destination node is *6*, then, when located at node *3*, link with interval [*5,9*] will correctly lead to the destination. At node *2* we take the link with interval [*6,8*], and at node *5* the interval with [*6;6*].

We store the postorder number information in a *"postorder assignment array"*, **paa[]**, of size n=|*destination nodes*|. The i-th entry of the array, paa[i], is the postorder number that was assigned to destination node i. We need one such array for every origin node. The intervals assigned to a given link may be different for different origin nodes (since the SP trees are also different). To store interval information as a function of the origin node, at each node k we use a *"pointer interval array"*, **pia[]**, of size n=|*origin nodes*|. The i-th entry, pia[i], points to the intervals $\{[i,j]_1,[i,j]_2,\ldots[i,j]_{links\,k}\}$ corresponding to node i as **origin**. The intervals are stored as structures with a maximum size $links_k$ fields in each structure ($links_k$ is the number of links emanating from node k). Note that it is possible that no interval is assigned to a link, since all links by definition can not be part of a shortest path tree. Figure 4.6 summarizes the implementation with paa[] and pia[] for a particular node.

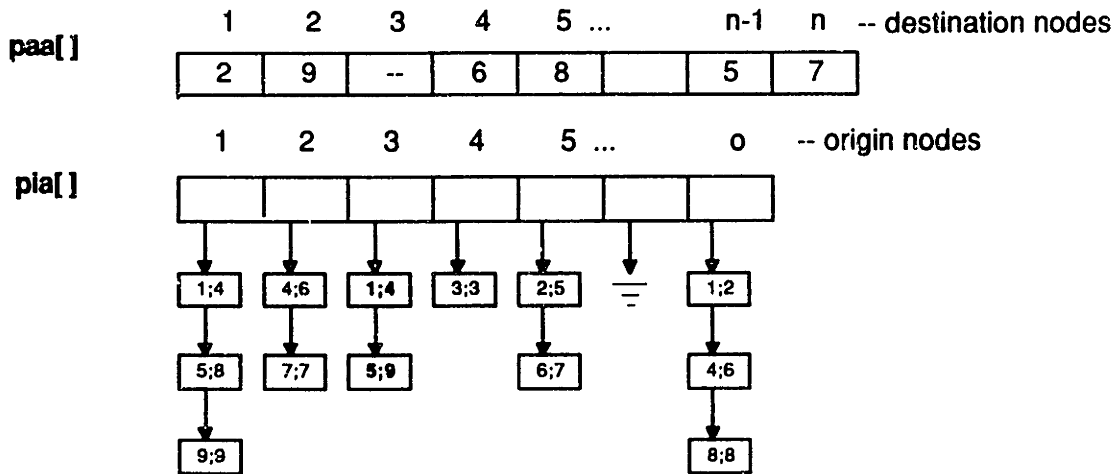


Figure 4.6: Implementation of Postorder Numbering

Storing information in the above way results in a very compressed representation of routing information. Note that at each node k we store a paa[] array of size $|destination_nodes|$, and a pia[] array with the size of $|origin_nodes|$. Each member of the array pia[] points to a linked list with maximum size equal to $links_k$. Thus, the maximum amount of information that we have to store and transmit at each node k is:

$$destination_nodes + origin_nodes * links_k$$

Recall that the amount of information that we have to store and transmit at each node k for the naive approach discussed in section 4.2.1 is:

$$destination_nodes * origin_nodes.$$

Since $links_k << destination_nodes$ and $links_k << origin_nodes$ the routing table construction with the postorder numbering approach requires much less memory than the naive approach. Consequently the amount of information to be transmitted is also substantially less.

4.3.1 Discussion

On the basis of the approach presented above the routing operations take place as follows:

- A vehicle, entering the system at origin node o, obtains and stores through the on-board computer the postorder number that was assigned to its destination from the paa[] array that is transmitted.
- At each node, the pia[] array corresponding to that node, is transmitted to the vehicle as well. The on-board computer, based on the postorder intervals assigned to each link that correspond to its origin node, chooses the link with the interval where the postorder destination number belongs. This process is repeated at each node until the vehicle reaches its destination.

There are several situations during the operation of such a system in which vehicles may not receive the correct information. For example, vehicles exiting the system for a short period and entering later on; information updating takes place before the vehicle reaches its destination; vehicles entering the system at time other than t_0 or $t_0+k^*t_{update}$, ($k=1,2,...$).

We will demonstrate how these problems can be treated best, using the case of information updating as an example. Let us assume that a vehicle departed its origin at time t_0. It is possible, that the travel time to its destination is bigger than the t_{update} time. It is also possible that the SP tree of the vehicle's origin node changes, thus resulting in new postorder assignment. If at time t_0+t_{update} this new information (the new postorder assignments) is provided by the system, the new information can be inconsistent with the postorder values that the vehicle obtained at the origin. The vehicle has to be informed of the change. Subsequently the vehicle proceeds as if it were entering the system at the current node and follow the steps given earlier, i.e., look up the new postorder number from the paa[] at the current node, and follow the interval recommendations as if the vehicle's origin node were the current node. This procedure is also the solution to the problem of drivers exiting and reentering the system. Thus if the vehicle finds inconsistent interval values in the pia[] table, it has to reenter the system, i.e., it has to follow its travel as if it started its journey at the current node. The can be an easy process, since the paa[] arrays are transmitted to the vehicle at each node anyway.

The problem of vehicles entering the system at times between information updates is the most serious of all. Especially, if the update time is large, then it is very possible

that vehicles will enter the system at times in the interval $(t_0; t_0 + t_{update})$. For example when:

- new vehicles enter into the system at any time, i.e., between the update times.
- vehicles leaving a node at t_0 might diverse from the assumed system travel speed, thus they may arrive at the next node earlier or later than it was anticipated in the calculations.
- vehicles in the system might get lost despite the routing information and get back to the proper node, or to a new node, at time when information is not provided. Also in some cases, drivers might not obey the routing recommendations and detour from the advised path later deciding to rejoin the system, again.

Although these problems may be alleviated somehow by frequent information updates in the next section we discuss approaches that can correct the problem, at least partially, at modest computational costs.

4.4 The Rolling Horizon Approach

In this section we introduce the Rolling Horizon Approach (RHA), that overcomes many of the problems and difficulties outlined in the previous sections. The main idea in RHA is that it tries to fill as many cells in the time-dependent routing tables as possible, (see Figure 4.1), based solely on information obtained from the shortest paths solved at time t_0. This method also overcomes the problems mentioned under the PNA method, such as inflexibility, lack of information for motorists entering the system at time other than t_0, and inconsistency due to slower or faster drivers.

Before introducing the method let us take a look at Figure 4.1 again. Note that shortest path calculations take place only at t_0 and $t_0 + t_{update}$ times. This means that routing information could be inserted directly into the rows of t_0 and $t_0 + t_{update}$. The cells between these two rows are empty. Our goal is to fill these entries based on the available information, without calculating shortest paths for the dt intervals between the update times.

Figure 4.7 shows part of the shortest path tree associated with node *1* as origin. For each node we also show the associated label values, i.e., the shortest travel times from the origin node to the particular node.

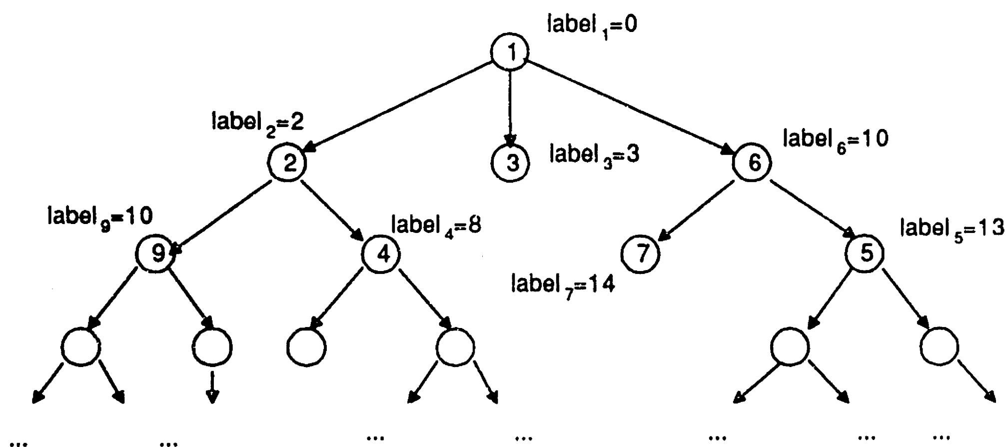


Figure 4.7: Shortest path tree with associated label values

In addition to the shortest travel time the label provides the following information as well: at node i, at time $t_0 + label_i$, the shortest paths between the node i and the nodes that are descendent of i follow the same subtree. This is quite evident for the case where

node i is the origin node. However, note that the above interpretation is valid for any node i of the same shortest path tree; and it is independent of which origin node the SP tree was calculated from. Hence, while we only calculate SP trees for time t_0, as a byproduct we also obtain SP subtrees for the descendent nodes of i for times equal to $t_0 + label_i$. As is shown on Figure 4.8 the SP tree of node *1* (SPT_1) calculated at t_0 also includes part of the SP tree for node *2* for time interval $t_0 + label_2$. This subtree, with node 2 as root, is a subset of the shortest path tree that would result if we had explicitly calculated the shortest paths from node 2 at time $t_0 + label_2$.

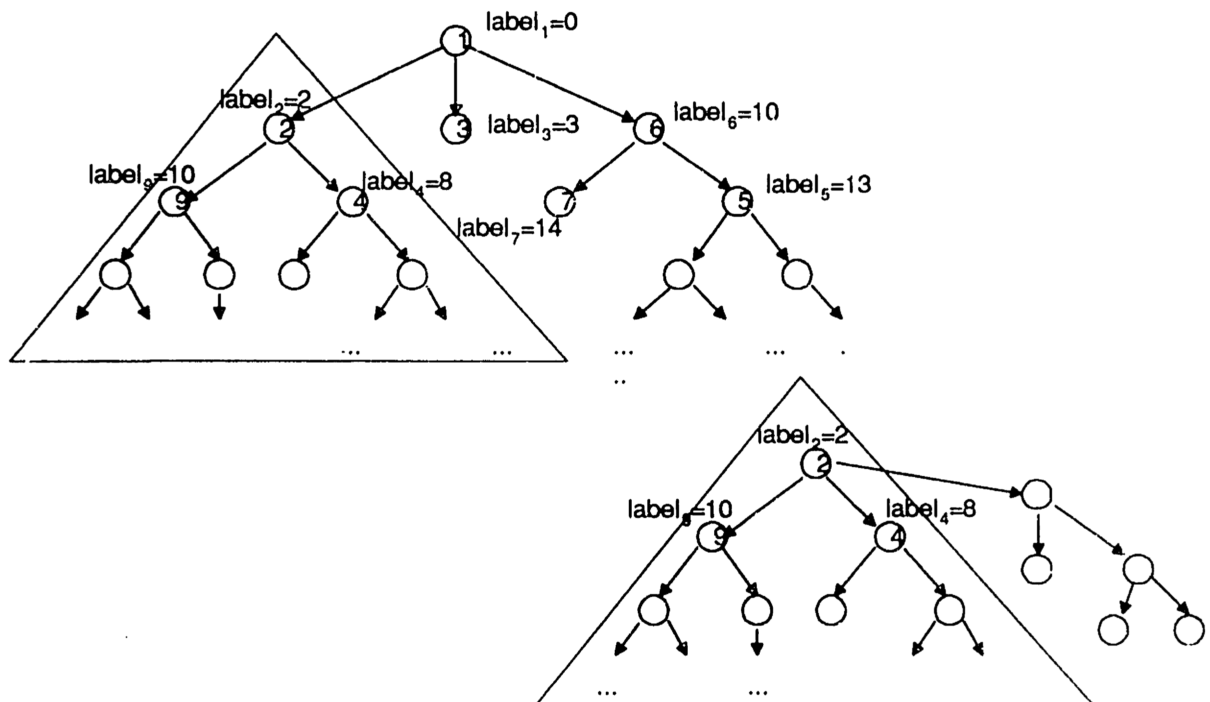


Figure 4.8: Identical subtrees in different shortes path trees

This observation implies that in order to complete the entries of the routing table we may not have to calculate shortest paths from every node to every other node for all *dt* time intervals. We may obtain useful information from the shortest path calculations at t_0. The following lemma formalizes the above discussion:

Lemma 1: Let G(N,A) be a time-dependent network and $SPT_i(t_0)$ the time dependent shortest path tree calculated from the origin node i at time t_0. For any node k, k $\in$ $SPT_i(t_0)$, the subtree with node k as its root is a subset of $SPT_k(t_0 + label_k)$.

The proof of the lemma follows directly from the previous discussion.

4.4.1 Implementation

For the implementation of the routing table for this approach we construct a matrix with number of rows equal to t_{update} / dt, and number of columns equal to the number of destination nodes. One such table is developed for each node (see for example Figure 4.9).

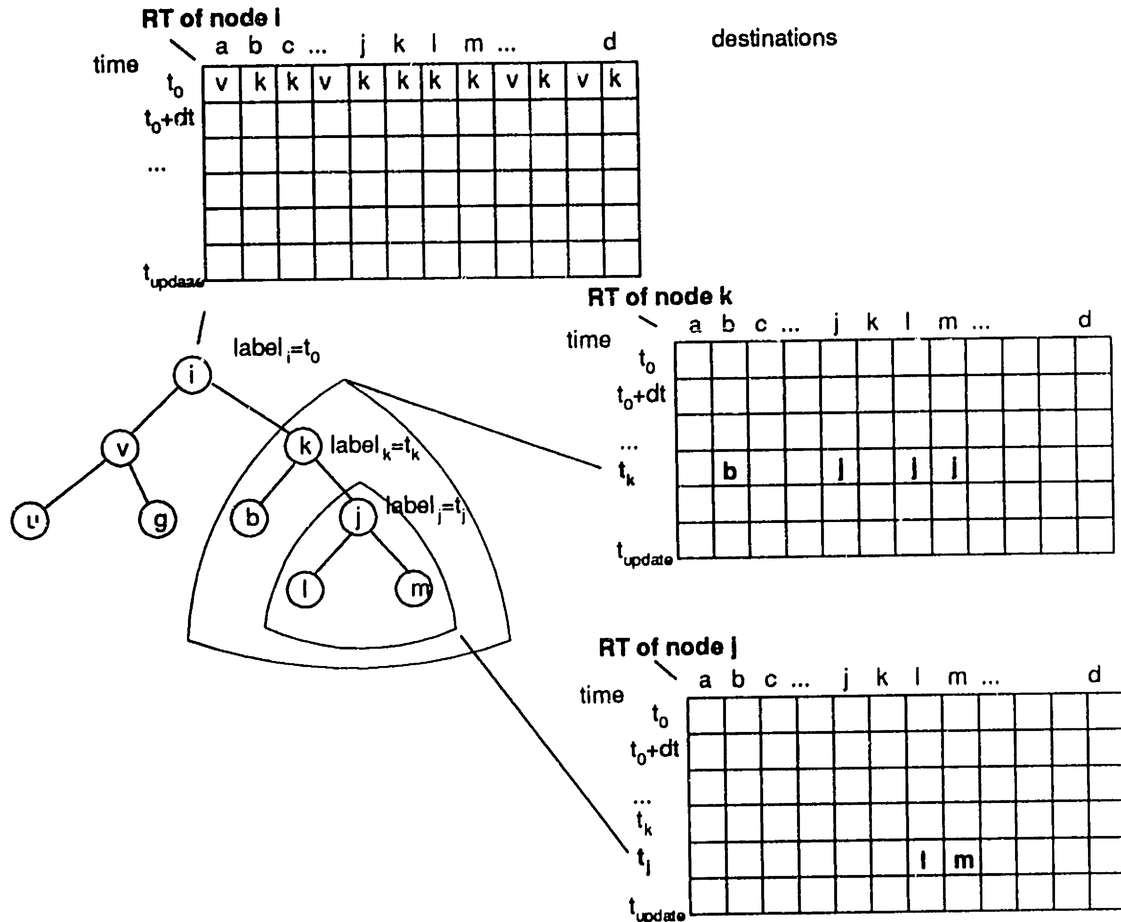


Figure 4.9: Construction of RHA routing tables

The first row of the routing table, for every node i, is completely filled. The entries are obtained from the shortest path tree $SPT_i(t_0)$. Using the same SP tree and based on lemma 1, we can fill entries of rows other than the first, in the routing tables of other nodes. For example, in the routing table of node k we can fill routing information in the row corresponding to time $t_0 + label_k$ ($label_k$ is the travel time on the shortest path from node i at t_0) and all the columns that correspond to destination nodes that are descendants of node k in the $SPT_i(t_0)$ (see Figure 4.9).

The process described above should be repeated with all SP trees calculated from all origin nodes at t_0. Remember that the t_0 row of each routing table will be totally filled, assuming that each node is an origin node, and other entries of the tables will be filled depending on the different subtrees of the SP trees (see Figure 4.10).

RT of node i

time \	a	b	c	...	j	k	l	m	...			d
t_0	v	k	k	v	j	k	k	k	v	k	v	j
t_0+dt	v	k	k	k	j	k		k		k	v	j
...	v		v	v	v		k			k	v	k
	v		v	v			k			k		k
	v						k			k		
t_{update}	v	k	k	k	j	k	k	v	v	k	v	j

...
...
...

RT of node j

time \	a	b	c	...	j	k	l	m	...			d
t_0	d	c	c	b	b	b	b	c	k	n	c	c
t_0+dt	d	c	c	c	c	c	b	b	k	n	c	c
...	d	c	b	c	c	c		c				c
t_g	d	c	c		c	c			k	n	c	b
	c		c		c				k		c	
t_{update}	d	c	b	c	b	c	b	c	k	n	c	b

Figure 4.10: RHA Routing Tables after Insertions

If $dt \to 0$ then the values that would be entered to the same entry of a routing table based on different $SPT_i(t_0)$ trees will all be the same. We will prove the above claim with the following lemma, which is an extension of lemma-1 that was presented earlier.

Lemma 2: Let G(N,A) be a time-dependent network with link costs $c_{ij}(t)$, where $c_{ij}(t)$ is the travel time on link *(i,j)* departing node *i* at time *t*. If the label of node *i* is equal in $SPT_j(t_0)$ and $SPT_k(t_0)$ then for any node *l* that is descendant of node *i* in both shortest path trees:

$$label_l^{SPT_k} = label_l^{SPTj}, \text{ for } \forall l \mid l \text{ is descendent of } i \text{ in both SPT.}$$

To prove the above lemma we have to look at Figure 4.11. Let us assume that the dynamic costs at t_0 are equal from node j and k, $c_{ji}(t_0) = c_{ki}(t_o)$. In this case it is easy to see that the shortest path calculations, from nodes j and k, at t_0 will result in the same label values on node i. Consequently, all label values of nodes that are descendants of node i in $SPT_j(t_0)$ and $SPT_k(t_0)$ will have the same label values. This reasoning can be easily extended to the general case, when:

$$label_j + c_{ji}(label_j) = label_k + c_{ki}(label_k),$$

which is a broader definition of the $c_{ji}(t_0) = c_{ki}(t_0)$ condition.

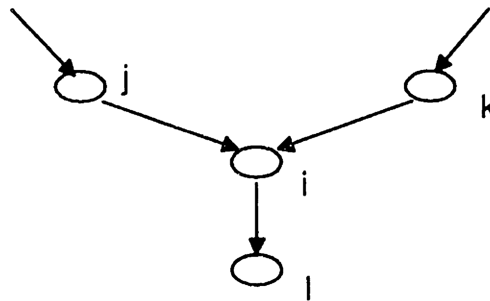


Figure 4.11: Example for Lemma 2

So far we discussed the case with $dt \to 0$. However, it is reasonable to assume that dt will usually correspond to intervals one minute or so large. In these cases it is possible, that based on different shortest path trees, different values enter in the same cell. In this case however, we know that the difference of label values is at most dt and we expect that the different shortest paths that go through this node during interval dt will not differ significantly. Therefore, any value is equally acceptable. In the implementation of our algorithm we keep the value that was entered from the latest shortest path tree calculation.

It is easy to show that with the RHA approach a vehicle traveling from its origin node to its destination node will receive at each node the proper information. If a vehicle starts from node i at t_0 and destination is node d (see Figure 4.10) , then the next node on the shortest path is node j; the vehicle arrives there at time t_g. It receives the routing information b that has been entered into cell $[t_g,d]$ in the routing table of node j. Hence, the vehicle actually receives correct information corresponding to time t_g and not to t_0.

The information entered in the t_g row was based on the SP calculation from node i (or from some other node with the same arrival time at node j). Since the information that was entered into the cells is based on the shortest path tree $SPT_i(t_0)$, the vehicle following the recommendations will always receive the correct information.

However, it is possible that a vehicle going to destination d arrives to a node j at time t_g for which no entry exists in the column corresponding to the vehicle's destination. For this case the recommended approach is the following:

if cell $[t_g,d]=\varnothing$, then the value of a previous cell $[t,d]$ should be inserted (or the closest t time interval).

Note that there is always a value in an earlier time interval, because at least at t_0 there must have been a routing recommendation.

The rolling horizon approach is well suited to address the problems of drivers arriving at nodes at times other than t_0, mentioned earlier. If for example a vehicle leaves the origin at t_0 and drives slower or faster than the predicted speed, then it will arrive at nodes later or earlier than it was anticipated in the calculations. The same situation can happen if a vehicle enters the system at time other than t_0. If a vehicle arrives at a node at time for which information exists, then the vehicle automatically picks up the right information and follows the right shortest path. If this is not the case, i.e., there was no information available for that arrival time, then the best possible information should be provided according to the process discussed earlier. The cases of slower or faster drivers, drivers getting lost temporary or exiting the system and later rejoining may be resolved in the similar way. A vehicle approaching a node always can pick up the best available information corresponding to its arrival at that node.

In general this "memoryless routing" (i.e., routing recommendation independent of the path the vehicle was traveling to the current node or its origin), provides a lot of flexibility. Furthermore with this method it is always possible to receive the latest calculated or modified information even en-route. Suppose, that a vehicle enters the system at node i at time other than t_0 and recommendation corresponding to time t_0 only is available. Suppose also that for the actual time of arrival at node i, the corresponding shortest path has changed (from the one calculated at t_0). The vehicle starts its route on the path corresponding to t_0. Since, under the RHA strategy at all nodes the best available

information is provided, there is a chance that during its travel the vehicle will arrive at a node k at a time where the correct shortest path information is available. Note also, that if the path is long or the vehicle entered the origin node at a time close to the next update then the vehicle may obtain quickly the new updated information at a subsequent node.

4.4.2 Experimental Results with the RHA

The success of the RHA routing method depends on the number of entries filled in the routing table. Thus the question of coverage becomes crucial in evaluating this approach. By coverage we mean the number of filled entries in the routing table as a percentage of the total number of entries. The total number of entries is equal to the product of the number of destination nodes and the number of time intervals between travel time updates, (see Figure 4.10).

Different nodes in the network have different coverage. The differences are attributed to network topology and traffic conditions (travel times). High coverage of a column d in the routing table of node j, implies that many shortest paths from various origin nodes towards node d go through node j. The more travel times from various origins to node j differ, the higher the coverage of column d will be. This is true since different label values imply that we can fill routing information in cells of the column d corresponding to different dt time intervals. This observation can be generalized also for the node coverage: if node j has high coverage it means that node j is part of many shortest path trees. This coverage may be higher if the node has label values (representing calculated travel times) evenly distributed in the different shortest path trees $SPT_i(t_0)$. Therefore, the coverage of a node j depends on:

- the number of times node j is part of shortest paths between any two nodes <u>and</u>
- the shortest path travel time distribution from origin nodes to node j.

Consequently the following factors are important on determining the coverage:

- the length of the update interval (in relation to travel times),
- geographical location of a node j in the network, and
- size of the network.

To address these issues more formally we have examined the performance of the RHA method on the Sudbury network (205 nodes, 578 links). The dt was set equal to 20

seconds and t_{update} time 5 minutes (therefore the number of dt intervals is 15). The conflicts filling into the dt intervals are resolved arbitrary (the last value that the algorithm has inserted).

Figure 4.12 shows the average coverage values for the 15 dt intervals.

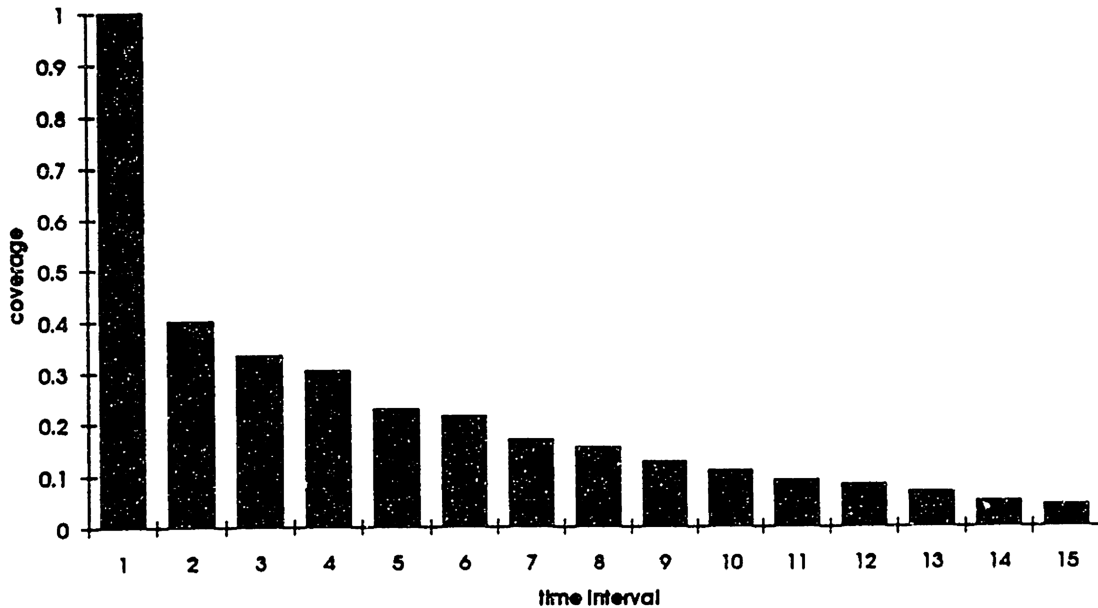


Figure 4.12: Coverage values of dt time intervals

The first interval (t_0) has 100% coverage (since the shortest path calculations for t_0 from all origins to all destinations are available). Note also that, even though shortest path calculations are available only for the time interval t_0, the time intervals after t_0 are also filled to a certain level providing shortest path recommendations for other time instances. Therefore, the results shown in Figure 4.12 support out a-priori expectations, that it is possible to derive information about shortest paths corresponding to time intervals other than t_0 based only on the $SPT_i(t_0)$ results.

The next step in our analysis is to evaluate the average coverage level for the entire network. The length of the t_{update} interval affects the average coverage. This is true since coverage is the number of cells filled as a percentage of the total number of cells in the routing tables (which is function of dt and t_{update}). In our analysis we varied the t_{update} times from 1 to 5 minutes while dt remained 20 seconds. Figure 4.13 shows the average coverage values of all nodes in the network for the different t_{update} times (black bars).

Figure 4.13 also shows the average coverage when we exclude from the evaluation the first row (the row corresponding to time t_0), (white bars).

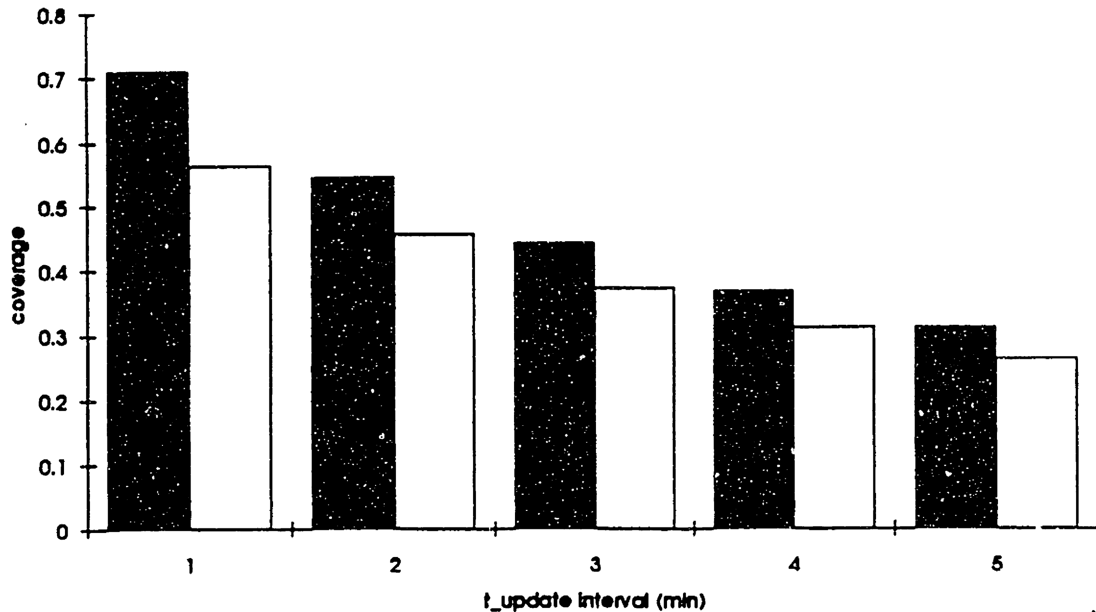


Figure 4.13: Overall coverage (black bar) and coverage for periods other than t_0 as a function of the length of update interval (white bar)

The results indicate that as t_{update} increases the coverage values diminish. The coverage (including the first row) assuming 5 minute t_{update} time is 31%. It is 44% percent for the case of 3 minutes and above 70% when t_{update} is equal to 1 minute. The coverage values corresponding to the case when the first rows are not included follow similar trends.

The explanation of the above behavior lies in the fact that travel times affect the possible range and distribution of the label values of a node i in the $SPT_i(t_0)$'s. Particularly, if shortest path travel times (at time t_0) do not exceed k minutes then the RHA method will not produce any recommendations corresponding to time intervals $t > t_0+k$, either. The average travel time of this network is 3.2 minutes.

In the network the coverage values of nodes also vary. Figure 5.14 shows the distribution of these coverage values for different t_{update} intervals (1-5 minutes.) The x axis corresponds to coverage intervals and the y axis shows the percentage of nodes with coverage into the corresponding interval.

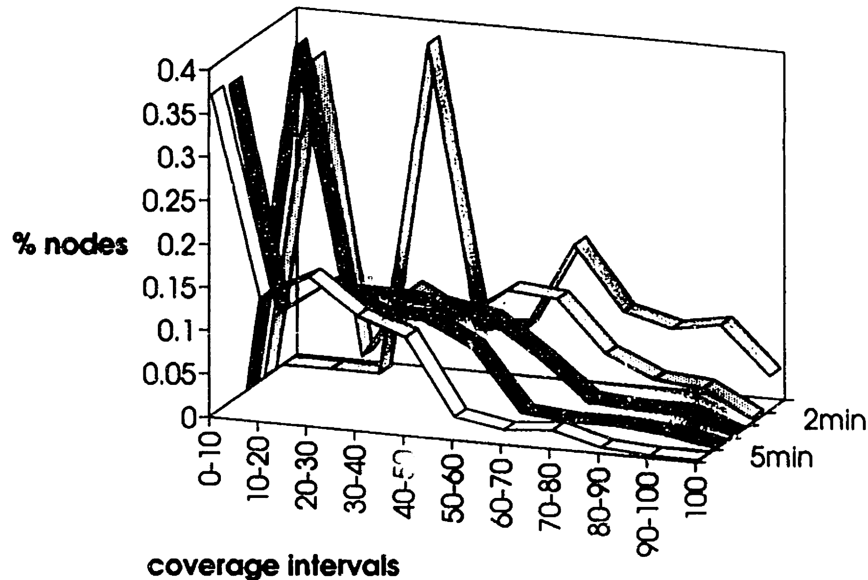


Figure 4.14 Distribution of nodes with different coverage values

The figure clearly shows that as we move to lower t_{update} times the higher coverage intervals become more frequent. For instance, for $t_{update} \leq 3$ all nodes have at least a 10-20% coverage. For these cases we can also see that the relative frequency of the highly covered nodes substantially increases.

From the results it is clear that if a node j is well covered, then a relatively large number of shortest paths go through that node. We expect that the well covered nodes will be clustered around the center part of the network. To examine this hypothesis we have selected a central node in the network and obtained the travel times from that node to all other nodes. We grouped the nodes into 30-second intervals according to their travel time distances from the central node, i.e., all nodes that were $0 < t < 30$ seconds away from the central node grouped under interval 0-30, etc. Figure 4.15 shows the average coverage as a function of the distance from the center node (t_{update} is 5 minutes). The results follow our a-priori expectation: the central nodes have higher coverage values and the coverage diminishes (exponentially) as the distance from the center increases. Nodes close to the center have high coverage values (43-66%). The average coverage for nodes 2-2.5 minutes away from the center node was above 30%. For comparison note that the

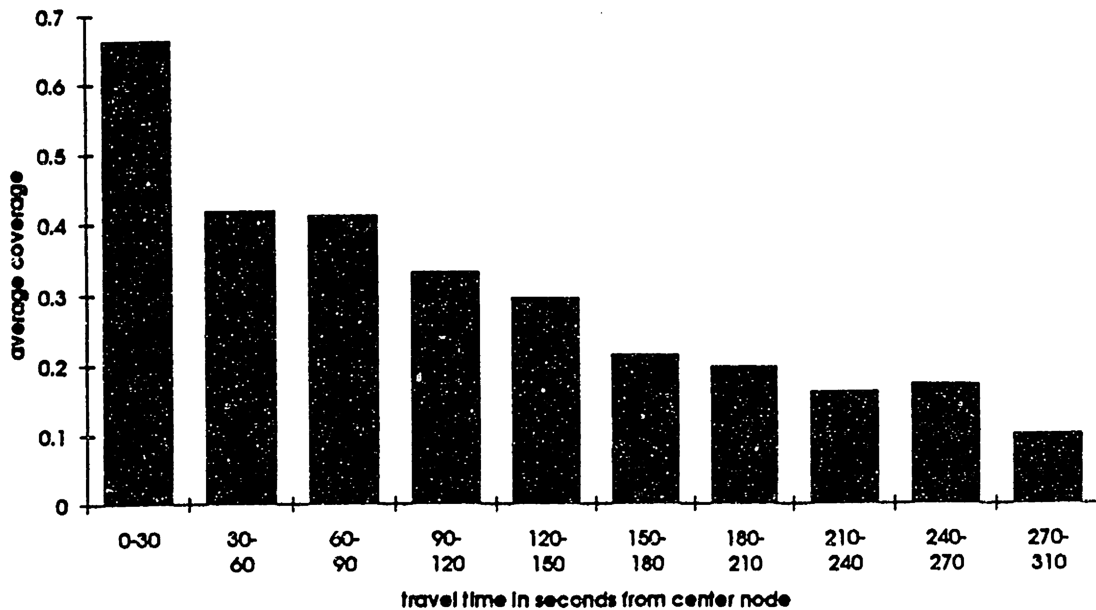


Figure 4.15: Average coverage as a function of the distance from the center

In conclusion the RHA method produces high coverage for the nodes in the central part of the network, where usually the demand for correct and frequent updates might be higher.

Let us now discuss the implications of the above observations for the RHA approach. In particular we will discuss the implication of having many or few entries filled in a column of a routing table. If the routing table of node i has routing values filled only in the t_0 row, this implies that node i was always a leaf node in all SP trees, i.e. there is no shortest path that goes through node i (or if it does the corresponding label of node i belongs to the $[t_0; t_0 + dt]$ interval). Let us look for example at node i on Figure 4.16a, (the arcs with bolded lines represent links that are used by many shortest paths).

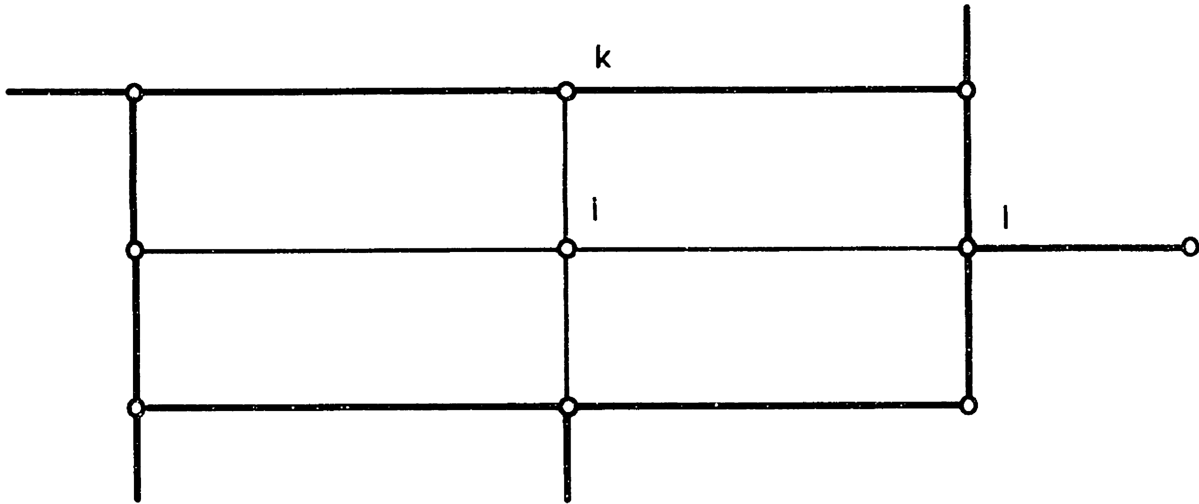


Figure 4.16a: Central node surrounded by nodes with high coverage

If node i is not a cutset node and its routing table is not well covered, then this would imply that the costs of links incident to i are high relative to the arc costs around that node. Hence, shortest paths do not use node i and vehicles will not be routed through node i. Therefore, the demand for routing recommendations at this node may be low, only consisting of the vehicles that enter the network at node i. Since most of the demand at this node is generated by vehicles originating their travel at node i, routing recommendation at t_0 should be adequate. The only problem may arise if vehicles start their travel at this node at time other than t_0. However, these vehicles will soon approach a node that is part of many shortest path trees, i.e., a node with high coverage value (e.g. node k or l on Figure 4.16a). In this case, the vehicle can pick up the correct information corresponding to its destination and from that node it can continue its travel on the correct shortest path.

The above argument should be true, in most real situations, when node i is in the central area. However, for nodes on the periphery the above may not hold. Assume for example, that a vehicle with destination node d arrives at node i at time t_a (see Figure 4.16b). If t_a is much later than t_0, most likely there will be no entry in the routing table that corresponds to t_a. This may be so since, in order to find a value in the row corresponding to t_a (reasonably larger than t_0) node i should have been part of a shortest path from some other node o to node d with $label_i^{SPT_o} = t_a$. However, if t_a is relatively larger than t_0 the probability to have such a path is very small. Since node i is on the side of the network, it is not likely that a shortest path that is t_a units of time away from node i

goes through node i (unless there is a destination node in the vicinity). Furthermore, given that most of the vehicles through node i are vehicles originating at i, it is likely that traffic conditions on the path from i do not change a lot and hence the recommendation at time t_0 may be adequate for at least the first part of the trip. Besides, when a car arrives at a node or enters the system at time t_a that is much later than t_0 this also means that the next update time is close. At that time (next update) the vehicles at all nodes will receive the correct shortest path recommendations.

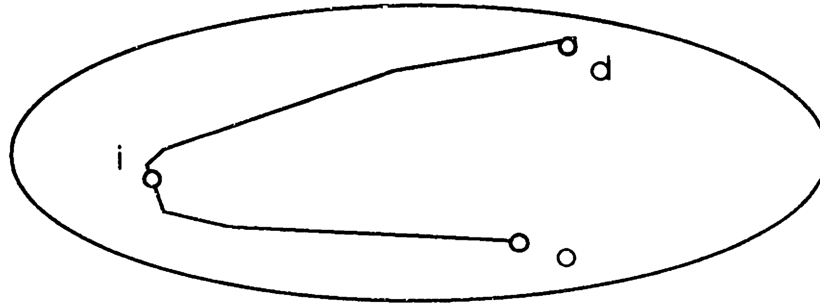


Figure 4.16b: Inadequacy of the RHA with nodes in the periphery

4.5 Computational Considerations

The results from the previous section indicate that it is possible to achieve good coverage with the RHA method. In this section we discuss some approaches that can be used in order to combine shortest path calculations and the RHA under one common strategy. The major goal with the RHA method is to fill the entries between update times using only existing information, i.e., the shortest path results at time t_0. We want to do that as opposed to calculating shortest paths for every dt time interval.

A possible strategy for combining the shortest path calculations with the RHA method is to calculate shortest paths only from origin nodes. Since the number of the origin nodes is smaller than the total number of nodes in the network, shortest path computation time will be also less. Note, that utilizing the special characteristics of the RHA method we can fill not only the columns of the origin nodes but also the columns for all nodes. Figure 4.17 shows for two different strategies the coverage values of the nodes that are in the central area of Sudbury network (note that origin/destination nodes are not included in the selected nodes.) The white bars show the coverage values obtained from

calculating shortest paths only from 53 origin nodes (strategy-2), and the black bars show the results obtained calculating shortest paths from all nodes of the network (strategy-1).

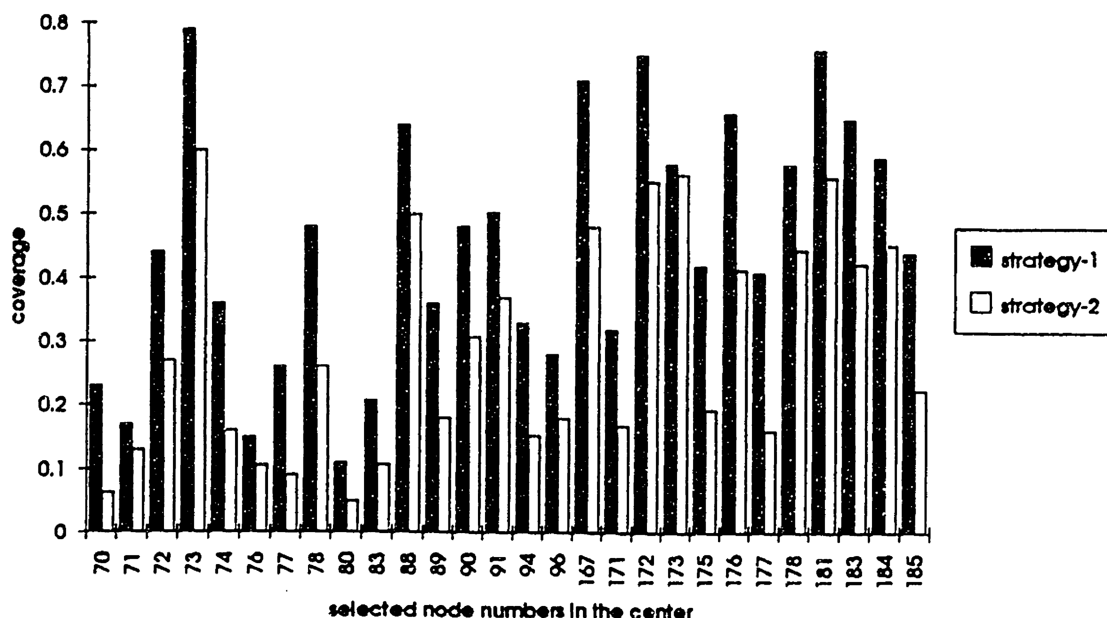


Figure 4.17: Comparison of coverage for Strategy-1 and Strategy-2

The coverage values from strategy-2 are slightly below the coverage values from strategy-1. Table 4.1 summarizes the results by comparing the average coverage values for the two strategies corresponding to the 28 central nodes and to all nodes of the network (the coverage values include the t_0 row as well).

	strategy-1	strategy-2
central nodes	45% coverage	29% coverage
all nodes	31% coverage	17% coverage

Table 4.1: Node coverage for Strategy 1 and 2

Note that in the case of strategy-2 the computational time for calculating the shortest paths is about 1/4 of the computation time of strategy-1 (we calculate shortest path trees from 53 nodes as opposed to 205). The coverage however has been reduced by only 45%. In the central area, strategy-2 produces 29% coverage as opposed to 45% of strategy-1.

Further analysis of the routing tables shows that the filled entries are not distributed uniformly within columns. They tend to be clustered around time intervals closer to t_0 (similar to the tendency illustrated in Figure 4.10). However, it is desirable that the cells of columns are filled more evenly, since vehicles entering the system at time t, later than t_0, would have a higher probability to find a correct recommendation.

We therefore introduce strategy-3 that tries to improve the coverage distribution. In this strategy we calculate shortest paths only from the origin nodes but for times t_0 and $t_0 + t_{update}/2$. Using this method we will have two base times and therefore the route recommendations will be clustered around two time intervals, improving the overall coverage distribution. Figure 4.18 shows node average coverage for the 15 dt intervals resulting from strategy-3.

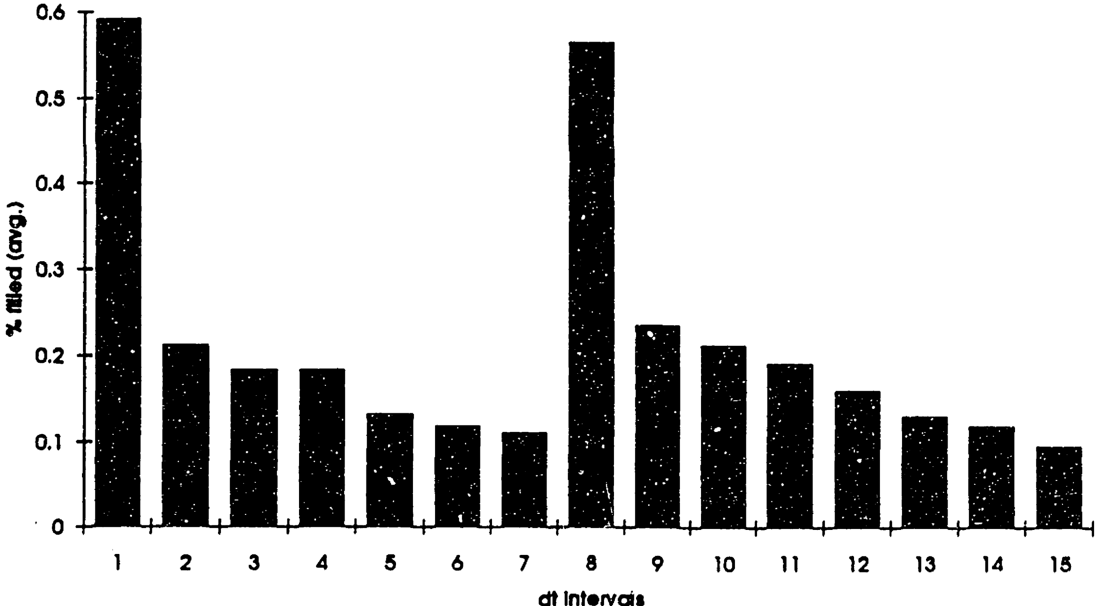


Figure 4.18: Node coverage resulting from Strategy-3

Figure 4.19 shows the coverage results on the central nodes for the three different strategies discussed already.

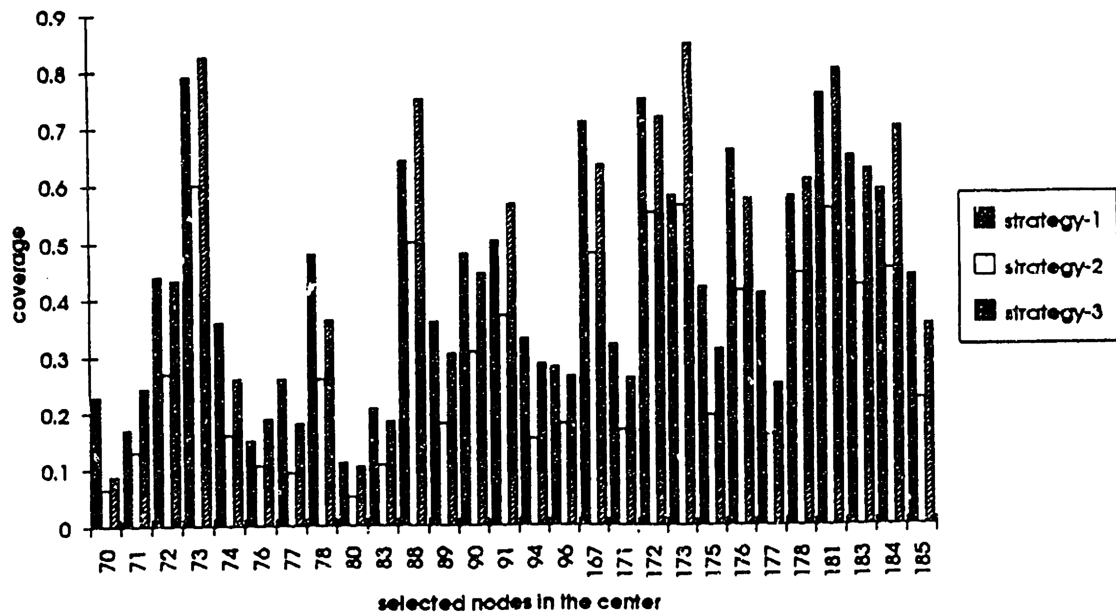


Figure 4.19: Selected node coverage for Strategy-1, Strategy-2 and Strategy-3

The above figure shows, as we expected, that strategy-3 always results in better coverage values than strategy-2, and it is very close to the coverage from strategy-1. For certain nodes it gives even better coverage than strategy-1 (e.g., nodes 88, 173, 181, etc.). This is a very promising result since the number of shortest paths trees that were solved for strategy-3 is about half of the number solved for strategy-1.

Table 4.2 summarizes the performance of the three different strategies (the numbers in parenthesis give the average coverage excluding the t_0 row).

	strategy-1	strategy-2	strategy-3
average coverage for all nodes	31% (26%)	17% (14%)	29% (24%)
average coverage for central nodes	45%	29%	43%
number of shortest paths solved	205	53	106

Table 4.2: Coverage values for Strategies 1, 2, and 3

The results are very interesting. The third method, although it is based on less shortest path information, than the first one, provides results that are more evenly distributed in the time horizon. Strategy-3 uses the available information more efficiently

than strategy-1. Strategy-1 for example, might fill the same entry of a routing table many times based on different shortest path trees. However, in strategy-3 this redundancy is minimized since the shortest path calculations are better distributed in time.

Table 4.3 shows the running times for the different strategies including the shortest path calculations and the RHA routing table generation. Note, that we used a very simple implementation for constructing RHA tables. It is possible that with a more sophisticated implementation, the running time will be further reduced. We could use for example a modification of the postorder tree visiting technique that we already discussed. The RHA method is also very scalable and easily supports parallel implementation.

	strategy-1	strategy-2	strategy-3
number of shortest paths solved	205	53	106
Running time	6.8 sec	1.9 sec	3.7 sec

Table 4.3: Computational results for different strategies

The only drawback of strategy-3 (and strategy-2) is that the t_0 rows are guaranteed to be filled only in the routing tables of the origin nodes. Therefore some columns in the routing tables of other nodes may not have routing information. We have already discussed in section 4.1.2 the implications of such omissions. It means that shortest paths do not go through node i. It also means that if vehicles follow shortest paths they will be never routed through node i, unless this node is their destination. One way to overcome the problem of no information is to utilize a strategy that combines strategies 1 and 3. For example strategy-1 is implemented every n-th upgrade and strategy-3 implemented in all other cycles. Under this strategy, if no information was available from strategy-3 cycles, the information obtained from the previous strategy-1 cycle is provided by default. This method therefore can combine the positive characteristics of strategy-3 and its drawbacks may be corrected by utilizing strategy-1 every n-th cycle.

In conclusion the RHA method provides flexibility for combining shortest path calculations with the construction of routing tables. There is a trade-off between running time and coverage but further research is needed.

Chapter 5 Conclusion

5.1 Conclusions

This thesis has focused on the solution of the dynamic shortest path problem approaches for and dynamic routing table generation in large transportation networks. The static shortest path problem has been studied extensively and a vast amount of literature already exists. The time-dependent shortest path problem has also been studied in quite some detail, but only for communications networks. A few differences in the characteristics of communications and transportation networks prevent these results from being directly applicable to our problem. So far, not much work has been done in the area of time dependent shortest path problems for transportation applications. This has been one of the main motivations for this thesis. The problem of time-dependent shortest path problem has been studied theoretically by Kaufman and Smith [KaSm92], and Koutspoulos and Xu [KaXu93]. Ziliaskopoulos and Mahmassani [ZiMa92], presented a new time-dependent shortest path algorithm for IVHS applications.

Because of storage constraints, there is a need for aggregating link travel times. Hence, travel time costs for each link should be stored in an array where each entry contains the aggregated average travel times for a dt time interval. In transportation networks these time intervals can be 30 seconds, 1 minute or even more. We showed that with such discretization, even when the FIFO assumption holds for the original data, there is a need for linear interpolation for obtaining cost values for time t (when t falls between the time instances of the given discretized cost values). The FIFO assumption still holds for the interpolated data.

Different types of algorithms were investigated to solve the dynamic shortest path problem. Assuming that the FIFO assumption holds we modified the binary-heap implementation of the Dijkstra's label setting algorithm and the dequeue implementation of Pape's label correcting algorithms to accommodate the time-dependency of the problem. These algorithms were chosen due to their efficiency in sparse networks.

The computational results showed that the dynamic dequeue implementation of Pape's algorithm is superior to the binary-heap implementation of Dijkstra's algorithm for all network sizes. The results were based on the running times for solving the shortest path from all nodes to all nodes in two real networks -- Sudbury (205 nodes, 578 links)

and Boston (4922 nodes, 11647 links) -- and three square grid networks of size 20x20 (400 nodes, 1520 links), 40x40 (1600 nodes, 6240 links) and 70x70 (4900 nodes, 19320 links). The results indicated that:

- Both dynamic shortest path algorithms solve the problem correctly
- The dynamic dequeue implementation outperforms the label setting algorithm
- The computational time of the algorithm in large size network exceeds the acceptable time requirements of real-time systems

The above conclusions also motivated the investigation of possible improvements. Two strategies were suggested. The first strategy -- terminate Dijkstra's algorithm when all destination nodes become permanent -- showed that in average about 10% running time improvements can be achieved. This was a notable result by itself but the dequeue implementation of the label correcting algorithm still outperformed this version of Dijkstra's algorithm.

As a second strategy for improvement, algorithms for distributed/multiprocessor environment were investigated. The basic approach consisted of the following steps: decompose the network into few subnetworks, solve the time-dependent shortest path parallel in each subnetwork, and organize partial results to obtain global solution. Time dependent virtual links were introduced between the origin nodes, cutset nodes, and destination nodes of the network. Three different virtual links were defined. Node-to-cutset virtual links connect each origin node with all cutset nodes of the same subnetwork. Cutset-to-cutset virtual links connect cutset nodes of the same subnetwork. Cutset-to-destination virtual links connect the cutset nodes with destination nodes of the network. The definition of cutset-to-destination virtual links enables the algorithm to find shortest paths to the destination nodes only. The cost of a virtual link (i,j) for time t was defined as the travel time on the shortest path from node i to node j, departing at time t.

The Distributed Dynamic Shortest Path (DDSP) algorithm proposed, solves the dynamic shortest path problem maintaining a two-level network structure. The lower level consists the subnetworks with original nodes and arcs enhanced with cutset-to-cutset virtual links and the higher level consists of the virtual link representation of the network. The algorithm solves the problem in four steps maintaining and exchanging shortest path results between the low subnetwork level and the high virtual link level. Depending on the

density of the network at each step the dynamic label correcting or the label setting algorithm was used.

The DDSP algorithm, using one processor, was faster than the serial (dequeue implementation) algorithm only when the number of destination nodes is very small (less then one percent). We also tested the distributed implementation (with four processors) of the DDSP algorithm in a real and various grid networks. Each network was divided into four subnetworks and one processor was assigned to each subnetwork. The parallel implementation of the DDSP algorithm was faster then the best serial algorithm. The computational results showed that the running time of the algorithm is strongly influenced by the number of destination nodes in the network. Due to the aggregate characteristics of cost on virtual links the DDSP algorithm may not identify the correct shortest path between certain nodes. Experimental results showed that this is the case in less than 1% of all shortest path calculations. The average and maximum differences were a few seconds, which correspond to about one percent of the average travel time in the network.

However, the DDSP algorithm was slower than the strategy where one fourth of the origin nodes were assigned to each processor and the serial algorithm was used (by each processor) to solve the assigned shortest paths. This is due to the fact that in grid networks the number of cutset nodes is large and the performance of the DDSP algorithm strongly depends on the number of cutset nodes.

Indeed the experiments clearly demonstrated that when the number of cutset nodes is small the DDSP algorithm performs better than other implementations, even with the same number of processors. In practice many cities, due to the existence of physical boundaries, e.g., rivers, can be naturally decomposed into subnetworks with very few cutset nodes. In such cases the DDSP algorithm has a lot of advantages.

The thesis also focused on the construction of dynamic routing tables. Taking into consideration the special characteristics and difficulties of dynamic routing tables two approaches were presented. The routing with Postorder Numbering Approach (PNA) minimizes the amount of storage and information transmission. Despite its efficiency this approach has several problems, for example with vehicles entering the system at times other than t_0.

The Rolling Horizon Approach (RHA) overcomes most of the problems associated with the PNA method. The basis of RHA is that it uses all the information contained in shortest path trees at time t_0 to generate complete routing tables. The important observation is that time-dependent shortest path trees carry information about subtrees of shortest path trees corresponding to other origin nodes and other starting times.

In order to evaluate the RHA method the resulting node coverage, i.e., the number of filled entries in the routing table as a percentage of the total number of entries, was examined. Coverage ranged from 31% to 71% for one of the transportation networks. The results also showed that the nodes in the central part of the network have higher coverage values.

Finally the thesis presented some heuristics that can be used in order to combine shortest path calculations and the RHA approach under one common strategy. For example we may calculate shortest paths only from the origin nodes and obtain routing information for the other nodes using the RHA method. Experiments showed that the coverage resulting values of this heuristic fell slightly below the coverage obtained from the initial approach where shortest paths were solved from all nodes. The other heuristic finds shortest paths from the origin nodes only but for two different time periods. The resulting coverage was very similar to the one obtained by the initial approach. However, the number of shortest paths to be solved was substantially reduced and the running time was about half of the running time required by the first strategy.

5.2 Future Research

The research presented in this thesis could be extended in several directions:

- Examine other methods to improve the standard shortest path algorithms -- e.g., upper limit on label values between new iterations.
- Examine strategies for network decomposition that optimize the performance of the DDSP algorithm.
- Examine possible improvements of the DDSP algorithm: new network structure, dynamic network building techniques, using other type of shortest path algorithms in different steps, new data structures.
- Investigate new algorithms and implementation approaches for the RHA for constructing routing tables.

- Study the coverage in transportation networks using probabilistic modeling of network and travel time characteristics.

- Further investigate new strategies that combine the DDSP (or other) shortest path algorithms and the RHA approach.

Bibliography

[AlHa91] Al-Habbal, M. "A Decomposition Algorithm for the All-Pairs Shortest Path Problem on Massively Parallel Computer Architectures", PhD Thesis, Massachusetts Institute of Technology, Cambridge, MA., (1991).

[AMOr93] Ahuja,R.K., Magnanti,T.L., Orlin, J.B. "Network Flows: Theory, Algorithms, and Applications", Prentice Hall, N.J., (1993).

[BBHK92] Ben-Akiva, M., Bernstein, D., Koutsopoulos, H.N., and Sussman, J.M., "The Case for Smart Highways", *Technology Review*, **95**(5), (1992).

[Bell57] Bellman, R., "Dynamic Programming", Princeton University Press, Princeton, N.J., (1957).

[Bell58] Bellman, R., "On a Routing Problem", *Quarterly of Applied Mathematics* **16**, (1958), pp. 87-90.

[BlHu77] Blewett, W.J., Hu, T.C., "Tree Decomposition Algortihm for Large Networks", *Networks* **7** (1977) pp. 289-296.

[CoHa66] Cooke, K.L., Halsey, E., "The Shortest Route Through a Network with Time-Dependent Internodal Transit Times", *Journal of Mathematical Analysis and Applications* **14**, (1966) pp. 493-498.

[DGKK79] R. Dial, F. Glover, D. Karney, D. Klingman, "A Computational Analysis of Alternative Algorithms and Labelling Technics for Finding Shortest Path Trees", *Networks*, Vol. **9** (1979) pp. 215-248.

[Dial69] Dial, R., "Algorithm 360 Shortest Path Forest with Topological Ordering", *Communication of ACM* **12** (1969) pp. 632-633.

[Dijk59] Dijkstra, E., "A Note on Two Problems in Connections with Graphs", *Numerische Mathematics*, **1**. (1959), pp. 269-271.

[Ford56] Ford, L.R., "Network Flow Theory", *Report P-923*. Rand. Corp., Santa Monica, CA. (1956).

[GaPa88] Gallo, G., Pallotino, S., "Shortest Path Algorithms", *Annals of Operations Research*, **7** (1988) pp. 3-79.

[Hall87] Hall, R.W., "The Fastest Path Through a Network with Random Time-Dependent Travel Times", *Transportation Science* **20**, (1987) pp. 182-188.

[Halp77] Halpern, J., "Shortest Route with Time Dependent Lenght of Edges and Limited Delay Possibilities in Nodes", *Zeitschrift für Operations Research* **21** (1977) pp. 117-124.

[Hoff91] Hoffmann, G., "Up-to-minute Information as we Drive -- How it can Help Road Users and Traffic Management", *Transport Reviews* **11**, (1991), pp. 41-61.

[Hu1969] Hu, T.C., "Integer Programming and Network Flows", Addison-Wesley, Reading, Massachusetts, 1969.

[John84] Johnson, M.J., "Updating Routing Tables after Resource Failure in a Distributed Computer Network", *Networks* **14** (1984) pp. 379-391.

[KaSm92] Kaufman, D.E., Smith, R.L., "Fastest Paths in Time-Dependent Networks for Intelligent Vehicle/Highway Systems Applications", University of Michigan, Working paper (1992).

[KlKa77] Kleinrock, L. Kamoun, F., "Hirearchical Routing for Large Networks", *Computer Networks* **1** (1977) pp. 155-174.

[KoXu93] Koutsopoulos, H.N., Haiping Xu, "An Information Discounting Routing Strategy for Advanced Traveler Information Systems (ATIS)," forthcoming in *Transportation Research C.*

[Muku92] Mukundan, A., "A Real Time System for IVHS Traffic Modelling", Master's Thesis, Massachusetts Institute of Technolgy, Cambridge, MA., (1992).

[OrRo90] Orda, A. Rom R., "Shortest -Path and Minimum-Delay Algorithms in Networks with Time-Dependent Edge-Lenght", *Journal of Associacion of Computer Machinery* **37** (1990) pp. 607-625.

[OrRo91] Orda, A. Rom R., "Minimum Weight Paths in Time-Dependent Networks", *Networks* **21** (1991) pp. 295-319.

[Pape74] Pape, U., "Implementation and Efficiency of Moore-algorithms for the Shortest Route Problem", *Matematical Programming* **7**, (1974) pp. 212-222.

[Pape80] Pape, U., "Algorithm 562: Shortest Path lengths", *ACM Transactions on Mathematical Software* **6**, (1980) pp. 450-455.

[PeUp89] Peleg, D. Upfal, E., "A Trade-Off Between Space and Efficiency in Routing Tables", *Journal of Associacion of Computer Machinery* **36**. (1989) pp. 510-530.

[Stra92] "Strategic Plan for Intelligent Vehicle-Highway Systemsin the United States" - Final Draft. IVHS America, (1992).

[Sega77] Segal, A., "The Modeling of Adaptive Routing in Data-Communication Networks", *IEEE Transactions on Communications*, Vol. Com-**25**. No.**1**, (1977) pp. 85-95.

[ZiMa92] Ziliaskopoulos, A.K., Mahmassani H.S., "A Time-dependent Shortest Path Algorithm for Real-Time Intelligent Vehicle/Highway Systems Applications", Presented at the 72nd Annual Meeting of the Transportation Research Board, Washington D.C., January 1993.