Routing Reconfiguration in IP Networks

by

Paolo Narváez

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2000

© Massachusetts Institute of Technology 2000. All rights reserved.

Author ........ 

Department of Electrical Engineering and Computer Science

May 18, 2000

Certified by ........ 

Kai-Yeung Siu
Associate Professor
Thesis Supervisor

Accepted by ........ 

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Routing Reconfiguration in IP Networks

by

Paolo Narváez

Submitted to the Department of Electrical Engineering and Computer Science on May 18, 2000, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

This thesis will focus on new methods for efficient IP routing reconfiguration in the presence of topological changes. A fast response to changing network conditions is essential to prevent network routing instability as well as to ensure quality of service parameters. We will identify problems with the current degree of adaptiveness of existing IP routing schemes. New algorithms and protocols will be presented to improve such adaptiveness at different time scales.

The first problem is about the recomputation of shortest path trees performed in link-state protocols. This recomputation is generally done by static SPT algorithms. This means that all the previous information on the previous SPT is deleted and needs to be recomputed from scratch. We develop many new and different dynamic shortest path algorithms to perform this recomputation in an efficient way. By using information from the previous tree, these new dynamic algorithms can achieve great savings in computational complexity. Furthermore, less changes take place in the routing table.

Another problem is about flooding in link-state protocols after changes in the cost metric of a link. Current link-state databases need to be synchronized over the entire routing area. This means that after a link-state change, even distant routers in the area need to be notified of changes in the topology. By developing a new local restoration algorithm, significant overhead savings and speed improvements can be achieved by keeping the information about the changes in a vicinity of nodes and performing all routing reconfiguration at a local level.

Finally, a routing table generally contains only one next-hop entry for every destination. This means that when a neighboring link fails, the forwarding engine has to wait until the routing table is recomputed. We develop a multiple path algorithm that allows the routing table to contain several possible next-hops for every destination. This way, the forwarding engine can automatically switch to another next-hop without having to wait for the routing table to be recomputed.

Thesis Supervisor: Kai-Yeung Siu
Title: Associate Professor
Acknowledgments

First of all, I would like to thank my supervisor Kai-Yeung Sunny Siu for his great support, guidance, and vision. During the past four years, Sunny has provided me with a huge number of interesting and relevant problems to work on. Without them, I could not have written anything in this thesis. I have spent endless hours discussing and working with him at any hour of the day and of the night. His passion and dedication for his work are truly inspiring. He would always find time to talk to me, even when his schedule was totally booked and was sleeping less than half the hours I was. His willingness to work long hours and to help his students at all times has always impressed me. I have also always been impressed by the generous manner in which I have been treated. Sunny has always given me as much freedom as I needed, and has motivated me to work by making me enjoy what I was doing. At the academic level, Sunny has given me the theoretical rigor I previously lacked. He has taught me how to look at technical topics from a fundamental viewpoint and to understand the real issues of important problems in the field. For all these things, I am forever grateful to him.

I would also like to thank Henry Tzeng, who was my supervisor for three consecutive summers at Bell Labs. The main topic in this thesis is a result of an early problem which he asked me to work on during my first summer internship. After that, he kept feeding me with relevant problems that kept arising at his research group. Without him, I would have lost touch with real-world problems and would have probably ended up working on much more boring topics.

My colleagues at MIT and Bell Labs have been extraordinarily helpful to the completion of this thesis. Anthony Kam and I shared a similar academic trajectory and timetable. We shared many of the experiences of completing a PhD. To date, he is the only person I know who has been capable of solving all my old qualifier problems. David Lecumberri worked with me on a few recent papers. Kayi Lee and Ching Law were the smartest algorithms guys, and it was a real pleasure to discuss advanced algorithms with them. Thit Minn, who knows everything about
wireless, taught me much about recent developments in his field. Richard Rabbat was the most responsible person in the lab, helped me considerably in setting up the computer network for our group. At Bell Labs, Andrea Francini hosted me for three summers and gave me much support in pursuing my research interests.

Craig Barrack deserves a special mention for having proofread parts of this thesis, just as he did with my master’s thesis. He is one of the few people in this world (the only one?) who will read and scrutinize anything I write.

I am also extremely thankful to all the teachers that I have had since my first years in school. Without the likes of Batoul Mohtassem, Andrew Mosedale, Neven Dragojlovic, George Staples, and Robert McNamara I would have never made it this far.

My sister gets many thanks for being a great sister, always available when I needed her.

Finally, my deepest gratitude goes to my mother and father, who have made everything possible. Besides my early years, they have had to make great sacrifices to keep me so far from home. They would have loved seeing me closer to them, but have always insisted in me pursuing my own interests for my own good. They have given me all the love, support, and encouragement that I needed to do what I most loved to do.
Contents

1 Introduction .................................................. 12
  1.1 Background .............................................. 13
  1.2 Link-State Protocols .................................... 15
  1.3 Problem Formulation ..................................... 16
  1.4 Shortest Path Recomputation ......................... 17
  1.5 Local Routing Restoration ............................ 18
  1.6 Multiple Path Algorithm ............................... 21

2 New Dynamic Algorithms for
   Shortest Path Tree Computation ......................... 24
  2.1 Background and Motivation ............................ 25
  2.2 Related Works .......................................... 28
  2.3 Definitions and Notations ............................. 30
  2.4 Algorithmic Framework ................................. 31
     2.4.1 Informal Discussion of the Algorithm .......... 35
     2.4.2 Multiple Link Weight Changes .................. 36
  2.5 Special Cases of the Basic Algorithm ............... 37
     2.5.1 Bellman-Ford Algorithm ......................... 37
     2.5.2 D’Esopo-Pape Algorithm ......................... 37
     2.5.3 Dijkstra Algorithm ............................... 38
  2.6 First Incremental Method ............................. 38
  2.7 Second Incremental Method ........................... 44
  2.8 Algorithmic Complexity ............................... 46
2.8.1 First Incremental Method ........................................ 48
2.8.2 Second Incremental Method ...................................... 50
2.9 Simulation Results .................................................. 51
  2.9.1 Random Network Generator .................................... 51
  2.9.2 Event Generation .................................................. 52
  2.9.3 Node Based Complexity .......................................... 53
  2.9.4 Degree Based Complexity ....................................... 56
  2.9.5 Further Discussion on Dijkstra Algorithms .................. 59
2.10 Proofs of Theoretical Results ................................... 63
  2.10.1 Correctness of the Basic Algorithm ......................... 63
  2.10.2 Algorithmic Complexity ....................................... 68
2.11 Chapter Summary ................................................... 72

3 New Dynamic SPT Algorithm based on a Ball-and-String Model 73
  3.1 Background and Motivation ...................................... 74
  3.2 Example ............................................................ 75
  3.3 Problem Formulation .............................................. 77
    3.3.1 Shortest Path Tree ......................................... 78
    3.3.2 Linear Programming Framework ............................. 78
    3.3.3 Physical Interpretation of SPT in a Ball-and-String Model .. 80
  3.4 Dynamic SPT Algorithm .......................................... 81
    3.4.1 Algorithm Execution in the Ball-and-String Model ........ 81
    3.4.2 Algorithm Specification ..................................... 83
    3.4.3 Informal Discussion of the Algorithm ..................... 88
  3.5 Algorithm Correctness and Complexity ........................ 90
    3.5.1 Single Edge Weight Change ................................. 93
    3.5.2 Multiple Edge Changes ...................................... 95
    3.5.3 Complexity ................................................... 98
  3.6 Simulation Results ............................................... 100
    3.6.1 Event Generation ............................................ 101
5.7 Chapter Summary ........................................ 147

6 Summary .................................................. 148
   6.1 Dynamic SPT Algorithms ............................ 148
   6.2 Local Routing Restoration .......................... 150
   6.3 Multiple Path Algorithms ........................... 150
List of Figures

2-1 Multiple SPTs in a simple network. .......................... 27
2-2 Initialization after a link failure. .......................... 39
2-3 New SPT using First Incremental Bellman-Ford. ............... 40
2-4 New SPT using First Incremental D’Esopo-Pape. ............... 41
2-5 New SPT using First Incremental Dijkstra. .................... 42
2-6 Initialization after a link cost improvement. .................. 43
2-7 New SPT using First Incremental Dijkstra. .................... 43
2-8 New SPT using 2nd Inc Dijkstra (after a link failure). ....... 46
2-9 New SPT using 2nd Inc Dijkstra (after a link cost improvement). .... 47
2-10 Sample network of size 620 and average degree 7. ............ 53
2-11 Complexity of static algorithms. ............................ 54
2-12 Complexity of algorithms using the First Incremental Method. .... 55
2-13 Complexity of algorithms using the Second Incremental Method. .... 56
2-14 Complexity of Dijkstra-based algorithms. ..................... 57
2-15 Complexity of static algorithms. ............................ 58
2-16 Complexity of First Incremental algorithms. ................... 58
2-17 Complexity of Second Incremental algorithms. .................. 59
2-18 Complexity of Dijkstra Algorithms. ........................... 60
2-19 Average complexity of Incremental Dijkstra algorithms. ........ 61
2-20 Worst-case complexity of Incremental Dijkstra algorithms. ....... 61
2-21 Changes in distance attributes vs. parent attributes using Dijkstra Algorithms. ......................... 62
2-22 Number of visited nodes in Dijkstra Algorithms. ................ 63
Comparative complexity of incremental worst cases with static method.

Shortest path tree before link change.

Shortest path tree after link change.

The ball-and-string models for the networks in Fig. 3-1 (left) and Fig. 3-2 (right).

Complexity of static algorithms.

Complexity of dynamic algorithms.

Tunneling packets from A to B.

Topology subject to routing loops.

Topology subject to routing loops when there are 2 link failures.

Different sets of loop-less paths that can be taken by packets.

Multiple possible paths to destination.
List of Tables

3.1 Observed algorithm behavior after a link failure. ........................ 104

5.1 MPA data structure for each destination node. ........................... 141
Chapter 1

Introduction

This thesis will focus on improving the degree of adaptiveness of IP routing to network changes. In IP networks, routing protocols are responsible for constructing the path that a packet follows while it travels from its source to its destination. Each router in the network is responsible for forwarding the packet to its next-hop destination. Packet forwarding is performed by each router independently based on its own routing table. Routing protocols are employed to exchange topology information among routers to facilitate the construction of these routing tables.

When the topology of the network changes, routing protocols need to rebuild the routing tables to ensure that packets are forwarded correctly. The speed with which a network can rebuild its routing tables and adjust to changes in the topology is critical for the efficient performance of IP networks. If the readjustment interval is too long, packets can be lost during this interval. Furthermore, a slow system can produce routing instability and route flapping. On the other hand, fast routing readjustments not only prevent these problems, but also enable improvements in the quality of service (QoS) that the network might provide. A quick system response allows the routing protocols to react to more network parameters such as link congestion and thus can provide better overall routing.
1.1 Background

Routing in today's Internet is a complex topic. Because of the decentralized and heterogenous nature of the Internet, many different routing protocols are deployed in different networks. Global connectivity is ensured through the hierarchical organization of these networks.

For administrative purposes, the Internet is divided into different Autonomous Systems (AS). Each AS consists of a self-sufficient network which can route packets anywhere within itself. An AS is free to deploy whichever routing protocol it wants for internal purposes. Routing protocols which satisfy this function of routing packets within autonomous systems are called Interior Gateway Protocols (IGP).

In order to route packets from one AS to another, every AS that wants to be connected to the Internet must first "aggregate" the routing information about its own network and then transmit this information to other autonomous systems. This function is performed by exterior gateway protocols such as BGP. Exterior gateway protocols are generally very slow and very inflexible. Oftentimes, it might take hours or even days for a major routing change to propagate throughout the entire Internet. Because the performance of exterior gateway protocols greatly depends on manual configurations and policy agreements between different networks and service providers, not much can be done to improve their performance without enforcing new standards and disrupting the heterogeneity of the Internet. Besides, in the current Internet paradigm, EGP routing is meant to be slow and quasi-static. Routes should only be added or torn down when new networks are created or removed, or when there are major changes to the structure of the backbone. This prevents local configuration errors and small network perturbations from flooding too much information into the entire Internet.

On the other hand, IGPs are used exclusively within a single AS. This gives the administrative domain a great deal of control and flexibility in choosing the type of protocol it needs for internal routing purposes. Furthermore, new routing protocols can be deployed within a single AS without having to be deployed throughout the
rest of the Internet.

There are two broad classes of IGPs: distance-vector and link-state protocols. Distance-vector protocols use a distributed shortest path algorithm, such as the distributed Bellman Ford algorithm, to compute the forwarding table at each router. On the other hand, in link-state protocols, routers share information on the state of the network, but perform all routing computation locally in a centralized manner.

Distance-vector protocols are noticeably simpler than link-state protocols but have generally very poor convergence properties. The most commonly used distance-vector protocol is the Routing Information Protocol (RIP) [12, 17, 18, 19]. Apart from the slow convergence problem, RIP suffers from important transient loop problems. When a change in network topology takes place, RIP will recalculate routes very slowly, oftentimes having to count to infinity. During the long transient period, routing loops can occur. New proposed distributed routing algorithms [28, 11] do solve some of RIP’s problems (routing loops are prevented). However, all distributed routing protocols still exhibit slow convergence properties. For an excellent discussion on the inadequacies of RIP and other distributed protocols for large networks see Chapter 4 in [13].

On the other hand, link-state protocols, while being much more complex to implement, have much better convergence and stability properties. Link-state protocols obtain topology information in a distributed fashion. However, all the route computation is performed internally, and therefore, changes in the routing table can be performed quickly and efficiently. For these reasons, link-state protocols such as Open Shortest Path First (OSPF) [22, 23] and IS-IS [14, 26] are becoming the dominant protocols on most autonomous systems.

Because EGP routing is not intended to be dynamic or highly adjustable and should probably stay this way, it will not be covered in this thesis. Similarly, although some of the concepts that we will discuss are also applicable to distance-vector protocols, we will only consider their application to link-state protocols.

The new concepts and mechanisms discussed in this thesis will follow the link-state routing paradigm. The new algorithms and protocols will be directly applicable to
existing link-state protocols such as OSPF or IS-IS. Nevertheless, they will oftentimes have applications in other fields as well.

1.2 Link-State Protocols

Within an AS, a link-state protocol can also have its own hierarchical levels. For scalability reasons, large autonomous systems can partition their network into different routing areas. The link-state paradigm is only applied within each of these areas. Routing between areas is similar to that which takes place between different autonomous systems.

Inside an area, each link is associated with a cost (weight) and routers exchange link state information so that each router in this area has a complete description of the area’s network topology. This internal description is called the link-state database. Using the link costs, each router computes a path with minimum cost from itself to each other router in the area, yielding a shortest path tree (SPT). The corresponding SPT is then used to build a forwarding (or routing) table which contains routing information to forward a packet towards its destination along the shortest path.

The main characteristic of link-state protocols is the separation of the link-state database from the actual shortest path computation. The database is constructed and updated distributively and contains complete topological information about the routing area. A copy of this database is kept at each router. Each router then uses any centralized shortest path algorithm to calculate the best next hop for every destination.

When the topology in a routing area changes (e.g., a link fails, recovers, or changes its routing cost), every router in the region is notified of the change. After updating the corresponding topology changes in its link state database, each router recomputes its SPT. This recomputation is generally done by deleting the current SPT and recomputing it from scratch.
1.3 Problem Formulation

There are three main problems that we wish to identify regarding the way in which link-state routing protocols operate today.

First of all, at the most local level and shortest time scale, a routing table generally contains only one next-hop entry for every destination. In some implementations, a routing table may contain more than one next hop only in cases where there are multiple shortest paths. In most implementations, the routing table contains only one next hop for every destination. This means that when a neighboring link fails, the forwarding engine has to wait until the routing table is recomputed. In many cases, this might mean a delay of several seconds. On the other hand, if the routing table could contain several possible next-hops for every destination, the forwarding engine could automatically switch to another next-hop without having to wait for the routing table to be recomputed.

Second, at a somewhat larger time scale, the recomputation of routing tables is generally done by static SPT algorithms. This means that all the previous information on the previous SPT is deleted and needs to be recomputed from scratch. On the other hand, if a dynamic algorithm could be used instead, recomputation after slight topological changes could be done much more efficiently by using information on the previous SPT.

Finally, at an even larger time scale, current link-state databases need to be synchronized over the entire routing area. This means that after a link-state change, even distant routers in the area need to be notified of changes in the topology. For large areas, this entails a significant communication overhead, even though what is happening in a remote part of the area might not affect many routing tables at all. Significant overhead savings and speed improvements would be achieved if the routing information and reconfiguration were kept at a local level.

In the next three sections, we elaborate over these three areas and we introduce our proposed solutions.
1.4 Shortest Path Recomputation

As mentioned earlier, when the topology in a routing area changes every router in the region is notified of the change. After updating its link state database, each router recomputes its SPT. Today, this is generally done by recomputing the SPT from scratch with the well known Dijkstra algorithm [7].

Usually, after some changes in the link states, the topology of the new SPT does not differ significantly from the old one. In fact, most often it does not change at all. Static algorithms that recompute the SPT from scratch are clearly inefficient because they do not take advantage of available information about the outdated SPT. Such recomputation can consume a considerable amount of CPU time, preventing other critical routing functions from being executed. Thus, it is desirable to compute the SPT using as little CPU time as possible.

To cope with the complexity of SPT computation, the traditional approach has been simply to limit the size of the routing area (e.g. an OSPF area). However, there is a growing interest among network operators to increase the size of a routing area in order to increase redundancy and optimize routing. By reducing the complexity of the SPT computation and thus eliminating this performance bottleneck, a larger routing area can be allowed.

Another drawback of using static SPT algorithms is that there may coexist multiple routes of the same shortest distance from one router to another; by recomputing a new SPT from scratch, a router may unnecessarily choose a different route of the same distance to forward its packets. This in turn may cause the router to change many entries in its forwarding table frequently, increasing the risk of routing errors or router failures. In addition to redundant updates in forwarding tables, unnecessary changes in SPT also cause undesirable fluctuation of traffic load on a given route.

In Chapter 2 we will explore a rich class of algorithms that can dynamically update the SPT following changes in the link states. These dynamic algorithms use information from the outdated SPT and update only the part of the SPT that is affected by the change. Our design objectives for these dynamic algorithms are
twofold. The first objective is to minimize the computational complexity required
to update an SPT. The second objective is to maintain routing stability by making
minimal changes to the structure of an existing SPT.

Unlike previous work on dynamic SPT algorithms that is based on the static
Dijkstra algorithm only, we present an algorithmic framework that also yields dy-
namic versions of other well-known static SPT algorithms such as Bellman-Ford and
D'Esopo-Pape [3]. This framework allows us to characterize dynamic SPT algorithms
in a unified way and to establish proofs of correctness for these algorithms. In particu-
lar, within our framework, we propose two different incremental methods to transform
static algorithms into new dynamic algorithms. The first method when applied to the
Dijkstra algorithm yields a dynamic algorithm similar to existing ones [10, 9, 15]. The
first method can also be used to transform other static algorithms (i.e., Bellman-Ford)
into their dynamic versions. The second incremental method yields new dynamic algo-
rithms that are faster on average. Furthermore, with the second incremental method,
the resulting dynamic algorithm will make the minimum number of changes to the
SPT structure following a link-state update, thus improving routing stability.

Using this algorithmic framework, in Chapter 3 we develop a single new dynamic
SPT algorithm. It uses an original search criterion, not derived from any existing stat-
ic SPT algorithm, and is provably better than the existing dynamic SPT algorithms.
The algorithm is also heavily based on the dual linear programming technique and
on its physical interpretation. This physical interpretation uses balls and strings to
model the system of constraints involved in computing an SPT. When using simple
data structures, this algorithm also has the lowest asymptotic complexity of any other
known algorithm.

1.5 Local Routing Restoration

In link-state protocols the process of collecting topological information from the net-
work is separated from the actual process of computing the correct routes. The former
is performed distributively by all the routers in an area who share information with
each other on the state of the links. Once a given router has the topological information on the routing area, it will construct a routing table based on the shortest path tree (SPT) that it computes for that area.

The main advantage of link-state protocols is that the routing computation is done exclusively inside the router. This means that the computation can be performed quickly and without relying on what other routers are doing. When there is a change in the cost metric of some link, the new information needs to be sent only once to every router in the area; the routing tables will be updated to their correct values immediately. This is in sharp contrast to distance-vector protocols, such as RIP [12, 17], where multiple routing packets may have to be sent many times between the same routers in order for their routing tables to converge to a steady-state and correct value.

Nevertheless, even though the amount of information that link-state protocols need to transmit is relatively small (compared to distance-vector protocols), it might still be too large when the cost metrics of the links vary too quickly or when the number of links in a routing area is too large. In principle, every router in an area should have at all times the latest information on the topology of the area. This topology information should be identical in all routers in order to avoid routing loops. This means that every time there is a cost metric change in one of the links, all the routers in the routing area must be notified. Flooding an entire area after a single link-state change is often unnecessary and is inefficient in terms of bandwidth and computation overhead required. Furthermore, flooding a large routing area might require a large amount of time. During the time that the new information is being propagated, different routers will have different link-state information, and therefore, transient routing loops are possible. Many of the Internet's problems with routing instability are associated with the long delays required to propagate routing information [16, 25, 27].

One way to reduce the total amount of routing information being transmitted is to have the router responsible for a given link to simply ignore changes in that link's metric. The new information will simply not be propagated and routing will continue along the old paths in a sub-optimal manner. After several metric changes,
the router can broadcast an update packet summarizing all the topological changes that have taken place. By limiting the frequency of such updates, the amount of network information traffic can be limited at the cost of some sub-optimal routing between update intervals.

However, when the link change that takes place is that of a link failure (cost metric becomes infinity), it is no longer possible to withhold this information from the rest of the network. Doing so would cause routing failure and packet loss. Therefore, information on the failure of a link must be propagated to some routers in the network in order to restore the paths that were traversing that link.

In Chapter 4 we will present two routing protocols to limit the routing information that needs to be propagated in a link-state protocol when any network link fails. The schemes will restore all the paths traversing the failed link by only performing local updates on the neighboring routers. The object of our scheme is to restore loop-free routing after a link failure, while spreading information about that failure to as few routers as possible. This is particularly important for very large routing areas where it is not desirable to disturb a huge number of routers when just a single link fails.

This approach is also useful to divert traffic when a link becomes particularly congested. In current link state protocols such as OSPF, it is not practical to change the metric of a link every time its congestion state changes. Because of the large amount of time required to flood an entire routing area with new information, changes in link cost-metrics take place in OSPF only when absolutely necessary. Since our approach does not require flooding, a link that is congested can temporarily advertise itself as being down and thus can temporarily divert traffic through a local alternative path. Because in our scheme very few routers need to be informed, this operation can be done quickly and as frequently as needed. The implementation of our schemes removes the major obstacle (global flooding) that currently prevents frequent topological updates to be performed.
1.6 Multiple Path Algorithm

Although a router using a link-state routing protocol has global information about the area network topology, the router itself only determines the local route of a packet — the next hop. When each router computes such next-hop information based on an SPT, then each packet is guaranteed to be forwarded along a loop-free path to its destination. If multiple SPTs rooted at router $S$ co-exist in a network, there can be multiple choices of next hops from $S$ to forward a packet, each of which will lead to a different shortest path of the same distance. In this case, the packet can be forwarded to any of these next hops, which would also result in a loop-free path. In some current implementation of OSPF, it is possible to build the routing table so that there might be more than one next hop for every destination. Each one of these next hops corresponds to the first hop of one of the multiple shortest paths to the destination.

In Chapter 5 we will present a new algorithm which generalizes this idea by allowing a router to forward packets to multiple viable next hops that are not necessarily part of a shortest path from the router to the packet’s destination. The essence of this algorithm is the implementation of an efficient data structure in a router that helps compute alternate viable next hops for each destination. When each router forwards its packets to any of the viable next hops, all packets will be routed to their destinations on loop-free paths. Even though the resulting path is not the shortest, the packet is guaranteed to travel loop free. Moreover, a degree of optimality is determined for each viable next hop. Different degrees of optimality yield paths of different distances.

An advantage of maintaining multiple viable next hops in a router for each destination is that recovery from an adjacent link failure can be performed much faster. A router supporting the new algorithm can recover from the adjacent link failure much faster since the recovery mechanism using alternate paths can be implemented locally by the forwarding engine in the router, and there is almost no delay incurred by the link failure. This is done by immediately using a different possible next hop, when
the primary next hop is not reachable.

Moreover, network capacity can be better utilized by balancing loads on different paths. However, some additional schemes may need to be implemented to preserve the FIFO arrival order of the packets if such property is assumed by the upper layer protocol at the destination. Otherwise, the destination may need a larger buffer and more processing to reassemble the packets delivered by different paths.

Another important application of our algorithm is Quality-of-Service (QoS) routing. Since some QoS parameters that need to be optimized may depend on the network traffic load or other time-varying factors and, finding global optimal path might be infeasible even for a moderate-size network. Moreover, even if some QoS parameters do not change frequently, determining the optimal path from a router may require global knowledge of these parameters in the network, and a local change in these parameters would therefore need to be disseminated to every node in the network. Hence, in most cases only suboptimal paths can be computed.

The new algorithm can be used to determine good approximate solutions to QoS routing by decoupling the underlying optimization problem into local and global computations. First, multiple viable paths that satisfy the global or less frequently changed QoS constraints are computed. Then, among these viable paths, one that is optimized with respect to the local or more frequently changed QoS parameters can be determined. Although such a heuristic approach does not always yield the optimal path, in many cases a satisfactory path can be found.

Another desirable property of our scheme is that the algorithm and its data structure can be efficiently implemented as an add-on component to existing link-state routing protocols such as OSPF. It only adds a small constant overhead to the shortest path tree computation of the underlying routing protocol. Each enhanced multiple next-hop router computes multiple paths based on the topology information exchanged by the routing protocols.

Conventional routers based on shortest path routing can co-exist with enhanced multiple next-hop routers in the same routing area and still guarantee loop-free routing for each packet. Such interoperability is a consequence of the loop-free routing
policy employed by our scheme. In general, we will show how a router using our scheme can interoperate with any router whose routing protocol uses the loop-free routing policy.
Chapter 2

New Dynamic Algorithms for Shortest Path Tree Computation

The OSPF and IS-IS routing protocols widely used in today’s Internet compute a shortest path tree (SPT) from each router to every other router in a routing area. Many existing commercial routers recompute an SPT from scratch following changes in the link states of the network. Such recomputation of an entire SPT is inefficient and may consume a considerable amount of CPU time. Moreover, as there may coexist multiple SPTs in a network with a set of given link states, recomputation from scratch causes frequent unnecessary changes in the topology of an existing SPT and may lead to routing instability. In this chapter, we present new dynamic SPT algorithms that make use of the structure of the previously computed SPT. Besides efficiency, our algorithm design objective is to achieve routing stability by making minimum changes to the structure of an existing SPT (while maintaining shortest path property) when some link states in the network have changed.

We establish an algorithmic framework that allows us to characterize a variety of dynamic SPT algorithms including dynamic versions of the well-known Dijkstra, Bellman-Ford, and D’Esopo-Pape algorithms, and to establish proofs of correctness for these algorithms in a unified way. The theoretical asymptotic complexity of our new dynamic algorithms matches the best known results in the literature. Extensive simulation results are also presented to show that our new dynamic algorithms...
outperform existing algorithms in most cases.

2.1 Background and Motivation

In today’s Internet, each datagram is forwarded by a router based on a forwarding table. Routing protocols are employed to exchange topology information among routers to facilitate the construction of forwarding tables. Examples of widely used link-state routing protocols include Open Shortest Path First (OSPF) and IS-IS [22, 23, 14, 26]. In these routing protocols, each link is associated with a cost (weight) and routers exchange link state information so that each router in a routing area (e.g., an OSPF area) has a complete description of the network topology. Using the link costs, each router computes a path with minimum cost from itself to each other router in the area, yielding a shortest path tree (SPT). The corresponding SPT is then used to build a forwarding table which contains routing information for forwarding a datagram to its destination along the shortest path.

When the topology in a routing area changes (e.g., a link fails, recovers, or changes its routing cost), every router in the area is notified of the change. After updating the corresponding topology changes in its data structure, each router recomputes its SPT. In most of today’s commercial routers, this recomputation is done by deleting the current SPT and recomputing it from scratch by using the well known Dijkstra algorithm [7].

Usually, after some changes in the link states, the topology of the new SPT does not differ significantly from the old one. In fact, oftentimes it does not change at all. Static algorithms that recompute the SPT from scratch are clearly inefficient because they do not take advantage of available information about the outdated SPT. Such recomputation can consume a considerable amount of CPU time, preventing other critical routing functions from being executed. Thus, it is desirable to compute the SPT using as little CPU time as possible.

To cope with the complexity of SPT computation, the traditional approach has been simply to limit the size of the routing area (e.g. an OSPF area). In fact, it has
been recommended that the size of an OSPF area be limited to 200 routers, based on the cost of the SPT computation [24]. Today, most router vendors limit the OSPF area to a size between 50 and 500 routers.

However, the downside of having smaller areas is that networks are more fragmented. Routers in different areas can only communicate with each other through the few gateways that exist between the different areas. In OSPF, these gateways are called Area Border Routers (ABR). These constraints prevent routing from being fully optimal since inter-area paths are constrained to go through a border router. Moreover, the border routers whose task is to forward traffic from one area to another are excessively loaded and become single points of failure. There is a growing interest among network operators to increase the size of a routing area in order to increase redundancy and optimize routing. By reducing the complexity of the SPT computation and thus eliminating this performance bottleneck, a larger routing area can be allowed.

Another drawback of using static SPT algorithms is that there may coexist multiple routes of the same shortest distance from one router to another; by recomputing a new SPT from scratch, a router may unnecessarily choose a different route of the same minimum distance to forward its packets. This in turn may cause the router to change many entries in its forwarding table frequently, increasing the risk of routing errors or router failures.

To illustrate this point, Figure 2-1 shows a routing area of an Internet Service Provider (ISP) consisting of four routers A, B, C, and D. The routers A, B, and D are access routers connecting to the customers, and router C connects to the Internet backbone. Initially, the link between D and C is down. The thick arrows indicate an SPT rooted at router A. The forwarding table in A contains many entries, each of which corresponds to the next hop information for each network reachable via C in the Internet. All routes from A to the Internet will go through B initially. However, if the link between D and C is recovered with a link cost of 1, then the SPT from A is not unique. When the SPT is recomputed from scratch, the result may indicate that all routes to the Internet from A should now go through D. Router A will then need
to change the next-hop information for all networks reachable via C. These changes may correspond to several thousand updates in the forwarding table of router A.

In addition to redundant updates in forwarding tables, unnecessary changes in the SPT also cause undesirable fluctuation of traffic load on a given route. (For an excellent discussion on instability of other routing protocols in the Internet, see [16, 25].)

In this chapter, we will explore a rich class of algorithms that can dynamically update the SPT following changes in the link states. These dynamic algorithms use information of the outdated SPT and update only the part of the SPT that is affected by the change. Our design objectives for these dynamic algorithms are twofold. The first objective is to minimize the computational complexity required to update an SPT. The second objective is to maintain routing stability by making minimal changes to the topology of an existing SPT.

The purpose of our work is restricted to dynamic SPT algorithms that can be used in link-state protocols. Our algorithms need to be executed in a centralized processor and require an a priori complete topology database (such as the link-state database in OSPF). There are other routing algorithms in the literature [11, 28] which use tree-like structures. However, these algorithms are distributive and operate under very different assumptions. Comparisons and analogies between such distributed routing algorithms and algorithms where all path calculation is centralized (like ours) are beyond the scope of this thesis.
Unlike previous work on dynamic SPT algorithms that is based on the static Dijkstra algorithm only, we shall present an algorithmic framework that also yields dynamic versions of other well-known static SPT algorithms such as Bellman-Ford and D'Esopo-Pape. This framework allows us to characterize dynamic SPT algorithms in a unified way and to establish proofs of correctness for these algorithms. In particular, within our framework, we propose two different incremental methods to transform static algorithms into new dynamic algorithms. The first method when applied to the Dijkstra algorithm yields a dynamic algorithm similar to existing ones [10, 9, 15]. The first method can also be used to transform other static algorithms (i.e., Bellman-Ford) into their dynamic versions. The second incremental method yields new dynamic algorithms that are faster on average. Furthermore, with the second incremental method, the resulting dynamic algorithm will make the minimum number of changes to the shortest path tree topology following a link-state update, thus improving routing stability.

In the next section, we shall further discuss some prior works. Section 2.3 introduces graph-theoretic definitions and notations to be used in this chapter and the following one. In Section 2.4, we describe our algorithmic framework by way of a basic algorithm for computing a shortest path tree. Section 2.5 explains how the well known static shortest path tree algorithms can be characterized in our algorithmic framework. Sections 2.6 and 2.7 describe two specific methods for implementing the basic algorithm, which convert static algorithms into dynamic algorithms. Section 2.8 discusses theoretical bounds on the asymptotic computational complexity of each specific dynamic algorithm. Section 2.9 presents simulation results illustrating the computational complexity of each dynamic algorithm. The proofs of the theoretical results are presented in Section 2.10.

2.2 Related Works

The problem of routing in data networks has been a subject of continual research interest for the past two decades [1, 5, 11, 13, 27, 28, 30, 32] and many routing protocols
have been studied and used in practical networks [6, 20, 26, 29, 34]. Recently, the
stability issues in Internet routing have attracted much attention [16, 25]. In fact, our
interest in dynamic SPT algorithms was partly motivated by the problem of routing
instability in the Internet.

To the best of our knowledge, the earliest work on dynamic SPT algorithms that
appears in the literature is an article by Spira [31]. The work proves a lower bound
complexity for the worst case scenario. The first efficient dynamic SPT algorithm
that we know of is presented by McQuillan et al. [20]. This algorithm is very similar
to the First Incremental Dijkstra algorithm that we present here, though the work [20]
contains no analytical proofs nor simulation results.

Two semi-dynamic algorithms to update SPTs are presented by Franciosa et al. [8].
Each semi-dynamic algorithm handles the case when the change in an edge weight
is either positive or negative. The algorithms are dynamic versions of Dijkstra's al-
gorithm, but can only handle integer edge weights. The paper only analyzes the
worst-case complexity of the algorithm in terms of the total size of the graph rather
than the number of nodes whose distances have changed (denoted by $\delta_d$ in this chap-
ter). Therefore, the complexity cannot be shown to be better than that of a static
algorithm.

Frigioni et al. [10, 9] present an algorithm similar to the one in [8] but it allows
the edge weights to be non-integers. The algorithm is also a dynamic version of the
Dijkstra algorithm. It is quite similar to our First Incremental Dijkstra algorithm;
the main difference is that our algorithm optimizes the initialization procedure. The
complexity of the algorithm in [9] is analyzed in terms of the number of nodes whose
distance attributes change. Their algorithm also introduces some optimization for
cases where there is a large number of edges (where the node degree is not bounded)
in order to bound the worst-case complexity in such cases.
2.3 Definitions and Notations

We now define some notation to be used in the rest of the thesis. We assume that the reader is familiar with basic graph theoretic definitions. Let $G = (V, E)$ denote a directed graph where $V$ is the set of nodes and $E$ is the set of edges in the graph. Let $Source(G) \in V$ denote the root or source node of $G$.

For each directed edge $e \in E$, we use $W(e)$ to denote the weight (distance) associated with $e$, $S(e)$ and $E(e)$ to denote respectively the source node and the end node of $e$. The length or distance of a directed path is the sum of weights of the edges on the path. Given a set of nodes $N \subseteq V$, we associate with it two sets of edges: $I(N) = \{e \in E \mid E(e) \in N\}$ (the set of edges directed into the nodes in $N$) and $O(N) = \{e \in E \mid S(e) \in N\}$ (the set of edges directed out of the nodes in $N$). These parameters only depend on the topology of the network, and their values do not change during the execution of the algorithm.

A rooted tree $T$ is a subgraph of $G$ such that $Source(G)$ is in $T$ and every node in $T$ is reachable from $Source(G)$ through a unique directed path using only edges in $T$. A node $x$ is a parent node of $y$ in $T$ if $x$ is the source node and $y$ is the end node of an edge in $T$. We associate with each node $n$ in a tree $T$ the following attributes: $P(n, T)$ is the parent node of $n$ and $D(n, T)$ is the distance attribute of $n$. Since $T$ is a tree, invoking $P(n, T)$ recursively determines a unique path from $Source(G)$ to any node in $T$.

The descendents of a node $n$ in $T$ are all the nodes that are reachable by $n$ through directed edges in $T$. We use $B(n, T)$ to denote a subset that includes $n$ and some descendents of $n$ in tree $T$. As we will see later, an execution of our algorithm involves the distance update of the subset $B(n, T)$ for various nodes $n$. The different ways the subset $B(n, T)$ can be chosen for each node $n$ will lead to different incremental methods for computing the SPT. For example, $B(n, T)$ can be chosen to be simply $n$ itself, the children (the next generation) of $n$ with itself, or all its descendents with itself in tree $T$. Specifically, the notation $B_{\text{max}}(E(e), T)$, which is always used in the initialization of the algorithm denotes the set including the end...
node of edge $e$ and all its descendents in $\mathcal{T}$.

A shortest path tree (SPT) in $\mathcal{G}$ is a rooted tree such that the length of every path from $Source(\mathcal{G})$ to every node in $\mathcal{G}$ is minimized. Since there can be different paths of the same minimum length between two nodes, an SPT in $\mathcal{G}$ is not unique. However, it follows from the definition, that the length of the path from $Source(\mathcal{G})$ to any node $n \in \mathcal{G}$ in any SPT is unique.

2.4 Algorithmic Framework

In this section, we present a broad class of algorithms that can compute an SPT from a source node to every other node in the graph. This class of algorithms is presented in a unified framework as the basic algorithm. Depending on the initialization procedure (see pseudocode), the basic algorithm includes as special cases some well-known static SPT algorithms, such as Bellman-Ford, Dijkstra, and D’Esopo-Pape as well as their dynamic versions. The proof of correctness of the basic algorithm in computing an SPT will be presented later in Section 2.10.

A tree data structure, denoted by $\mathcal{T}$, is maintained by the basic algorithm to keep track of an existing potential shortest path tree. In particular, every node $n$ in the graph $\mathcal{G}$, along with its parent attribute $P(n, \mathcal{T})$ and distance attribute $D(n, \mathcal{T})$ is present in $\mathcal{T}$. This data structure $\mathcal{T}$ changes progressively during the computation, and when the execution of the algorithm is completed, it will represent an SPT.

In addition to the data structure $\mathcal{T}$, the basic algorithm also maintains a list $Q$ that contains a subset of nodes and their parent and distance attributes temporarily. In particular, each element in $Q$ is of the form \{n, (p, d)\}, where $p$ and $d$ respectively denote the potential parent of $n$ and the potential distance of $n$ from the source node with respect to some tree. The instruction $\text{ENQUEUE}(Q, \{(n, (p, d))\})$ adds one more element to $Q$; if node $n$ is already in $Q$ with attributes $p_{old}$ and $d_{old}$, the new attributes $(p, d)$ will replace the old ones only if the new distance $d$ is less than $d_{old}$. At any instant, only one set of attributes is maintained for each node in $Q$. When an $\text{EXTRACT}(Q)$ instruction is executed, a single element (to be defined by
a specific algorithm) is selected and removed from $\mathcal{Q}$. Different implementations of $\text{ENQUEUE}(\mathcal{Q}, \{n, (p, d)\})$ and $\text{EXTRACT}(\mathcal{Q})$ in the basic algorithm yield different specific algorithms to be discussed later.

The basic algorithm also makes use of arithmetic using the symbol for infinity ($\infty$). By this term, we refer to any number that is much larger than the distance of the longest path in the network. It is also the largest number we can store. On most computers, this term would be the largest number that can be held by a variable type. For example if we perform the arithmetic operation $\infty - x - \infty$, the result would be $-x$. On the other hand, since no number can be larger than $\infty$, $\infty + x - \infty = 0$.

The basic algorithm contains an initialization procedure (step 1) and an iterative loop (steps 2-4). The following pseudocode formally describes the algorithm. An informal description of the algorithm will follow after the pseudocode.

### The Basic Algorithm

#### STEP 1: Initialization

(A) *Static Version*

$\forall (n \in \mathcal{V})$

- $P(n, \mathcal{T}) \leftarrow \emptyset$
- $D(n, \mathcal{T}) \leftarrow \infty$

$\text{ENQUEUE } (\mathcal{Q}, \{\text{Source} (\mathcal{G}), (0, 0)\})$

(B) *Dynamic Version* (applicable when an outdated SPT exists)

Case 1 (Edge $e$ increases its weight by $\Delta$):

- $W(e) \leftarrow W(e) + \Delta$
- if $e$ is in $\mathcal{T}$
  
  $\forall n \in \mathcal{N} = B_{\max}(E(e), \mathcal{T})$
\[D(n, \hat{T}) \leftarrow D(n, \hat{T}) + \Delta\]
/* increase the distance of each descendent of \(E(e)\) and itself by \(\Delta\) */
\[
\forall e' \in \mathcal{N} \quad /* consider edges directed into descendents of \(E(e)\) and itself */
\quad \text{if } D(E(e'), \hat{T}) > D(S(e'), \hat{T}) + W(e')
\quad /* if the increased distance of a node is larger than its distance in another path */
\quad \text{newdist} \leftarrow D(S(e'), \hat{T}) + W(e')
\quad /* choose the smaller distance as the new potential distance */
\quad \text{ENQUEUE}(Q, \{E(e'), (S(e'), \text{newdist})\})
\quad /* update the node with the new distance and new parent in the list \(Q\) */

Case 2 (Edge \(e\) decreases its weight by \(\Delta\)):
\[W(e) \leftarrow W(e) - \Delta\]
\text{if } D(S(e), \hat{T}) + W(e) < D(E(e), \hat{T})
/* if the reduced weight of \(e\) yields a smaller distance for \(E(e)\) */
\[
\Delta' \leftarrow |D(S(e), \hat{T}) + W(e)| - D(E(e), \hat{T}) \quad /* keep track of the difference */
\quad P(E(e), \hat{T}) \leftarrow S(e) \quad /* set the parent attribute of the end node */
\quad \forall n \in \mathcal{N} = B_{\text{max}}(E(e), \hat{T})
\quad D(n, \hat{T}) \leftarrow D(n, \hat{T}) + \Delta'
\quad /* reduce the distance of each descendent of \(E(e)\) and itself by the difference */
\quad \forall e' \in \mathcal{O}(\mathcal{N})
\quad /* consider all edges directed out of the descendents of \(E(e)\) and itself */
\quad \text{if } D(E(e'), \hat{T}) > D(S(e'), \hat{T}) + W(e')
\quad /* if the reduced distance of a node also reduces the distance of another node */
\quad \text{newdist} \leftarrow D(S(e'), \hat{T}) + W(e')
\quad /* choose the smaller distance as the new potential distance */
\quad \text{ENQUEUE}(Q, \{E(e'), (S(e'), \text{newdist})\})
\quad /* update the node with the new distance and new parent in the list \(Q\) */
STEP 2: Node Selection

if $\mathcal{Q} = \emptyset$

    Terminate

else

    \{y, (x, d)\} ← EXTRACT(\mathcal{Q}) /* select and remove an element from the list */
    \Delta ← (d - D(y, \hat{T})) /*compute the difference between new and old distances */
    if $\Delta \geq 0$ /* if there is no improvement in distance */
        Go to Step 2 /* discard this information and go back to step 2 */
    \P(y, \hat{T}) ← x /* otherwise, update the parent attribute of the selected node */
    \mathcal{N} ← \mathcal{B}(y, \hat{T})
/* consider the selected node and a selected subset of its descendents */

STEP 3: Distance Update

\forall n \in \mathcal{N}

    D(n, \hat{T}) ← D(n, \hat{T}) + \Delta
/* update the distance of the selected node and each descendent considered */

STEP 4: Node Search

\forall e \in O(\mathcal{N}) /* consider all edges directed out of the selected subset of nodes */

    if $D(E(e), \hat{T}) > D(S(e), \hat{T}) + W(e)$
/* if the reduced distance of selected node also reduces the distance of another node */

        newdist ← D(S(e), \hat{T}) + W(e)
/* choose the smaller distance as the new potential distance */

        ENQUEUE(\mathcal{Q}, \{E(e), (S(e), newdist)\})
/* update the node with the new distance and new parent in the list \mathcal{Q} */

Go to Step 2
2.4.1 Informal Discussion of the Algorithm

To help understand the basic algorithm, here we shall give a brief description of its executions. As the execution is the same for the static versions and the dynamic versions after initialization, we shall only discuss the dynamic version, since the initialization of the static version is trivial.

First, the goal of the initialization phase is to identify those nodes that may be affected by the change in the link state. The potentially affected nodes are those that are no longer connected to the root through the same shortest path as before. Moreover, in the initialization, we only want to select a minimal set of such nodes whose changes in distance will be subsequently propagated to their descendents in the original tree, which can be updated.

In case 1, after updating the changed edge \( e \) with the new increased weight, we check if the edge is in the existing SPT. If it is, we select its end node \( E(e) \) and all descendent nodes of \( E(e) \) that are reachable in the existing SPT. All such nodes will be included in a set \( N \) for further updating. The intuition is that the set \( N \) covers all the affected nodes. Now for each node that is attached to a link directed into a node in \( N \), we compute the potential distance \( \text{new dist} \) by adding the distance of the parent node and the weight of the edge connecting the two nodes.

In case 2, after updating edge \( e \) with the new decreased weight, we update the distance of all its descendents (the set \( N \)) using the same difference change \( \Delta' \). For every edge \( e \) originating from nodes in \( N \) we compute the potential new distance \( \text{new dist} \) of its end node \( E(e) \) by adding the distance of its source node \( S(e) \) and the new weight \( W(e) \). And if this potential new distance is in fact smaller than the old distance of \( E(e) \).

After the initialization step, we extract from the list \( Q \) in step 2 any element with the smallest (the exact procedure is determined by the specific algorithm). This selected node is then updated (if the new distance is better) with its new parent indicated in \( Q \) and the structure of \( \hat{T} \) is modified accordingly to reflect the new child-parent relation. Moreover, in step 3, the selected node together with any of its
descendent nodes (to be specified by the method used and denoted by the set \( \mathcal{N} \)) in the existing tree \( \hat{T} \) will now have their correct distances updated as their old distances incremented by \( \Delta \).

Finally, in step 4, we consider each node \( E(e) \) that is attached to an edge \( e \) directed out of a node \( S(e) \) in \( \mathcal{N} \). If the potential new distance \( \text{newdist} \) of \( E(e) \) (which is equal to \( W(e) \) plus the distance of \( S(e) \)) is smaller than its old distance, we then enqueue in the list \( Q \) the node \( E(e) \). After this step is completed, the execution of steps 2 and 3 will be repeated until the list \( Q \) is empty.

At each iteration, a new node and possibly some of its descendents are updated with a better distance and a better parent that reflects that distance. In this manner a change in a link-state will propagate along the old SPT until a new valid SPT is constructed.

### 2.4.2 Multiple Link Weight Changes

When there are several link weight changes occurring at once, it is always possible to run the algorithm sequentially for each weight changed. However, some optimization can be achieved by updating several of these changes together. Even though a separate initialization needs to be done for every weight change, the main body of the algorithm needs to be executed only twice. We now describe how such optimization can be realized.

First of all, we initialize all the weight increases. For each link whose weight increases, the first four lines of the code for Case 1 are executed. This ensures that all the affected nodes have updated their distance attributes in \( \hat{T} \). Then, we execute the next four lines of the code in Case 1 for a set \( \mathcal{K} \) that includes all the nodes previously updated. This ensures that all the necessary nodes are enqueued in the list \( Q \). The rest of the algorithm (steps 2-4) is then executed in a normal way.

After the SPT is updated with all the weight increments, we update it again for all the weight decrements. As before, the first six lines of the initialization (Case 2) are executed for each weight decrement, and then the last four lines are executed with a set \( \mathcal{N} \) which includes all the nodes previously updated. After the initialization, the
rest of the algorithm (steps 2-4) is executed normally.

2.5 Special Cases of the Basic Algorithm

Here we will show how the best known static algorithms can be viewed as special cases of our basic algorithm. For these cases, the static version of the initialization procedure in our basic algorithm is used. The branching function $B(n, T)$ in the static case is defined as

$$B(n, T) = \{n\} \quad \forall n \in \mathcal{V}.$$  \hspace{1cm} (2.1)

In this way, the new SPT is constructed in $\hat{T}$ by changing the attributes of one node at a time during each iteration. This method is used in several well known static algorithms for computing the SPT. The only differences between these algorithms consist in the different ways in which the temporary list $Q$ is implemented.

2.5.1 Bellman-Ford Algorithm

The simplest way of implementing the list $Q$ is by using a FIFO queue. New nodes and their attributes are enqueued at bottom of the queue and extracted from the top of the queue. When this queuing discipline is used in the static method, it gives rise to the Bellman-Ford algorithm [2]. We will refer to it as the static Bellman-Ford algorithm.

2.5.2 D’Esopo-Pape Algorithm

Another way of implementing $Q$ is by using a queue where (as in Bellman-Ford’s case) nodes are extracted from the top of the queue. However, unlike Bellman-Ford’s case, a node is enqueued at the bottom of the queue only when that node has not been enqueued in the list $Q$ before. If some node $n$ has already been inserted and extracted from $Q$, the next time when $n$ is inserted in $Q$ it will be at the top. When
the static version of the initialization procedure is used, this queuing discipline gives rise to the D'Esopo-Pape algorithm [3]. We will refer to it as the static D'Esopo-Pape algorithm.

2.5.3 Dijkstra Algorithm

Yet another way of implementing $Q$ is by using a priority queue where the node enqueued with the smallest distance attribute is always extracted first. When the static version of the initialization procedure is used, this algorithm is equivalent to the Dijkstra algorithm [7].

To implement the priority queue, one can use a linked list where inserted nodes can be placed anywhere. The node with the smallest distance attribute can then be extracted by searching through the linked list sequentially. The resulting algorithm will be referred to as the static Linear Dijkstra algorithm. A more efficient way to implement the priority queue is to use a binary heap data structure, and the resulting algorithm in this case will be called the static Heap Dijkstra algorithm.

2.6 First Incremental Method

The four static SPT algorithms presented in the preceding section can be transformed into dynamic algorithms by using the dynamic version of the initialization procedure of the basic algorithm. The branching function $B(n, T) = \{n\}$ is the same as in the static algorithms.

We call this method the first incremental method and the resulting algorithms “First Incremental Bellman-Ford,” “First Incremental D’Esopo-Pape,” “First Incremental Linear Dijkstra,” and “First Incremental Heap Dijkstra.”

We will illustrate with a number of figures how the first incremental algorithms work on a simple network. For simplicity, we assume each link in the network is bidirectional and have the same cost (indicated in bold-face numbers in Figure 2-2) in each direction. In these figures, the solid thick arrows between nodes represent the directed edges that are in the current SPT ($\hat{T}$). The thin dashed segments represent
other edges in the network that are not in \( \mathcal{F} \).

Figure 2-2 shows how the algorithm is initialized after a link failure (link cost becomes infinity) between nodes 1 and 7 (link \((1,7)\)). First link \((1,7)\) is removed from \( G \). The dotted enclosed area includes the set of nodes \( \mathcal{P} \) that are descendents of node 7. These are the nodes that might have their distance attributes increased as a result of the link failure. The nodes outside \( \mathcal{P} \) will be unaffected by the link failure. The shaded nodes denote the nodes in the germinal set (to be defined later in Section 2.10.1 when the theoretical results are proved) that are initially stored in \( Q \). These nodes are necessarily the point of entry into \( \mathcal{P} \) for all potential new shortest paths. This initialization procedure is the same for all the dynamic algorithms.

Figure 2-3 illustrates how the First Incremental Bellman-Ford algorithm recomputes the SPT. We assume the link distance is the same as in Figure 2-2 (except for the failed link) and for simplicity, the link distance is not indicated in Figure 2-3 and subsequent figures referring to the same network. The number next to each node now represents the order in which that node is extracted from \( Q \). Each number is also placed next to the link indicating the parent attribute with which it was extracted. Note that in Bellman-Ford's algorithm, the initial order of extraction from \( Q \) is arbi-
Figure 2-3: New SPT using First Incremental Bellman-Ford.

Our example indicates that node 14 is the first node extracted from Q, and its parent attribute is node 17. The next node extracted is node 9 with parent attribute node 4.

Note that with this algorithm, nodes 11, 12, and 14 are extracted more than once. Since a node might be extracted with incorrect attributes, it will need to be enqueued and extracted again to obtain correct attributes. Also, note that the parent attribute of node 13 has changed from node 12 to node 8 in a new SPT. This change still gives a correct (but different) SPT. The changes that were performed on the tree data structure were unnecessary. Node 13 could have maintained the same parent. Although this algorithm does not alter the SPT structure of the nodes whose distance is not affected by the change, it cannot guarantee that when there are several possible SPTs that the minimum number of changes will be made in the tree structure. (In fact, the unnecessary routing change in this example motivates us to consider more stable dynamic algorithms using the second incremental method to be discussed in the next section.)

Figure 2-4 shows how the First Incremental D’Esopo-Pape obtains a new SPT for the same problem. Some nodes are still extracted more than once, but the total
The number of extractions is smaller than in the Bellman-Ford algorithm (10 extractions instead of 12). In this example, the improvement is caused by the heuristics of D'Esopo-Pape. For example, when node 11 is enqueued for a second time in $Q$, it receives first priority in $Q$ so that it is extracted immediately after with correct attributes. Because node 11 is extracted earlier (extraction #6 rather than #9), it allows for nodes 12 and 14 to have correct attributes the very first time they are extracted.

Figure 2-5 shows how the First Incremental Dijkstra algorithm obtains a new SPT for the same problem. Because of the order in which nodes are extracted (the node with the smallest distance attribute first), every node is extracted only once. Therefore, this algorithm performs the minimum number of extractions. The cost incurred in such an improvement is that some data structure is needed in order to search for the node with the smallest distance attribute in $Q$.

On the other hand, Figure 2-6 shows how the basic algorithm is initialized when edge (1,7) decreases its cost by some 8 (from 9 to 1). The set of nodes inside the dotted curve (set $P$) will have their distance attributes decreased by 8, but the SPT structure inside $P$ will remain unchanged. Only the nodes outside of $P$ might have to
change parent attributes in order to maintain shortest distances from the source node. In this case, the shaded nodes can decrease their distance attributes by changing their parent attribute to some node in \( P \) (indicated by the thin arrow). Note that all SPT paths from \( P \) to nodes outside \( P \) must necessarily pass through the shaded nodes (the germinal set). All nodes in the germinal set are inserted in the list \( Q \). This initialization procedure is the same for all our dynamic algorithms.

Figure 2-7 shows how the First Incremental Dijkstra algorithm reconstructs an SPT after the initialization in Figure 2-6. Each number next to a node represents the order and the parent attribute of the extracted node. This algorithm only extracts each node once from \( Q \). The shaded nodes have all been visited by the algorithm and have had their distance attributes decreased. The nodes outside of \( P \) that are not shaded are the nodes that could not decrease their distance attributes, and therefore, they are not affected by the algorithm.
Figure 2-6: Initialization after a link cost improvement.

Figure 2-7: New SPT using First Incremental Dijkstra.
2.7 Second Incremental Method

Another set of dynamic algorithms can be obtained by changing the branching function $B(y, T)$ used in Step 2 of the basic algorithm so that it includes as many descendents as possible. For optimization purposes, the only nodes that should not be included in $B(y, T)$ are those that will be directly affected by some node extraction from $Q$ (there is no point in updating a node that we know will be later updated with a smaller distance once it is extracted from $Q$). Likewise, every node included in $B(n, T)$ should be removed from $Q$ (there is no point in keeping a node with a known non-optimal distance in $Q$). To specify exactly how the selection of the nodes is done, we present the following algorithm that defines the set $N = B(y, \hat{T})$ in Step 2 of the basic algorithm. The instruction $\text{DEQUEUE}(n, Q)$ removes node $n$ and its attributes from $Q$. The notation $B_1(k, \hat{T})$ denotes the set including node $k$ and its children in $\hat{T}$.

$$K \gets \{y\} \quad /* \text{the temporary queue starts with node } y */$$

$$N \gets \emptyset \quad /* \text{the branch set is initially empty */}$$

step (i)

if $K = \emptyset$  /* if the temporary queue is empty */

*Stop*  /* we are done. $N$ contains the desired output */

else

$\quad k \gets \text{EXTRACT}(K) \quad /* \text{pick any node from the temporary queue */}$

$\quad N \gets \{N, k\} \quad /* \text{add this node to the output set */}$

step (ii)

$\forall n \in B_1(k, \hat{T}) \quad /* \text{for all the immediate children of } k */$

$\quad \text{if } n \in Q \quad /* \text{if the child is in the candidate list } Q */$

$\quad \quad \text{if } D(n, \hat{T}) + \Delta \leq D(n, Q)$

$\quad \quad /* \text{and if the new flooded distance for the child is better */}$

$\quad \quad /* \text{than that in candidate list } Q */$
DEQUEUE\( (n, Q) \) /* remove it from the candidate list */
\[ K \leftarrow \{K, n\} \] /* add the child node \( n \) to the temporary queue */

else
\[ K \leftarrow \{K, n\} \] /* add the child node \( n \) to the temporary queue */

Go to step (i)

We call this method the second incremental method and its four algorithms are referred to as “Second Incremental Bellman-Ford,” “Second Incremental D’Esopo-Pape,” “Second Incremental Linear Dijkstra,” and “Second Incremental Heap Dijkstra.”

The first incremental method changes the attributes of only one node during each iteration. The second incremental method is less conservative since at each iteration an entire branch of \( \mathcal{T} \) changes its attributes. The simulation results shown in later sections indicate that the extra work performed by the second incremental method pays off in terms of average performance.

For example, Figure 2-8 shows how the Second Incremental Dijkstra algorithm obtains a new SPT after the initialization in Figure 2-2. First, node 7 is extracted from \( Q \) with node 2 as its parent attribute. All the descendents of node 7 have their distance attributes updated. Then node 8 is extracted with node 5 as its parent attribute; all its descendents have their distance attributes updated. Finally, node 9 is extracted with node 4 as its parent attribute. Each node is extracted at most once from \( Q \) (there are only three extractions in this case). Nevertheless, because of the distance updates, some nodes (8 and 9) are visited more than once by the algorithm. The advantage of this second incremental method over the first one (shown in Figure 2-5) is that even though some nodes are visited more than once, many fewer iterations take place, the list \( Q \) is much smaller and simpler, fewer search operations are needed, and most visited nodes only require a simple distance attribute update.

Another advantage of the second incremental method is that it guarantees that the minimum number of parent attributes will be changed. In other words, a link is removed from the current SPT tree \( (\mathcal{T}) \) only if that link cannot be part of any
possible SPT. This result is proved in Theorem 5 in Section 2.10. In this example, node 13 maintains node 12 as its parent attribute and does not change it to node 8 as done in the first incremental method.

Figure 2-9 shows the new SPT calculated by the Second Incremental Dijkstra algorithm after the initialization stage of the basic algorithm (see Figure 2-6). As seen from the figure, a minimum number of parent attribute changes are made.

2.8 Algorithmic Complexity

Here we will state theoretical bounds for the complexity of the dynamic algorithms presented in the preceding sections (four dynamic algorithms corresponding to each incremental method). The proofs of these theoretical bounds will be given in the Section 2.10.

The complexity of algorithms is measured in terms of the following:

- total number of comparisons made between the distances of two nodes,
- total number of comparisons made between two elements in the list \( Q \), and
Figure 2-9: New SPT using 2nd Inc Dijkstra (after a link cost improvement).

Note that in the worst case, the updated SPT can be completely different from the outdated SPT, and the complexity of dynamic SPT algorithms is no better (or sometimes even worse) than the static algorithms. However, as we discussed earlier, it is most often the case that a change in the weight of an edge will only yield small changes in the topology of the SPT. Thus, it is more useful to measure the complexity of dynamic SPT algorithms in terms of parameters that reflect the changes in the SPT.

Given that there is a change in the weight of some edge, let $\delta_d$ denote the minimum number of nodes that must change their distance or parent attributes (or both) and $\delta_{pd}$ the minimum number of nodes that must change their distance and parent attributes. Note that the parameters $\delta_{pd}$ and $\delta_d$ depend only on the topology of the network, the current SPT, and the link-state change, but not on the algorithm used. They reflect the minimum amount of structural change that any dynamic algorithm needs to perform in order to recompute the SPT. Also, let $D_{max}$ denote the maximum node degree, which corresponds to the maximum number of ports any router or switch in the network can have. We will evaluate the complexity of our algorithms in terms of these parameters. We shall focus on the case where there is a change in the weight of only a single link.
For comparison with the well-known static algorithms, we state the following known results regarding their complexity [33]:

- Bellman-Ford: $O(|E| \cdot |V|)$
- D'Esopo-Pape: *No Polynomial Upper Bound*
- Linear Dijkstra: $O(|V|^2)$
- Heap Dijkstra: $O(|E| \cdot \lg |V|)$
- Fibonacci Dijkstra: $O(|V| \cdot \lg |V| + |E|)$ \hspace{1cm} (2.2)

The Fibonacci Dijkstra algorithm uses the data structure of a Fibonacci heap to implement the priority queue for $Q$. Although the algorithm has the lowest asymptotic complexity among others shown above, it is mostly of theoretical interest because of the large constant overhead involved in operating Fibonacci heaps [33]. Though the worst-case asymptotic complexity of its dynamic versions will be given later for the sake of completeness, we shall not simulate them since they are not practical data structures.

### 2.8.1 First Incremental Method

The following summarizes the complexity of the dynamic SPT algorithms derived using the first incremental method:

- Bellman-Ford: $O(D_{\text{max}} \cdot \delta_d^2)$
- D'Esopo-Pape: *No Polynomial Upper Bound*
- Linear Dijkstra: $O(\delta_d^2 + D_{\text{max}} \cdot \delta_d)$
- Heap Dijkstra: $O(D_{\text{max}} \cdot \delta_d \cdot \lg \delta_d)$
- Fibonacci Dijkstra: $O(\delta_d \cdot \lg \delta_d + D_{\text{max}} \cdot \delta_d)$ \hspace{1cm} (2.3)

The complexity of the initialization is just the number of edges visited during step
1 of the basic algorithm because this stage does not involve queuing or searching operations. In Case 2 of the initialization procedure (when an edge decreases its weight), \(|B_{max}(E(e))| \leq \delta_d\). Therefore, the initialization complexity is always bounded by \(O(D_{max} \cdot \delta_d)\), which is always less than the complexity of the rest of the algorithm.

However, in Case 1 of the initialization procedure (when an edge increases its weight), \(|B_{max}(E(e))| \geq \delta_d\). Therefore, in rare cases, the initialization complexity might be higher than that of steps 2-4. If we want to obtain a strict linear complexity bound, we need to slightly modify the initialization procedure so that only the nodes that must change distance attributes are included in the set \(\mathcal{N}\). The initialization complexity after such a modification is then bounded by \(O(D_{max} \cdot \delta_d)\), which is always less than the complexity of the rest of the algorithm.

The following pseudocode illustrates how the set \(\mathcal{N}\) can be obtained in \(O(D_{max} \cdot \delta_d)\) time if our algorithm keeps track of all the possible shortest paths. Keeping track of all shortest paths can be easily achieved with an additional small constant overhead \((O(D_{max})\) times more space) in the algorithm. This is done by adding an alternative parent attribute to every node in \(Q\) or \(\hat{T}\) whenever an alternative shortest path is found (when \(D(E(e), \hat{T}) = newdist\) in Step 4 of the basic algorithm or when \(D(n, \hat{T}) + \Delta = D(n, Q)\) in step (ii).

\[
\begin{align*}
\mathcal{K} & \leftarrow \{y\} \\
\mathcal{N} & \leftarrow \emptyset \\
\mathcal{P} & \leftarrow \emptyset \\
\text{step (A)} \\
\text{if } \mathcal{K} = \emptyset \\
\quad \text{Goto step (C)} \\
\text{else} \\
\quad k & \leftarrow \text{EXTRACT} (\mathcal{K}) \\
\quad \mathcal{N} & \leftarrow \{\mathcal{N}, k\}
\end{align*}
\]
step (B)
\[ \forall n \in B_1(k, \hat{T}) \]
if \( n \) has an unmarked alternative parent
mark \( k \) in node \( n \)
\[ P \leftarrow \{P, n\} \]
else
\[ K \leftarrow \{K, n\} \]
if \( k \) is the alternative parent of some node \( n \)
mark \( k \) in node \( n \)
if all parent and alternative parents are marked in \( n \)
\[ K \leftarrow \{K, n\} \]
\[ P \leftarrow P - \{n\} \]
unmark everything in \( n \)

Go to step (A)

step (C)
\[ \forall n \in P \]
\[ P(n, \hat{T}) = \text{any unmarked alternative parent of } n \]
remove the marked alternative parents of \( n \)

Because in practice \( |B_{max}(E(e))| \) hardly exceeds \( \delta_d \) by much, our simulations do not include this modification. The modification is given mainly to be able to obtain a strict theoretical bound on the complexity of the initialization procedure.

### 2.8.2 Second Incremental Method

The following summarizes the complexity of some dynamic algorithms that can be derived using the second incremental method:
Bellman-Ford \( O(D_{\text{max}} \cdot \delta_d^3) \)

D'Esopo-Pape \textit{No Polynomial Upper Bound}

Linear Dijkstra \( O(\delta_p d \delta_d + \gamma \cdot D_{\text{max}} \cdot \delta_d) \)

Heap Dijkstra \( O(\gamma \cdot D_{\text{max}} \cdot \delta_d \cdot \log \delta_d) \)

Fibonacci Dijkstra \( O(\delta_d \cdot \log \delta_d + \gamma \cdot D_{\text{max}} \cdot \delta_d) \), \hspace{1cm} (2.4)

where \( \gamma \) denotes the \textit{redundancy} factor, which represents the average time that each node is visited by the algorithm. This factor, although very close to 1 in practice, can take values between 1 and \( \delta_{pd} \).

Since in the theoretical worst-case, the term \( \gamma \) can be as large as \( \delta_{pd} \), the worst-case asymptotic complexity of the second incremental algorithm is worse than that of the first incremental algorithm. However, since in practice, there are very good reasons to believe that \( \gamma \) is very close to one, in practice the algorithms in the second incremental method generally outperform those in the first incremental method. In the simulation section, we will discuss why \( \gamma \) is generally small and why the second incremental method performs well.

### 2.9 Simulation Results

In this section we investigate via simulations the complexity of the different shortest path algorithms presented earlier. Each algorithm is simulated in a wide variety of randomly generated networks.

#### 2.9.1 Random Network Generator

The networks generated in the simulations are obtained by using a modified version of the random topology generator (RTG) software [35].

The software first randomly places the desired number of nodes \( N \) in a 300 x 300
grid. It then checks the Euclidean distance $d$ between every two nodes and places an edge between two nodes with the probability $P(d)$ described below:

$$P(d) = \begin{cases} k & \text{if } d \leq l \\ k \cdot e^{-l/d} & \text{if } d > l \end{cases}$$  

(2.5)

The Euclidean distance of an edge is then used as the link weight. Any two nodes within distance $l$ of each other have a constant probability of being joined by an edge. Likewise, nodes with a separation greater than $l$ have a probability of being connected which decays exponentially with their separation. For convenience, we express $l$ in terms of the normalized distance $L$:

$$l = \frac{300}{\sqrt{N}} \cdot L$$  

(2.6)

The parameter $k$ is adjusted by the software such that the resulting network configuration will have an average node degree ($\# \text{ edges} \div \# \text{ nodes}$) equal to a constant $D$. As an example, when we set $N = 620$, $L = 0.5$, and $D = 7$, we obtain the network in Figure 2-10. For most of the simulations we used parameters $L = 0.5$ and $D = 7$. We find that changing parameter $L$ does not cause noticeable changes in the results. The complexity with respect to changes in parameter $D$ will be analyzed in Section 2.9.4.

### 2.9.2 Event Generation

Once a random network is obtained, we simulate link failures and recoveries on that network. In every network, $(25 \cdot D)$ edges are randomly selected (with uniform probability). First, a link failure is simulated (the edge weight becomes infinity) on each of the edges, and later, each of the edges recovers its previous weight.

After each event, the shortest path tree is recomputed using the different algorithms described in the preceding sections. During the computation, we keep track
of the number of operations that the algorithm performs. To be consistent with the complexity measure we stated earlier for the algorithms, we increment the observed complexity of the algorithm every time any one of the following occurrences takes place:

- A comparison is made between the distances of two nodes sharing a common edge.
- A comparison is made between two elements in the queue.

2.9.3 Node Based Complexity

During the first set of simulations, we compare the average observed complexity of the different dynamic SPT algorithms. For each network size, 40 different networks of that size are randomly generated ($L = 0.5, D = 7$). For each network, 25x7 or 175 link failures and 175 link recoveries were simulated. After each simulation the
observed complexity of that event is recorded. The average observed complexity over the $40 \times 350$ events of the same network size is then computed.

Figure 2-11 illustrates how the average observed complexity of each static SPT algorithm increases as the network size increases. Because of the sparseness of the graph, it turns out that the Bellman-Ford and D’Esopo-Pape algorithms, which have the highest asymptotic complexity, actually perform better on average. As expected, Linear Dijkstra has a much worse performance than any other algorithm, and for large networks, it becomes computationally prohibitive to use this algorithm. It is possible that for networks larger than those simulated, Heap Dijkstra will perform better than Bellman-Ford. Nevertheless, our results indicate that D’Esopo-Pape clearly performs better than the other three algorithms in the networks simulated.

When we perform the same experiment using the first incremental method, we notice a tremendous reduction in the average algorithmic complexity. Figure 2-12 compares the average observed complexity of the different algorithms using the first incremental method. As expected, all these dynamic algorithms have observed complexity orders of magnitudes lower than the static algorithms. Furthermore, unlike the static algorithms, the first incremental method yields dynamic algorithms with
complexity sublinear with respect to the network size. That is, the average observed complexity grows slower than the network size.

Figure 2-12 shows that among the dynamic algorithm, Linear Dijkstra has by far the highest complexity. The simpler Bellman-Ford and D'Esopo-Pape algorithms achieve much better results. However, with the first incremental method, Heap Dijkstra yields the lowest complexity. This happens because the Heap Dijkstra algorithm maintains a low-complexity priority queue that can reduce the number of iterations required in the basic algorithm.

Figure 2-13 shows the average observed complexity of the second incremental method. In general, there is some improvement relative to the first incremental method. With this second method, the Heap and Linear Dijkstra outperform the other two algorithms. Because in the second incremental method an entire branch of nodes is selected, at each iteration the selection of nodes for update is more optimized than in the first incremental method, thus yielding a lower complexity. Because Dijkstra algorithm chooses the node with smallest distance attribute at each iteration, it outperforms the other two algorithms. Perhaps surprisingly, because the number of iterations and the priority queue remain small using the second incremental method,
Linear Dijkstra with its low overhead outperforms Heap Dijkstra. Also notice that Bellman-Ford also yields a lower complexity than D'Esopo-Pape. The heuristics used in D'Esopo-Pape do not seem to have any positive effect using the second incremental method.

Figure 2-14 compares the Linear and Heap Dijkstra algorithms in both the first and second incremental methods. With Heap Dijkstra, the second incremental method performs slightly better than the first incremental one. With Linear Dijkstra, the second incremental method significantly outperforms the first method. The improved performance of the second method is due to fewer iterations performed and a smaller priority queue. The smaller priority queue allows for simple efficient search mechanisms (as in Linear Dijkstra) and renders complicated search structures unnecessary.

2.9.4 Degree Based Complexity

Figures 2-15 to 2-18 illustrate the average observed complexity with respect to the average node degree. The size of each simulated network is 500 nodes.

Figure 2-15 shows how the observed complexity of the static method varies with
the average node degree. We observe that D'Esopo-Pape always performs better than Bellman-Ford. Linear Dijkstra performs the worst, especially when the node degree becomes very large.

Figure 2-16 shows that using the first incremental method, Bellman-Ford and D'Esopo-Pape perform better than Linear Dijkstra and Heap Dijkstra when the node degree is small. However, when the node degree increases, and the network becomes more complex, both Dijkstra algorithms yield better performance. Notice that the observed complexity for all the algorithms decreases as the node degree increases. This is because having a larger number of edges in the graph implies that changing the weight of any one edge is less likely to have a significant impact on the structure of the SPT.

Figure 2-17 illustrates that when using the second incremental method, Dijkstra algorithms always perform better than Bellman-Ford and D'Esopo-Pape. As the node degree increases, this difference becomes greater. As before, all the algorithms exhibit lower complexity as the node degree increases.

Figure 2-18 compares the Dijkstra algorithms using the first and second incremental methods. The second incremental method always performs better (specially with
Figure 2-15: Complexity of static algorithms.

Figure 2-16: Complexity of First Incremental algorithms.
the Linear Dijkstra algorithm). However, as the node complexity increases, the Heap Dijkstra algorithm with the first and second methods start exhibiting comparable performances.

### 2.9.5 Further Discussion on Dijkstra Algorithms

The simulation results presented so far indicate that the dynamic Dijkstra algorithms perform better than dynamic Bellman-Ford and D’Esopo-Pape. Here we want to compare in more detail the complexity of Linear and Heap Dijkstra algorithms under the first and second incremental algorithms. Figure 2-19 compares the average complexity of the two methods. Each data point represents the average observed complexity measured after $40 \times 350$ events (as in the preceding figures). There is a different data point for each different network size simulated. The position in the horizontal axis represents the averaged complexity observed when the first incremental method is used, while the vertical axis represents the averaged observed complexity for the same set of events when using the second incremental method. The solid line has slope 1. We observe that in all cases, the second method yields a better performance.
than the first method. This is especially true for the Linear Dijkstra algorithm which becomes very inefficient under the first incremental method when the network size is large.

Figure 2-20 compares the worst observed complexity of Dijkstra algorithms using the two incremental methods. Each data point represents the maximum observed complexity of $40 \times 350$ events. Although there is a greater variance in the performance than in the average case, most of the time, the second incremental method still performs better than the first one. With Linear Dijkstra, this is always the case.

We also would like to point out some intuitive reasons for the better performance of the Dijkstra algorithms using the second incremental method. As proven in the Section 2.10, the second method has far fewer nodes extracted from $Q$ ($\delta_{pd}$ rather than $\delta_d$), although the number of visited nodes might be greater ($\gamma \cdot \delta_d$ rather than $\delta_d$) by some redundancy factor $\gamma$. Our simulation results indicate that the reduction in the number of queuing operations outweighs the increase in the numbers of nodes visited.

Figure 2-21 compares, for each incremental Dijkstra algorithm, the number of nodes that changed their distance attributes with the number of nodes that changed
Figure 2-19: Average complexity of Incremental Dijkstra algorithms.

Figure 2-20: Worst-case complexity of Incremental Dijkstra algorithms.
Figure 2-21: Changes in distance attributes vs. parent attributes using Dijkstra Algorithms.

their parent attributes. Recall that the second incremental Dijkstra algorithm changes the minimum number of parent attributes, while both algorithms change the minimum number of distance attributes. Each data point represents the worst case of $40 \times 350$ events. The steepest line is a unity slope line, while the other line is the best fit line to the data points obtained using the second incremental method. Although there is a large variance of results, in most cases we observe that $\delta_{pd} << \delta_d$, and on average, the difference is by more than one order of magnitude.

Figure 2-22 illustrates the number of nodes visited during the execution of the two incremental Dijkstra algorithms. Each data point represents the average (over $40 \times 350$ events) number of nodes that each algorithm visited for a given network size. Recall that in the first incremental Dijkstra algorithm, each node will be visited only once, whereas in the second incremental algorithm, a node can be visited more than once. Thus, the extra number of nodes visited in the second algorithm compared with the first algorithm indicates the redundancy factor $\gamma$ (see Section 2.8.2), which in practically never above 1.2 as seen from the figure.

Finally, Figure 2-23 compares the worst-case observed complexity (worst of $40 \times$
350 events) of Heap Dijkstra using the second incremental method with the average complexity of static Heap Dijkstra algorithm. The figure shows that even under the worst case observed, the dynamic algorithm rarely performs worse than the static algorithm, which recomputes the entire SPT from scratch.

### 2.10 Proofs of Theoretical Results

#### 2.10.1 Correctness of the Basic Algorithm

We will first show that the basic algorithm correctly computes a new SPT for the case where only one edge in the network changes its weight. We will then indicate how the proof can be extended to the case where multiple edges can change their weights.

Recall from Section 2.3 that the SPT for a given $\mathcal{G}$ is not unique. Let $\mathcal{S}$ be the set of SPTs in $\mathcal{G}$. For every $\mathcal{T} \in \mathcal{S}$, the distance attribute $D(n, \mathcal{T})$ contains the shortest distance between node $n$ and $Source(\mathcal{G})$. Note that even though the SPT is not unique, the distance attributes in $\mathcal{T} \in \mathcal{S}$ are identical, and hence $D(n, \mathcal{T})$ is unique.

**Lemma 1** In each stage of the basic algorithm, if $d = D(n, \hat{T}) \neq \infty$, $d$ is the
Figure 2-23: Comparative complexity of incremental worst cases with static method.

length of the path in $\tilde{T}$ between node $n$ and $\text{Source}(G)$.

**Proof:** In the static version of initialization, the only finite distance attribute belongs to the source ($= 0$). In the dynamic version of initialization, every node $n$ in $\tilde{T}$ has a distance attribute equal to the length of the path in $\tilde{T}$ between $n$ and the source.

During steps 2-4, a new distance attribute for some node $n$ is updated in $\tilde{T}$ by adding the distance attribute of $n$'s new parent to the weight of the edge between the parent and $n$. Therefore, the new distance attribute represents the length of the some path in $\tilde{T}$ from the source to $n$ going through the new parent node. \hfill $\square$

**Lemma 2** In the basic algorithm, $\tilde{T}$ always maintains a tree structure.

**Proof:** It is clear that at the end of initialization, $\tilde{T}$ maintains a tree structure. During steps 2-4 of the basic algorithm, every node in $\tilde{T}$ has a distance that is smaller than the distance of any of its descendents. The only time when the structure of $\tilde{T}$ can change is during step 2 when a node $n$ selected from the list $Q$ changes parent in $\tilde{T}$. However, for the node $n$ to be selected from $Q$, the new parent must have a smaller distance than $n$. Therefore, the parent of $n$ cannot be a descendent of $n$ and hence there cannot be any cyclic path in $\tilde{T}$. Moreover, each node in $\tilde{T}$ has only one parent. Thus, $\tilde{T}$ maintains a tree structure. \hfill $\square$
Lemma 3  During steps 2-4, $D(n, \hat{T}) \geq D(n, T^*) \forall n \in \mathcal{N}, \forall T^* \in \mathcal{S}$.

Proof: According to Lemma 1, $d = D(n, \hat{T})$ always represents the distance from the source to node $n$ going through some path in $\mathcal{G}$. Since, by definition $T^*$ defines a tree of shortest paths, the distance through this path cannot be more than the distance through the path defined by $\hat{T}$.

Lemma 4  The basic algorithm will terminate.

Proof: A node $n$ and its new attributes can only enter $\mathcal{Q}$ if the new distance attribute is smaller than the distance attribute for $n$ in $\hat{T}$. Because there are only a finite number of paths from the source to each node, using Lemma 2 and Lemma 1 we know that the distance attribute for some particular node can only improve a finite number of times. In other words, a node can only enter $\mathcal{Q}$ a finite number of times. Since at least one node is selected and removed from $\mathcal{Q}$ in each iteration, the algorithm must terminate.

For convenience, we introduce the following definition:

Definition 1  A node $n$ is said to be consolidated at some step of the algorithm if its attributes in $\hat{T}$ will not change further in the execution of the algorithm.

Lemma 5  If node $n$ is enqueued in $\mathcal{Q}$ with its distance attribute $d_{\text{min}}$ equal to its shortest distance from the source, after a finite number of steps, node $n$ will be consolidated with its distance attribute $d_{\text{min}}$.

Proof: By Lemma 4, we know that $n$ will be eventually extracted from $\mathcal{Q}$ before the algorithm terminates. Since a node can enter $\mathcal{Q}$ only when its distance can be reduced, node $n$ with its shortest distance $d_{\text{min}}$ cannot re-enter $\mathcal{Q}$. In other words, $n$ will be consolidated after a finite number of steps.

Lemma 6  If node $n$ is consolidated with its shortest distance, all descendents of $n$ with respect to any SPT will be consolidated with their shortest distances.
Proof: It suffices to prove the statement for all children (next generation) nodes of \( n \) in any SPT since we can then apply the result recursively for all descendents. Let \( n' \) be any of \( n \)'s children nodes connected to \( n \) via edge \( e \) in any SPT. Since \( n \) is consolidated with its shortest distance \( d_{\min} \), the value \( d_{\min} + W(e) \) must be the shortest distance of \( n' \).

After node \( n \) is consolidated with its shortest distance \( d_{\min} \) during step 3, its child \( n' \) will either have an updated distance \( d_{\min} + W(e) \) or be examined in step 4 of the algorithm. In the former case, \( n' \) cannot improve its distance further and will be consolidated. In the latter case, \( n' \) will enter the list \( Q \) with distance \( d_{\min} + W(e) \), and according to Lemma 5, \( n' \) will also become consolidated. \( \square \)

We now define a germinal set in order to proceed with the rest of the proof.

Definition 2 Let \( D \) be the set of nodes whose distance to the root through some path defined by some current rooted tree \( \tilde{T} \) is larger than the minimum distance. We define a germinal set \( A \) with respect to a current tree \( \tilde{T} \) to be any set of nodes whose descendents with respect to any valid SPT include all nodes in \( D \).

Lemma 7 Suppose at some instant in the execution of the basic algorithm, the tree data structure \( \tilde{T} \) is not an SPT. If all nodes in any germinal set \( A \) (with respect to current tree \( \tilde{T} \)) are enqueued in the list \( Q \) with their shortest distances, iterating through steps 2-4 of the basic algorithm will result in \( \tilde{T} \) representing an SPT.

Proof: According to Lemma 6, the nodes in \( D \) (as explained in Definition 2) will be consolidated with their shortest distances. Since all distance attributes are distances of real paths (Lemma 1), no distance attribute can be less than the optimal distance. Therefore, the distance attributes of nodes not in \( D \) are their shortest distances, and they are already consolidated. Hence, when all nodes are consolidated, \( \tilde{T} \) represents an SPT. \( \square \)

Any germinal set of nodes serves as "seeds" to (re)construct an entire SPT. The correctness of the basic algorithm then follows by proving that the initialization of
the algorithm enqueues a germinal set of nodes $A$ with their shortest distances in the list $Q$.

**Lemma 8** The initialization of the basic algorithm enqueues a germinal set of nodes with their shortest distances in the list $Q$.

**Proof:** When the algorithm starts from scratch (static version), the initialization enqueues only $Source(g)$ (with the shortest distance being 0). Since any node is a descendent of the source node $Source(g)$ with respect to any SPT tree, any set containing $Source(g)$ is a germinal set.

Now consider the dynamic version of the initialization. Suppose an edge $e$ in $\hat{T}$ (the original SPT) increases its weight by $\Delta$ and suppose that the distance of each node $B_{max}(E(e), \hat{T})$ has been updated with an increase of $\Delta$. After the update, let $D_{\hat{T}}$ be the set defined in Definition 2. Note that $D_{\hat{T}}$ is a subset of $B_{max}(E(e), \hat{T})$ containing those nodes each of which can find a shorter path from the source without passing through $e$. These alternative paths must pass through nodes in $R = B_{max}(E(e), \hat{T})$ that can decrease their distance label in $\hat{T}$ by choosing a parent $p \notin R$. The set $A$ of all such nodes is a germinal set and will be enqueued with their shortest distances after the initialization procedure.

On the other hand, suppose the weight of an edge $e$ in $\hat{T}$ decreases. The set $D_{\hat{T}}$ then consists of nodes each of which can find a shorter path from the source through the edge $e$. The nodes in $R = B_{max}(E(e), \hat{T})$ are first removed from $D_{\hat{T}}$ by updating their new distance attributes (the parent attribute remains the same). The remaining nodes in $D$ are not in $R$, but their shortest paths must pass through nodes in $R$. The set of nodes not in $R$ that can decrease their distance label by selecting a parent in $R$ is therefore a germinal set and will be enqueued with their shortest distances after the initialization procedure. $\square$

**Theorem 1** Upon termination of the basic algorithm, $\hat{T}$ represents an SPT with all nodes in $G$ which are reachable from $Source(G)$.

**Proof:** From Lemma 4 we know that the basic algorithm will terminate. From
Lemma 8 we know that after initialization, $Q$ will contain a germinal set. According to Lemma 7 the basic algorithm will compute an SPT.

To show that the basic algorithm also works when multiple edge weights are changed at once (see end of Section 2.4), we need to show that the set of enqueued nodes after the extended initialization procedure also constitutes a germinal set.

For the case where there are multiple edge weight increments, during initialization, every node in $D_f$ (same meaning as in Definition 2) is first updated with a larger distance attribute. Then, all the nodes in $D_f$ that can get a smaller distance attribute through a parent outside of the updated area $R$ will be enqueued in $Q$. Since all the new alternative paths must pass through these enqueued nodes, they form a germinal set.

For the case where there are multiple edge weight decrements, it is sufficient to observe that any node that is affected by the changes (ultimately decrements its distance from the source) must be a descendent (in one of the new SPT tree) of the end node of one of the edges whose weights have been reduced. Since the set of these end nodes is a germinal set, the set which includes all their direct descendents that can improve their distance (and are enqueued in $Q$) must also be a germinal set.

2.10.2 Algorithmic Complexity

First Incremental Method

Theorem 2 The First Incremental Bellman-Ford algorithm has a complexity of $O(D_{max} \cdot \delta_d^2)$.

Proof: Let $R_d$ denote the set of $\delta_d$ nodes whose distance attributes must change. The nodes in $R_d$ are extracted in cycles. After extraction cycle $i$, each visited node in $R_d$ has a distance attribute that is not more than the length of the shortest of all paths from the source that have at most $i$ hops inside in $R_d$. Since the shortest path has at most $(\delta_d - 1)$ hops in $R_d$, the distance attribute of each nodes in $R_d$ cannot change after extraction cycle $(\delta_d - 1)$. Each extraction cycle involves at most $\delta_d$ nodes (only nodes in $R_d$ will enter $Q$). After a node extraction, at most $D_{max}$ edges
will be visited. Therefore, the complexity of this algorithm is $O(D_{\text{max}} \cdot \delta_d^2)$. □

To derive the complexity for the dynamic Dijkstra algorithms, we first prove the following lemmas.

**Lemma 9** Given any shortest path tree $T$, let $T_{\text{min}}^N$ be any tree containing nodes with the $N$ smallest distance attributes. Also let $N_{\text{new}}^N$ denote the set of nodes not in $T_{\text{min}}^N$ but adjacent to nodes in $T_{\text{min}}^N$. Using the Dijkstra algorithm, every node $n \in N_{\text{new}}^N$ chooses a parent node $p \in T_{\text{min}}^N$ such that its distance attribute $D(p, T_{\text{min}}^N) + W(n, p)$ is minimized. The node $n$ with the smallest distance attribute in $N_{\text{new}}^N$ has the $(N + 1)^{\text{st}}$ smallest distance attribute in any SPT.

**Proof:** The distance attribute of every node in any SPT $T$ is larger than that of any of its ancestor nodes. If $n_{N+1}$ is the node with the $(N + 1)$ smallest distance attribute for any $T \in S$, the parent of $n_{N+1}$ must be in $T_{\text{min}}^N$. Therefore, $n_{N+1}$ must be in $N_{\text{new}}^N$ and its distance attribute in $N_{\text{new}}^N$ must be the same as that in $T \in S$. The other nodes in $N_{\text{new}}^N$ will have a distance attribute that is at least their distance attribute in $T$. Therefore, $n_{N+1}$ is the node in $N_{\text{new}}^N$ with the smallest distance attribute. □

**Lemma 10** In the Dijkstra algorithms, if a node $n$ is extracted from $Q$, it will not re-enter $Q$ before the termination of the algorithm.

**Proof:** The proof is by induction. It can be shown that after the initialization, the first node to be extracted from $Q$ has its distance attribute equal to its shortest distance, and thus it will not re-enter $Q$. We now assume that the set of nodes $N_{\text{ext}}$ extracted previously have shortest distance attributes. At any time, $Q$ includes all the nodes adjacent to nodes in $N_{\text{ext}}$. Each node $n \in Q$ has the parent in $N_{\text{ext}}$ such that its distance attribute is minimized. According to Lemma 9, the next node extracted will have a shortest distance attribute. By induction, every node extracted must have a shortest distance, and therefore cannot re-enter $Q$.

**Theorem 3** The First Incremental Linear Dijkstra algorithm has a complexity of $O(\delta_d^2 + D_{\text{max}} \cdot \delta_d)$, the First Incremental Heap Dijkstra has a complexity of $O(D_{\text{max}} \cdot \delta_d^2)$.
\( \delta_d \cdot \lg \delta_d \), and the First Incremental Fibonacci Dijkstra has a complexity of \( O(\delta_d \cdot \lg \delta_d + D_{max} \cdot \delta_d) \).

**Proof:** After the initialization, \( Q \) contains all the nodes that can improve their distance attribute by choosing a parent with a shortest distance attribute. From Lemma 10, we know that when a node is extracted from \( Q \) it will never re-enter \( Q \). Therefore there will be a maximum of \( \delta_d \) extractions. After each extraction all the edges leaving the extracted node are visited, which in turn will update the distance attribute of some node in \( Q \).

Since there are at most \( \delta_d \) nodes in \( Q \), extracting a node from \( Q \) takes \( O(\delta_d) \) complexity with linear search and \( O(\lg \delta_d) \) complexity with binary and Fibonacci heaps. Updating a distance attribute in \( Q \) takes \( O(1) \) complexity with a regular list and a Fibonacci heap, while requiring \( O(\lg \delta_d) \) complexity with a binary heap. Therefore, the complexity of First Incremental Linear Dijkstra is \( O(\delta_d + D_{max} \cdot \delta_d) \), that of First Incremental Heap Dijkstra is \( O(D_{max} \cdot \delta_d \cdot \lg \delta_d) \), and that of First Incremental Fibonacci Dijkstra is \( O(\delta_d \cdot \lg \delta_d + D_{max} \cdot \delta_d) \).

---

**Second Incremental Method**

**Theorem 4** The Second Incremental Bellman-Ford algorithm has a complexity of \( O(D_{max} \cdot \delta_d^3) \).

**Proof:** As in the proof of Theorem 2, we will denote as \( R_d \) the set of \( \delta_d \) nodes whose distance attributes must change. After extraction cycle \( i \), each visited node in \( R_d \) has a distance attribute that is not more than the length of the shortest of all paths from the source that have at most \( i \) hops inside \( R_d \). As in the first incremental case at most \( \delta_d^2 \) nodes will be extracted from \( Q \). However in the second incremental case, after extracting one node, the algorithm might visit at most all the edges \( (D_{max} \cdot \delta_d) \) in \( R_d \). Therefore the complexity of this algorithm is \( O(D_{max} \cdot \delta_d^3) \).
Lemma 11  In the Second Incremental (Linear or Heap) Dijkstra algorithm, only the nodes (δpd of them) whose parent attribute must necessarily change are extracted exactly once from Q.

Proof: Every node inserted in Q (step 4) must be outside the branch B(y, Ṭ), while its parent in Q is inside the branch. Therefore, any node n inserted in Q must have a parent attribute in Q that is different from that in Ṭ.

Let P be the set of nodes that have been consolidated with optimal distance attributes in Ṭ. At any time after initialization, Q contains all nodes adjacent to nodes in P that can improve their distance attribute by changing the parent attribute in Ṭ to some node in P. According to Lemma 9, the nodes extracted from Q will be consolidated with incrementing distance attributes.

Furthermore, when node n is extracted from Q with a shortest distance attribute d, every other node with shortest distance attribute less than d has already been consolidated. Since n was not dequeued from Q, the segment between n and P(n, Ṭ) cannot be part of an SPT. Therefore, node n must change its parent attribute to the new one contained in Q.

Theorem 5  The Second Incremental (Linear or Heap) Dijkstra algorithm obtains a new valid SPT in Ṭ, while modifying the minimum number of parent attributes in Ṭ.

Proof: During initialization, the only nodes that change parent attributes are the descendents of the modified edge and the descendents of the incrementing edge (in Ṭ) that can maintain the same distance attribute by adopting an alternative parent.

During the rest of the algorithm, a parent attribute can be changed only when a node is extracted from Q. According to Lemma 11, only the nodes that have a parent in Ṭ different from any parent in T ∈ S will change the parent attributes.

Theorem 6  The complexity of the Second Incremental Linear Dijkstra algorithm is O(δpδd+γ·Dmax·δd), that of Second Incremental Heap Dijkstra is O(γ·Dmax·δd·lg δd), and that of Second Incremental Fibonacci Dijkstra is O(δd·lg δd+γ·Dmax·δd).
Proof: According to Lemma 11, only $\delta_{pd}$ nodes will be extracted from $Q$. After a node extraction at most $\delta_d$ nodes will be visited. Therefore the total number of edges visited is $\gamma \cdot D_{max} \cdot \delta_d$, where $\gamma$ is at worst $\delta_{pd}$.

For Second Incremental Linear Dijkstra, an extraction requires $O(\delta_d)$ complexity while an update or removal requires $O(1)$ complexity. Therefore the total complexity is $O(\delta_{pd}\delta_d + \gamma \cdot D_{max} \cdot \delta_d)$. For Second Incremental Heap Dijkstra, every operation on $Q$ has cost $O(\lg \delta_d)$. In the worst case, some operations might be done on $Q$ every time a node is visited. Therefore the complexity is $O(\gamma \cdot D_{max} \cdot \delta_d \cdot \lg \delta_d)$. For the Second Incremental Fibonacci Dijkstra, an extraction or removal requires $O(\lg \delta_d)$ while an update only requires $O(1)$ complexity. Therefore, its total complexity is $O(\delta_d \cdot \lg \delta_d + \gamma \cdot D_{max} \cdot \delta_d)$.

2.11 Chapter Summary

In this chapter, we have presented new dynamic algorithms for computing a shortest path tree (SPT) in a directed graph. Our dynamic algorithms are derived using a unified algorithmic framework, under which the best known SPT algorithms (static or dynamic) can also be derived as special instances. Within this algorithmic framework, we have proposed two specific approaches that transform known static SPT algorithms to new dynamic ones. These new dynamic algorithms have theoretical asymptotic complexity bounds that match the best known results. We have also performed extensive simulations and discussed the implications of the simulation results on the average-case as well as worst-case complexity of our algorithms. Among all the dynamic algorithms we studied, the algorithms using the first incremental method give rise to the best theoretical worst-case time bounds, while those using the second incremental method give rise to the best practical or average running times. The most promising algorithm for practical purposes is the Second Incremental Heap Dijkstra algorithm. Its novelty is that it provably minimizes the number of link changes in the SPT, while exhibiting in our simulations a faster average performance than all the other incremental algorithms.
Chapter 3

New Dynamic SPT Algorithm based on a Ball-and-String Model

A key functionality in today's widely used interior gateway routing protocols such as OSPF and IS-IS involves the computation of a shortest path tree (SPT). In many existing commercial routers, the computation of an SPT is done from scratch following changes in the link states of the network. As there may coexist multiple SPTs in a network with a set of given link states, such recomputation of an entire SPT not only is inefficient, but also causes frequent unnecessary changes in the topology of an existing SPT and creates routing instability.

We saw in Chapter 3 how an entire set of dynamic SPT algorithms can be derived to solve these problems. This chapter uses the framework presented in Chapter 2 to develop a single new dynamic SPT algorithm that makes use of the structure of the previously computed SPT. Our algorithm can also be derived by recasting the SPT problem into an optimization problem in a dual linear programming framework, which can be interpreted using a ball-and-string model. In this model, the increase (or decrease) of an edge weight in the tree corresponds to the lengthening (or shortening) of a string. By stretching the strings until each node is attached to a tight string, the resulting topology of the model defines an (or multiple) SPT(s). By emulating the dynamics of the ball-and-string model, we can derive an efficient algorithm that propagates changes in distances to all affected nodes in a natural order and in a most
economical way. Compared with existing results, our algorithm has the best-known performance in terms of computational complexity as well as minimum changes made to the topology of an SPT. Proofs of correctness for our algorithm and simulation results illustrating its complexity are also presented.

3.1 Background and Motivation

Recall that when the topology in a routing area changes (e.g., a link fails, recovers, or changes its routing cost), every router in the region is notified of the change. After updating the corresponding topology changes in its link state database, each router recomputes its SPT. In most of today's commercial routers, this recomputation is done by deleting the current SPT and recomputing it from scratch by using the well-known Dijkstra algorithm [7].

Chapter 2 tackles this problem by introducing a new framework to convert various conventional static SPT algorithms into dynamic ones. By adjusting various parameters, the framework can produce dynamic versions of not only the Dijkstra algorithm, but also the Bellman-Ford [2] and D'Esopo-Pape [3] algorithms. This framework characterizes dynamic SPT algorithms in a unified way and establishes a general proof of correctness for all the algorithms that fit the framework. The difference between each of the specific dynamic algorithms in this framework is the way in which nodes are stored and extracted from a general set. The work in Chapter 2 only considers extraction criteria that are derived from the search criteria of the corresponding static SPT algorithms.

In this chapter, we will present a single novel algorithm that can dynamically update the SPT following changes in the link states. Our dynamic algorithm also uses information of the outdated SPT and updates only the part of the SPT that is affected by the change. Instead of presenting a general set of dynamic algorithms, this chapter will present a single dynamic SPT algorithm. The new algorithm presented in this chapter fits in the framework of Chapter 2. However, unlike all the algorithms discussed in the Chapter 2, our new algorithm described in this chapter
uses a new search criterion, not derived from any existing static SPT algorithm, and is provably better than the dynamic SPT algorithms described in Chapter 2. Because our new algorithm can also be derived from a physical ball-and-string model, we will occasionally refer to it as the ball-and-string algorithm.

In the next section, we will first give an example of an SPT recalculation problem from which we will obtain some insight. In Section 3.4, we describe our algorithm in detail. Section 3.5 shows the algorithm works, proves its correctness, and analyzes its complexity. Section 3.6 presents simulation results illustrating the observed complexity of the new algorithm. Concluding remarks are given in Section 3.7.

3.2 Example

We introduce informally the SPT recomputation problem with a simple example. With this example, we wish to show how conventional search criteria used in conventional SPT algorithms might not be all that efficient when used in dynamic SPT algorithms.

Figure 3-1 shows a network comprising six nodes, where each node is labeled with a letter (A to F). The links between the nodes are illustrated with either dotted or solid lines, and the number next to each line is the weight of that edge. When the link between nodes A and C has a weight of 8, the SPT rooted at node A consists of the solid lines in the figure; the shortest distance of each node is the number inside each node.

At some point in time, the edge between nodes A and C changes its weight from 8 to 2. This weight change will significantly alter the structure of the SPT. The new SPT is shown in Figure 3-2. We will follow step by step the different computations that one needs to do in order to dynamically change the tree corresponding to the old SPT into the new SPT. The entire computation will be done using two different methods. The first method uses the familiar notion of shortest distance as used by the Dijkstra algorithm (and its dynamic versions). The second method employs a new notion (that of maximum decrement) which will prove to be more efficient for
First of all, it is clear that the edge between $A$ and $C$ (which decreases its weight) will remain in the SPT and that the shortest distance of node $C$ will decrease to 2. Because node $C$ has a shortest distance of 2, it can further decrease the potential shortest distance of nodes $B$ (from 7 to 6) and $E$ (from 10 to 7) by becoming their parent in the new tree. Now we have the choice of continuing the modification of the tree by following node $B$ or following node $E$.

If we give preference to nodes with the smallest potential distances (similar to the Dijkstra algorithm), we will choose node $B$ because its potential shortest distance 6 is smaller than the potential shortest distance of node $E$ (7). When we are at node $B$, its parent in the tree is changed from $A$ to $C$ and its distance label is reduced to 6. Since node $B$ has a child $D$, the distance label of $D$ is now reduced from 11 to 10. Since $D$ does not have a directed edge to another neighbor, its distance change does not affect any other node. Then, we return to node $E$ and decrease its distance label to 7 by making it a child of $C$. Since $F$ is the child of $E$, its distance label is reduced from 12 to 9. Finally, node $D$ decreases its distance label from 10 to 9 by becoming a child of $E$.

On the other hand, if after $C$'s distance label change, we decide to modify $E$ before $B$ ($E$ has the the greatest distance label decrease), the total computation becomes
simpler. In this case, the parent of $E$ becomes $C$ and its distance label becomes 7. The distance label of its child $F$ becomes 9. We now have a choice of modifying node $B$ or node $D$. Since node $D$ has the largest potential shortest distance decrease (from 11 to 9, rather than from 7 to 6), node $D$ will become a child of node $E$ and acquire a distance label of 9. Finally, $B$ will become a child of $C$ and acquire a distance label of 6.

Notice that when nodes $E$ and $D$ were chosen ahead of node $B$, the total computation is simpler. Specifically node $D$ is visited only once and its distance label changes directly from 11 to 9, rather than changing in two steps first from 11 to 10 and then from 10 to 9. The notion that we have used in the second computation is to give priority to nodes whose potential shortest distance decreases the most (or increases the least). This new notion is different from the idea inherited from Dijkstra's algorithm that the node with the lowest distance is always best. It is by generalizing this new notion that we obtain the algorithm described in this chapter.

### 3.3 Problem Formulation

In this section, we will first introduce some graph theoretic notations. We will then formally define the shortest path tree (SPT) problem in terms of a linear programming
framework. Using the duality principle of linear programming, we can recast the SPT problem into an equivalent ball-and-string model. With the ball-and-string model, the dynamics of our SPT algorithm can be understood in an intuitive way.

3.3.1 Shortest Path Tree

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a directed graph where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges in the graph. Let $N$ denote the total number of nodes in $\mathcal{V}$ and $E$ the total number of edges in $\mathcal{E}$. The graph $\mathcal{G}$ consists of a root node denoted by $n_0$, and all other nodes $n_1, ..., n_{N-1}$ are assumed to be reachable from $n_0$ by a directed path in $\mathcal{G}$. Each edge $e_i$ ($i = 0, ..., E - 1$) has an associated weight (cost) of $w_i$, which is assumed to be positive.

A rooted tree $\mathcal{T}$ is a subgraph of $\mathcal{G}$ such that every node in $\mathcal{G}$ is reachable from the root $n_0$ through a unique directed path using only edges in $\mathcal{T}$. The length of a path is the sum of the weights of the edges in that path. The distance of a node $n_i$ ($i = 1, ..., N - 1$) in $\mathcal{T}$ is the length of the directed path in $\mathcal{T}$ connecting $n_0$ to $n_i$. The shortest distance of a node $n_i$ is the length of a shortest path in $\mathcal{G}$ connecting $n_0$ to $n_i$. A tree $\mathcal{T}$ is said to be a shortest path tree (SPT) for $\mathcal{G}$ if the distance of every node $n_i$ in $\mathcal{T}$ is the shortest distance of $n_i$ in $\mathcal{G}$. Note that there can be multiple SPTs in a graph, but the shortest distance of any node is unique.

3.3.2 Linear Programming Framework

It is well known that the SPT problem can be formulated as an optimization problem in a linear programming framework.\footnote{There are both primal and dual formulations of the SPT problem in a linear programming framework. We present here the dual formulation to motivate the ball-and-string model.} To see this, for every directed edge $e_i$ connecting node $n_j$ to node $n_k$, we create a (row) vector $\bar{v}_i$ of dimension $N$ (number of nodes) which contains all 0's except a $-1'$ at the $j^{th}$ entry and a $+1'$ at the $k^{th}$ entry. The concatenation of all such vectors $\bar{v}_i$ results in a connectivity matrix $\bar{A}$ (of dimension $E \times N$). Let $\bar{u}$ be an $N$-dimensional (column) vector such that its $i^{th}$ entry $u_i$
corresponds to a potential distance of node \( n_i \). If \( u^*_i \) is the shortest distance of node \( n_i \) (\( u^*_i = 0 \)), then clearly \( u^*_j - u^*_i \leq w_{ij} \), where \( w_{ij} \) is the weight of the edge connecting \( n_i \) to \( n_j \). Thus, if \( \bar{w} \) denotes the (column) vector of edge weights and \( \bar{u}^* \) represents the vector of shortest distances, then \( \bar{u}^* \) must satisfy the following feasibility constraint:

\[
\mathbf{A} \bar{u} \leq \bar{w} ; \quad u_0 = 0 .
\]  

(3.1)

Clearly, there are infinitely many vectors \( \bar{u} \) that satisfy the above matrix inequality and only one of them corresponds to the vector of shortest distances \( \bar{u}^* \). However, it can be shown that \( \bar{u}^* \) can be found by solving the following linear optimization problem:

\[
\max (\bar{b}^T \bar{u})
\]  

subject to the feasibility constraint given by Equation (3.1), where \( \bar{b} \) is an \( N \)-dimensional vector with each entry equal to 1. Note that \( \bar{b}^T \bar{u} \) represents the sum of the potential distances of all nodes.

The feasibility constraint of Equation (3.1) ensures that the difference of the distances \( u_i \) and \( u_j \) associated with nodes \( n_i \) and \( n_j \) is always no greater than the weight of the edge connecting the nodes. This is clearly the case in an SPT where the distance \( u_i \) is the length of a shortest path from \( n_0 \) to \( n_i \). Interestingly, the vector of shortest distances can be obtained by maximizing the sum of the potential distances \( u_i \) of all nodes, while respecting the feasibility constraint.

In order to determine an SPT from the solution \( \bar{u}^* \) of the above optimization problem, one can substitute \( \bar{u}^* \) into the feasibility constraint \( \mathbf{A}^T \bar{u} \leq \bar{w} \). If the row \( i \) of the matrix inequality is an equality, it means that edge \( e_i \) can be used in an SPT.
3.3.3 Physical Interpretation of SPT in a Ball-and-String Model

There is an intuitive way of interpreting the linear programming formulation of the SPT problem using a ball-and-string model. This interpretation has been used in the past to give an intuitive understanding of static SPT algorithms [21, 4] such as Dijkstra or the auction algorithm [4]. To the best of our knowledge, this is the first time the ball-and-string model is used to understand and derive a dynamic SPT algorithm.

Consider a collection of balls, each of which corresponds to a node in $\mathcal{V}$. Each edge $e_i$ with weight $w_i > 0$ connecting nodes $n_j$ and $n_k$ in $\mathcal{G}$ is associated with an inelastic string of length $w_i$ connecting the balls representing $n_j$ and $n_k$. Let $u_i$ denote the Euclidean distance between the balls representing $n_0$ and $n_i$. In the following, the terms "ball" and "node" will be used interchangeably.

Since none of the strings is allowed to be stretched beyond its length, it is clear that in this illustration, the vector of Euclidean distances $\bar{u}$ always satisfies the feasibility constraint (3.1). Moreover, a string becomes "tight" as soon as the Euclidean distance between the two nodes attached equals its length. The optimal solution of (3.2) is then achieved by sequentially pulling the balls (in the same direction) so as to maximize their distances from the ball representing the root node $n_0$. In other words, when the balls are stretched out from $n_0$ as much as possible, their distances form a vector $\bar{u}^*$ that corresponds to the optimal solution. If there is only one SPT in the graph, the strings that are tight will then define the SPT. If there are multiple SPTs, each string that is tight will correspond to an edge that is part of some SPT. Our algorithm can be easily extended to compute all such edges.

In fact, the execution of the original (static) Dijkstra algorithm can also be interpreted in terms of the above ball-and-string model [3]. Let us imagine that the ball corresponding to the root node $n_0$ is anchored at a fixed position, while other balls are "floating" in the sense that their attached strings are not tight. All the balls are initially in the same position. In the first iteration, the ball connected to $n_0$ with the
shortest string is first pulled away from \( n_0 \) (always in the same direction) and will be anchored along with \( n_0 \).

In the second iteration, the floating ball that has a string connected to the anchored nodes and is closest (through that string) to \( n_0 \) is then pulled away (along the same direction as before) until that string becomes tight and the ball is anchored. The procedure is repeated until all floating balls are anchored. The Euclidean distance of an anchored ball from the root ball then represents its shortest distance.

### 3.4 Dynamic SPT Algorithm

#### 3.4.1 Algorithm Execution in the Ball-and-String Model

Recall that in computing a new SPT from an outdated SPT, our objectives are to minimize the computational complexity and the changes to the topology of the SPT. In fact, the ball-and-string model gives us useful insights into how an efficient dynamic SPT algorithm with these objectives should be designed. Before we present a formal specification of our algorithm, we next describe its execution in terms of the ball-and-string model.

Let us now imagine that the root node \( n_0 \) is anchored at a fixed position and the rest of the nodes are allowed to drop under gravity. Since gravity pulls down on these nodes as much as possible, the resulting Euclidean distance of a node \( n_i \) from the anchored root node is equal to the shortest distance of \( n_i \). When the length of a string increases, the ball attached to the lower end of the string (as well as others) will fall down for some distance until at least one of its attached string becomes tight. When the length of a string decreases, the ball attached to the lower end of the string (as well as others) will be raised accordingly. After the length of the string is modified, the hanging ball-and-string model always represents the set of valid SPTs.

Figure 3-3 illustrates the ball-and-string models for the networks in Figure 3-1 and Figure 3-2. The straight lines represent the tight strings and the curly lines represent the loose strings. The shortest paths from node A to all other nodes are composed of
the straight lines in the figures. When the link distance between A and C is reduced from 8 to 2, node C is “raised” to a higher level. It can be seen from the figure that some loose strings in the original model now become tight and that the affected nodes rearrange themselves in a natural way into their optimal positions. When the length of the string between A and C is reduced gradually, the sequence of distance changes that takes place with this ball-and-string model corresponds to the optimal execution we discussed for our example at the end of section 3.2.

Figure 3-3: The ball-and-string models for the networks in Fig. 3-1 (left) and Fig. 3-2 (right).

The execution of our algorithm simulates the dynamics of the balls when a string gets longer or shorter. When there are multiple strings that change lengths, the algorithm separates the length increments and decrements of the strings into two sets, and then simulates the dynamics for one set after the other. Within each set, the changes in the string lengths are simulated as if they took place at the same rate. In other words, all the balls that will change distances will move at the same rate.
The efficiency of our algorithm is due to the fact that the changes in distances are propagated to all affected nodes in a natural order and in a most economical way.

If the directed edge $e_k$ connecting $n_i$ to $n_j$ has a different weight than the edge $e_l$ connecting $n_j$ to $n_i$, then we need to stretch our imaginations a little and change the inelastic string for a magic elastic string. This elastic string representing the two edges has a nominal zero length but can be stretched up to a certain distance when pulled. When $n_i$ is above $n_j$ (due to gravity), the elastic will stop stretching at length $w_k$; when $n_j$ is above $n_i$, it will stop stretching at length $w_l$.

3.4.2 Algorithm Specification

For each directed edge $e \in \mathcal{E}$, we use $W(e)$ to denote the weight (distance) associated with $e$, $S(e)$ and $E(e)$ to denote respectively the source node and the end node of $e$. The length or distance of a directed path is the sum of weights of the edges on the path. Given a set of nodes $\mathcal{N} \subseteq \mathcal{V}$, we associate with it two sets of edges: $I(\mathcal{N}) = \{ e \in \mathcal{E} \mid E(e) \in \mathcal{N} \}$ (the set of edges directed into the nodes in $\mathcal{N}$) and $O(\mathcal{N}) = \{ e \in \mathcal{E} \mid S(e) \in \mathcal{N} \}$ (the set of edges directed out of the nodes in $\mathcal{N}$). The variable $\text{root}(\mathcal{G})$ denotes $n_0$, the root node of the SPT. These parameters only depend on the topology of the network, and their values do not change during the execution of the algorithm.

A tree data structure, denoted by $\hat{T}$, is maintained by our algorithm to keep track of an existing potential shortest path tree. In particular, every node $n$ in the graph $\mathcal{G}$, along with its parent attribute $P(n, \hat{T})$, a list of children $C(n, \hat{T})$, its distance attribute $D(n, \hat{T})$, and a Boolean attribute $V(n, \hat{T})$ with value floating or anchored, is present in $\hat{T}$. This data structure $\hat{T}$ changes progressively during the computation, and when the execution of the algorithm is completed, it will represent an SPT, and each node $n$ will have its Boolean attribute $V(n, \hat{T}) = \text{anchored}$.

In addition to the data structure $\hat{T}$, our algorithm also maintains a list $Q$ that temporarily contains a subset of nodes together with three attributes. In particular, each element in $Q$ is of the form $\{ n, (p, d, \Delta) \}$, where $p$ denotes a potential parent for node $n$, $d$ denotes a potential distance for node $n$, and $\Delta$ denotes the potential
distance change for \( n \).

The instruction \( \text{ENQUEUE}(\mathcal{Q}, \{ n, (p, d, \Delta) \}) \) adds one more element to \( \mathcal{Q} \). If node \( n \) is already in \( \mathcal{Q} \), the new attributes \( (p, d, \Delta) \) will replace the old ones only if the new \( \Delta \) is smaller than the old one. At any instant, only one set of attributes is maintained for each node in \( \mathcal{Q} \). When an \( \text{EXTRACTMIN}(\mathcal{Q}) \) instruction is executed, the element with the smallest \( \Delta \) is selected and removed from \( \mathcal{Q} \). If there is more than one element with the minimum \( \Delta \), the smallest distance \( d \) is used as a tie breaker. If more than one element has the smallest \( d \) among the elements with the smallest \( \Delta \), then any of these elements can be chosen by \( \text{EXTRACTMIN} \). A \( \text{REMOVE}(n, \mathcal{Q}) \) operation removes the entry for node \( n \) from the list \( \mathcal{Q} \).

The algorithm also uses two more auxiliary subroutines. The function \( \mathcal{B}_{\text{max}}(n, \mathcal{T}) \) denotes the set of nodes (including \( n \)) that are descendents of \( n \) in the tree \( \mathcal{T} \). This set can be easily computed by descending directly down \( \mathcal{T} \). Likewise, the function \( \mathcal{B}_{\text{val}}(n, \mathcal{T}) \) denotes the subset of \( \mathcal{B}_{\text{max}}(n, \mathcal{T}) \) that is reachable from \( n \) through some path of anchored nodes. It can be easily computed by descending directly down \( \mathcal{T} \) and stopping when a node found in the path is floating.

First we formally define these subroutines using pseudocode. Then we will describe the algorithm itself with pseudocode.

**Subroutine 1:** \( \mathcal{N} \gets \mathcal{B}_{\text{val}}(E(e_i), \mathcal{T}) \)

\[
\begin{align*}
\mathcal{N} & \gets \emptyset \\
\mathcal{A} & \gets E(e_i) \\
\text{while } \mathcal{A} \neq \emptyset & \\
& \quad n \gets \text{EXTRACT } (\mathcal{A}) \\
& \quad \mathcal{N} \gets \mathcal{N} \cup \{ n \} \\
& \quad \forall n' \text{ s.t. } \mathcal{P}(n, \mathcal{T}) = n' \\
& \quad \quad \text{if } \mathcal{V}(n', \mathcal{T}) \neq \text{floating}
\end{align*}
\]
\[ \mathcal{A} \leftarrow \mathcal{A} \cup \{n'\} \]

Subroutine 2: \[ \mathcal{N} \leftarrow \mathcal{B}_{\max}(E(e_i), \bar{T}) \]

\[ \mathcal{N} \leftarrow \emptyset \]
\[ \mathcal{A} \leftarrow E(e_i) \]
while \( \mathcal{A} \neq \emptyset \)
    \[ n \leftarrow \text{EXTRACT (A)} \]
    \[ \forall n' \text{ s.t. } P(n, \bar{T}) = n' \]
    \[ \mathcal{A} \leftarrow \mathcal{A} \cup \{n'\} \]

The algorithm itself contains an initialization procedure (step 1) and an iterative loop (steps 2-4). If there are multiple edge weight changes, only the weight increments or the weight decrements can be handled at a time. After the pseudocode, we give an informal description of the algorithm.

The Ball-and-String Algorithm

STEP 1: Initialization

(A) Static Version
\[ \forall (n \in \mathcal{V}) \neq \text{root}(\mathcal{G}) \]
\[ P(n, \bar{T}) \leftarrow \emptyset \]
\[ D(n, \bar{T}) \leftarrow 0 \]
\[ V(n, \bar{T}) \leftarrow \text{floating} \]
\[ C(n, \bar{T}) \leftarrow \emptyset \]
ENQUEUE (\( \mathcal{Q} \), \{root(\mathcal{G}),(\emptyset,0,0)\})
(B) Dynamic Version (applicable when an outdated SPT exists)

Case 1 (Edge $e_i \in \mathcal{E}^+$ increases its weight by $\Gamma_i$):

$\mathcal{N} \leftarrow \emptyset$

$\forall e_i \in \mathcal{E}^+$  /* for each edge whose weight has increased */

$W(e_i) \leftarrow W(e_i) + \Gamma_i$  /* update its weight */

if $S(e_i) = P(E(e_i), \hat{T})$  /* if $e_i$ is in $\hat{T}$ */

$\mathcal{N} \leftarrow B_{val}(E(e_i), \hat{T})$

/* consider all descendent nodes of $E(e_i)$ reachable via anchored nodes in */
/* the existing SPT */

$\forall n \in \mathcal{N}$

$V(n, \hat{T}) \leftarrow \text{floating}$  /* make all such descendent nodes floating */

$\hat{\mathcal{N}} \leftarrow \hat{\mathcal{N}} \cup \mathcal{N}$

$\forall e \in \mathcal{I}(\hat{\mathcal{N}})$  /* for each edge directed into a floating node */

if $V(S(e)) = \text{anchored}$  /* if its source node is anchored */

$\text{newdist} = D(S(e), \hat{T}) + W(e)$

/* keep track of its end node's potential new distance */

$\Delta = \text{newdist} - D(E(e), \hat{T})$

/* and the increase in distance */

ENQUEUE($Q$, \{E(e), (S(e), newdist, $\Delta$)\})

/* update the node with the new distance and new parent in the list $Q$ */

Go To Step 2

Case 2 (Edge $e_i \in \mathcal{E}^-$ decreases its weight by $\Gamma_i$):

$\forall e_i \in \mathcal{E}^-$  /* for each edge whose weight has decreased */

$W(e_i) \leftarrow W(e_i) - \Gamma_i$  /* update its weight */

$\text{newdist} \leftarrow D(S(e_i), \hat{T}) + W(e_i)$

if $\text{newdist} < D(E(e_i), \hat{T})$  /* if the reduced weight of $e_i$ yields a smaller */
/* distance for $E(e_i)$ */

Go To Step 2
\[ \Delta \leftarrow \text{newdist} - D(E(e_i), \hat{T}) \quad /* \text{keep track of the difference} */ \]

ENQUEUE \((Q, \{E(e_i), (S(e_i), \text{newdist}, \Delta)\})\)

/* update the node with the new distance and new parent in the list \(Q\) */

Go To Step 2

STEP 2: Node Selection

if \(Q = \emptyset\)

Terminate

else

\(\{y, (x, d, \Delta)\} \leftarrow \text{EXTRACTMIN}(Q)\)

/* select and remove the element from the list with the smallest \(\Delta\) */

\(C(x, \hat{T}) \leftarrow C(x, \hat{T}) + \{y\}\) /* update tree structure, by adding the new edge */

\(C(P(y, \hat{T}), \hat{T}) \leftarrow C(P(y, \hat{T}), \hat{T}) - \{y\}\) /* and deleting the old edge */

\(P(y, \hat{T}) \leftarrow x\) /* update the parent attribute of the selected node */

\(N \leftarrow B_{\max}(y, \hat{T})\) /* consider the selected node and all its descendents */

STEP 3: Distance Update

\(\forall n \in N\)

\(D(n, \hat{T}) \leftarrow D(n, \hat{T}) + \Delta\)

/* update the distance of the selected node and each descendent considered */

\(V(n, \hat{T}) = \text{anchored} \quad /* \text{each such node now has the correct distance} */\)

if \(n \in Q \land P(n, \hat{T}) \neq P(n, Q)\)

REMOVE\((n, Q)\) /* no need to update its distance further */

STEP 4: Node Search

\(\forall e \in O(N)\) /* consider all edges directed out of the selected subset of nodes */

if \((D(E(e), \hat{T}) > D(S(e), \hat{T}) + W(e)) \lor (V(E(e)) = \text{floating})\)
/* if the reduced distance of a selected node also reduces the distance of */
/* another node */
newdist ← D(S(e), \hat{T}) + W(e)
\Delta ← newdist - D(E(e), \hat{T})
/* choose the smaller distance as the new potential distance */
ENQUEUE (Q, \{E(e), (S(e), newdist, \Delta)\})
/* update the node with the new distance and new parent in the list Q */
Go to Step 2

3.4.3 Informal Discussion of the Algorithm

To help understand the algorithm, here we shall give an informal discussion of its execution. As the execution is the same for the static version and the dynamic version after initialization, we shall only discuss the dynamic version, since the initialization of the static version is trivial. Also note that the execution of the static version of this algorithm is equivalent to Dijkstra's algorithm.

First, the goal of the initialization phase is to identify those nodes that may be affected by the changes in the cost metrics. The potentially affected nodes are those that are no longer connected to the root through the same shortest path as before. Moreover, in the initialization, we only want to select a minimal set of such nodes whose changes in distance will be subsequently propagated to their descendents in the original tree, which can be updated.

It is assumed that in the existing (outdated) SPT, all nodes were anchored. When there are weight increments, the potentially affected nodes will be marked floating in the initialization. As mentioned earlier, the algorithm works by first updating the SPT with all edges that increase their weights (case 1) and then updating the SPT with all edges that decrease their weights (case 2).

In case 1, after updating each edge $e_i$ with the new increased weight, we check if the edge is in the existing SPT. If it is, we select its end node $E(e_i)$ and all descendents
nodes of $E(e_i)$ that are reachable via anchored nodes in the existing SPT. All such nodes will be marked floating and included in a set $\hat{N}$ for further updating. The intuition is that the set $\hat{N}$ covers all the affected nodes. Moreover, we only have to consider those nodes that are directly attached to anchored nodes in the existing SPT (their descendent nodes will be updated in subsequent steps of the algorithm).

Now for each anchored node that is attached to a link directed into a floating node in $\hat{N}$, we compute the potential distance $\text{newdist}$ of the floating node by adding the distance of the anchored node and the weight of the edge connecting the two nodes. Moreover, we keep track of the difference $\Delta$ between the old distance and the potential new distance. The floating node with $\Delta$, its potential distance $\text{newdist}$, and its potential new parent (the anchored node) will be enqueued in the list $Q$.

Case 2 is a bit simpler in the initialization step. After updating each edge $e_i$ with the new decreased weight, we compute the potential new distance $\text{newdist}$ of its end node $E(e_i)$ by adding the distance of its source node $S(e_i)$ and the new weight $W(e_i)$. And if this potential new distance is in fact smaller than the old distance of $E(e_i)$, then we enqueue to the list $Q$ the node $E(e_i)$ with its potential new parent $S(e_i)$, its potential new distance $\text{newdist}$, and the difference $\Delta$ between its old distance and $\text{newdist}$.

After the initialization step, we extract from the list $Q$ in step 2 an element with the smallest (least positive or most negative) $\Delta$. In other words, we select a node with the smallest increase in distance (in case 1) or largest decrease in distance (in case 2). This selected node is then updated with its new parent indicated in $Q$ and the structure of $\hat{T}$ is modified accordingly to reflect the new child-parent relation. Moreover, the selected node together with all its descendent nodes (denoted by the set $\mathcal{N}$) in the existing tree $\hat{T}$ will now have their correct distances updated as their old distances incremented by $\Delta$ (step 3). In addition, these nodes will be marked anchored. And if any such node is already in the list $Q$ (unless its parent attribute in $Q$ is the same as its parent in $\hat{T}$, indicating that further change will take place) it will be removed from $Q$ and no longer considered by the algorithm.

In step 4, we consider each node $E(e_i)$ that is attached to an edge $e_i$ directed out
of a node $S(e_i)$ in $\mathcal{N}$. If the potential new distance $newdist$ of $E(e_i)$ (which is equal to $W(e_i)$ plus the distance of $S(e_i)$) is smaller than its old distance, or if $E(e_i)$ has been marked floating, we then enqueue in the list $Q$ the node $E(e_i)$ with its potential new distance, change in distance, and its potential new parent $S(e_i)$ for further updating. After this step is completed, the execution of steps 2 and 3 will be repeated until the list $Q$ is empty.

### 3.5 Algorithm Correctness and Complexity

In this section, we will prove the correctness of the algorithm and bound its asymptotic complexity. We start by putting forth some definitions that will help us explain how the algorithm manipulates the graph and the data structures. We will then prove some properties of SPTs when some links change weight. Along with some invariants of the algorithm, these properties will be used to prove that the algorithm always produces a correct output. Similarly, the framework used to describe the correctness of the algorithm will be used to calculate its worst case complexity.

The object of the algorithm is to compute a new SPT after the weights of the graph change. Let $\mathcal{G}$ and $\mathcal{G}'$ be the old and new network respectively, and $\Gamma$ is the set of weight changes that occur in $\mathcal{G}$. For brevity, we will refer to the event when the topology of the network changes as $\beta$. The input to the algorithm is $\mathcal{G}$, $\Gamma$, as well as an old SPT tree ($T$) valid for $\mathcal{G}$. The desired output of the algorithm is a new SPT tree ($T'$) valid for $\mathcal{G}'$. Furthermore, when there are multiple new SPTs for $\mathcal{G}'$, the desired output of the algorithm $T'$ is the new SPT that differs by the minimum number of edges from $T$.\footnote{\footnotetext{$T'$ is not necessarily unique since there can still be several SPTs that differ from $T$ by the minimum number of edge changes} \textsuperscript{2}}

We first take a look at how the optimal SPT tree is affected by the weight changes. We separate the full set of nodes in the graph $\mathcal{N}$ into $k$ smaller subsets $\mathcal{K}_1$, $\mathcal{K}_2$, ... $\mathcal{K}_k$. Within each subset (referred to as branch), the structure of the SPT can remain unchanged after the link weight changes. The branch containing the root node in $\mathcal{N}$
Definition 3  Given an old SPT $T$ and a new network topology $G'$, two nodes $n_j$ and $n_k$ are said to be in the same branch $K_i$ if they are connected in both the old SPT $T$ and in any new SPT by some edge that does not change weight during event $\beta$.

This definition of can be used transitively to find all the nodes in a given branch. For a given old $T$ and new $G'$, the definition of branch gives unique sets of nodes. This can be easily seen graphically by using the ball and string model. We first represent $G'$ using the ball and string model, and then discard all the string (edges) that are not tight. The remaining tight edges gives us all the possible edges that can be part of some SPT for $G'$. Clearly this set of edges is unique. If we now overlap the old SPT $T$ onto this set of edges, we immediately obtain our branches $K_i$. The branches will consists of the nodes that are connected by overlapping edges of $T$ (that did not change weight) and the tight edges of the ball-and-string model.

Moreover, for any two nodes $n_j$ and $n_k$ in the same branch $K_i$, we next show that after the event $\beta$, both nodes must have changed their optimal distance by the same amount $\Delta$. For example, if the optimal distance of $n_j$ changes from 8 to 5, it must follow that if the old distance of $n_k$ is 12, it's new distance is 9.

Lemma 12  The change in the optimal distances that takes place after the weight change event $\beta$ is constant within a given branch $K$.

Proof:  We know that if nodes $n_j$ and $n_k$ are members of the same branch $K$, the SPT path that connects the two remains unchanged. Somewhere along that path there exists some node $n_i$ that is a common ancestor of both $n_j$ and $n_k$ in both $T$ and any new SPT. Since the shortest directed path from $n_i$ to either $n_j$ or $n_k$ has not changed, the difference between the optimal distances of the three nodes $n_i$, $n_j$, and $n_k$ must have remained the same. Therefore, the change in optimal distance must be constant within the same branch $K$. $\square$.

We call the potential of a branch $K$ the change in optimal distance that has taken place in every node of the branch. The potential of a branch will be denoted by $\phi(K)$. 


Different branches $\mathcal{K}$ can be ordered and grouped according to their potential. Except for the root branch $\mathcal{K}_0$, the branches are ordered in order of increasing potential, $\phi(\mathcal{K}_i) \leq \phi(\mathcal{K}_{i+1})$. If two branches have the same potential, then the branch containing the node with the smallest optimal distance in $\mathcal{G}'$ is ordered first. If two branches still cannot be ordered according to the above rule, they can be ordered in any way relative to each other and are treated indistinguishably. In other words, if two such branches carry the indices $i$ and $i+1$, the symbol $\mathcal{K}_i$ can refer to either of the two branches. This ordering of branches will be shown to be the same order in which the algorithm manipulates these branches. Now we are ready to make the most important observation about the role of branches in recomputing an SPT.

Lemma 13 In any branch $\mathcal{K}_i$ that is not the root branch, there exists exactly one node $n_r \in \mathcal{K}_i$ who has a parent (in any new SPT tree) in another branch $\mathcal{K}_j$, such that $\phi(\mathcal{K}_j) \leq \phi(\mathcal{K}_i) - \Gamma_e$ ($\Gamma_e$ is the weight change of edge $e$, the edge connection $n_r$ to its new parent), and has the smallest optimal distance of any node in the branch. Every other node $n \neq n_r$ in $\mathcal{K}$ has the same parent in $\mathcal{T}'$ as in $\mathcal{T}$ and is connected to that parent through an edge that does not change weight.

Proof: A given branch $\mathcal{K}_i$ contains $l$ number of nodes. By definition, every pair of nodes in this branch is connected (through $\mathcal{T}$) by a set of unchanged links. Because there is connectivity there must be at least $l - 1$ such unchanged links; because of the uniqueness of the path, there can be at most $l - 1$ such links. The ends of any of these $l - 1$ unchanged edges correspond to the $l - 1$ nodes in $\mathcal{K}_i$ that have the same parent in $\mathcal{T}'$ as in $\mathcal{T}$.

The remaining node $n_r$ must therefore have a parent in some other branch $\mathcal{K}_j$. Since it can change its parent to some other node not in $\mathcal{K}_i$ without disrupting the connectivity in $\mathcal{K}_i$, $n_r$ must be an ancestor of all the other nodes in $\mathcal{K}_i$, and thus must have the smallest optimal distance. Now, let us assume that the old optimal distance of $n_r$ is $u_r$ and the new one is $u'_r$. Likewise the old optimal distance of the new parent $n_p$ is $u_p$ and the new one is $u'_p$. Because of optimality, we know that $u'_r - u'_p = W'(n_p, r_p)$. Because of feasibility, it is also true that $u_r - u_p \leq W(n_p, r_p)$.
It follows that \((u'_p - u_p) \leq (u'_r - u_r) - \Gamma_e\), or \(\phi(K_j) \leq \phi(K_i) - \Gamma_e\). \(\square\)

We can think of a branch \(K\) as a subtree of the original SPT \((T)\) that does not need to change internally but only changes its position with respect to the rest of the tree. The node \(n_r \in K\) can be thought of as a mini-root for that branch since it is the one node that is the ancestor of all the other nodes in the branch and by changing parent attaches the entire branch \(K\) into a new position with respect to the other branches. The mini-roots of each branch are the only nodes that must absolutely change their parent node in order for the new SPT to be valid. The new SPT with the minimum changes \((T')\) is the same as the old SPT \(T\) except that the mini-roots have changed their parents. We will show that our algorithm recomputes the SPT by rearranging these branches in the minimum number of iterations, without modifying their internal structure.

### 3.5.1 Single Edge Weight Change

We first prove the case where only one edge in \(G\) changes weight. For this case, we show that our algorithm recomputes a new SPT by reordering, in the order described above \((K_1, K_2, ...)\), the different branch from the original SPT \(T\). We now make the following definition:

**Definition 4** A branch \(K_i\) is said to be consolidated at some instant \(\alpha\) of the execution of the algorithm if the distance and parent attributes of all the nodes in \(K_i\) are the same as in \(T'\) and will no longer change after \(\alpha\).

We will show that at each iteration, a new branch is identified and consolidated. If there are \(k\) branches in the graph (excluding the root branch), after \(k\) iterations, the algorithm is done.

We will show this using inductive reasoning. We first prove the base case.

**Lemma 14** 1) After initialization, all the mini-roots of the branches that have a parent in the root branch in \(T'\) are enqueued with their new optimal distances in \(Q\).

2) After the first iteration, branches \(K_0\) and \(K_1\) are consolidated.
Proof: We look at the two possible cases:

For case 1 (weight edge increases), the initialization marks as 'floating' all the nodes that are descendents of the edge that has increased its weight. The set of nodes that are still marked as 'anchored' comprises the root branch (which is always consolidated). The set of nodes that are marked 'floating' comprises all the branches that need to be modified in the new tree. Every one of these nodes searches for a parent in the root branch. If any one of the mini-roots has an optimal parent in the root branch, it will find it at this stage and the mini-root will be enqueued in $Q$ with its optimal distance. At the beginning of the first iteration, the node with the lowest $\Delta$ (and $d$ to break ties) label is extracted from $Q$. According to lemma 13, this node must be the mini-root of the branch with the lowest potential. If there are several branches with this same potential, the one with lowest distance label $d$ is chosen. This is branch $K_1$. During the rest of the iteration the rest of the branch is updated (changing the optimal distance by the same amount) and thus the whole branch is consolidated.

For case 2 (edge weight edge decrement), the algorithm enqueues in $Q$ the end node of the edge that decreases its weight. Any other node that might have to change parent will be a descendent of this node. Therefore, right after initialization, all the nodes in $T'$ (in this case, just one) that have a parent in the root branch are already enqueued in $Q$. The initially enqueued node will decrease its optimal distance by at least as much as any other node in $G$. This node is the mini-root of branch $K_1$. During the first iteration the entire $K_1$ branch will be consolidated.

Now we prove the inductive step:

Lemma 15 If:

1) At the end of the $i$th iteration, all the nodes that have parents (in $T'$) in branches $K_0$ to $K_i$ are enqueued with optimal distances in $Q$.

2) During the $i + 1$ iteration of the algorithm, branch $K_{i+1}$ is consolidated.

Then:

1) At the end of the $i + 1$ iteration, all the nodes that have parents (in $T'$) in
branches $K_0$ to $K_{i+1}$ are enqueued with optimal distances in $Q$.

2) During the $i+2$ iteration of the algorithm, branch $K_{i+2}$ is consolidated.

Proof:

By assumption 2), during the $i+1$ iteration, $K_{i+1}$ is consolidated. Therefore the nodes in the branch acquire their optimal distance label. During that iteration, all the nodes in $K_{i+1}$ check their outgoing edges; if through one of these edges, an end node $n_e$ can acquire an optimal distance, node $n_e$ is enqueued in $Q$ with its optimal distance and parent. Therefore by the end of the iteration, all the nodes in $T'$ that have a parent in $K_{i+1}$ are enqueued in $Q$. Because of assumption 1), this is true for all branches $K_0$ to $K_{i+1}$.

According to lemma 13, the mini-root of branch $K_{i+2}$ must have a parent in branches $K_0$ to $K_{i+1}$. Therefore, at the beginning of the next iteration $i+2$, this mini-root must be enqueued in $Q$. Since it must have the lowest $\Delta$ (and lowest $d$ among equal $\Delta$) in $Q$, it will be extracted with the optimal distance. During the rest of the iteration, the rest of $K_{i+2}$ is consolidated. □

Theorem 7 Let the data structure $\hat{T}$ be a valid SPT for graph $G$. The graph $G$ is modified into $G'$ by a single weight change $\Gamma$. After running the algorithm, the new $\hat{T}$ will contain a valid SPT for $G'$. Furthermore, this new SPT has the minimum number of edge changes compared to the old SPT.

Proof: The correctness follows directly from the inductive reasoning of lemmas 14 and 15. Because only one edge of the SPT is modified for every branch, we know that the algorithm is changing the minimum number of SPT edges.

3.5.2 Multiple Edge Changes

When $\Gamma$ consists of multiple edge changes, it is harder to determine the operation of the algorithm. The general case must be broken into 3 different cases where the algorithm will behave differently. Recall that the algorithm only allows either edge weight increments or decrements at a time.
Case 1: Weight Increments

When $\Gamma$ consists of weight increments, the multiple edge change case is pretty similar to the single edge case described earlier. The only difference is that at initialization, the set of nodes marked as 'floating' will be the union of the descendents of all the edges that increase weight. Since $\Gamma_e$ is nonnegative, lemma 13 still implies that every mini-root attaches itself to another branch with a smaller or equal potential, and the rest of the proof proceeds in the same manner as before.

Case 2: Parallel Weight Decrements

Now we consider the case where $\Gamma$ consists of weight decrements, where every edge that decrements is not a descendent in $T'$ of some other edge that decrements. In other words, when traveling down the new tree $T'$ (from the root node to any other node), we cannot encounter more than one edge that has changed weight.

In this case, all the edges that decrease weight will be attached to the root branch. Except for the mini-roots attached to the root branch, all the other mini-roots will still be attached to other branches with a smaller or equal potential ($\Gamma_e = 0$ everywhere else). Therefore, all the mini-roots that have a parent with a greater potential must be the ends of the edges that decrease weight. During initialization these nodes are enqueued with optimal distance into Q. After initialization, the algorithm proceeds as above.

Case 3: Serial Weight Decrements

Now we consider the remaining case where there might be a weight decrement on an edge $e_c$ that is a descendent in $T'$ of another edge $e_p$ that also decreases its weight. This scenario is much more complex because $e_c$ now has a negative $\Gamma_e$. Therefore, lemma 13 no longer guarantees that the end node $n_c$ of $e_c$ will attach itself to a branch with a smaller or equal potential. Furthermore, because $e_c$ is not attached to the root branch, unlike in the previous case, $n_c$ will not be enqueued with its optimal distance during initialization. Therefore, because lemma 13 no longer holds after initialization,
we can no longer guarantee that the algorithm will reconstruct the new tree in the defined order of branches \((K_1, K_2, \ldots)\).

For the moment, we just concentrate on the event where only two weight changes in series take place. When two such changes take place, the algorithm performs a set of operations with the same effect as if the algorithm had been executed in two separate occasions, first after the change of \(e_p\) and then after the change of \(e_c\).

To see this, we first define \(S_c\) to be the set of nodes that must be descendents of \(e_c\) in \(T'\). In other words, in every possible new SPT with minimum number of changes, the nodes in \(S_c\) are descendents of \(e_c\). We also define the sets \(K^p_i\) to be the branches defined when only the weight of \(e_p\) changes, \(K^c_i\) to be the branches defined when only \(e_c\) changes, and \(K^{pc}_i\) to be the branches defined when \(e_c\) changes in a new SPT that already accommodates the changes brought along by \(e_p\). As usual, \(K_i\) contain the branches for the complete operation, when \(e_c\) and \(e_p\) both change at the same time.

The first thing we observe is that all the nodes outside of \(S_c\) are not affected at all by the weight change in \(e_c\). Therefore, these nodes will be restructured just like in the previous cases. The branches \(K_i\) to which these nodes belong are placed in the correct positions in the right order relative to each other.

However, the nodes in \(S_c\) will not be necessarily reordered according to the \(K_i\) branch order. First, before \(n_c\) is extracted with its optimal distance, the nodes in \(S_c\) might be pre-reordered either as part of some branch \(K^c_i\) or some branch \(K^p_i\). Since \(n_c\) is by definition a child of some node not in \(S_c\), it will be eventually extracted with optimal distance. After this takes place, any nodes in \(S_c\) previously extracted in branches \(K^c_i\) will be updated with optimal distance (they already have the optimal parents). Then any node in \(S_c\) that was not previously extracted will be extracted in some branch \(K_i\). Finally the nodes in \(S_c\) previously reordered in \(K^p_i\) will now be reordered in \(K^{pc}_i\).

In other words, the nodes in \(S_c\) can change their parents once or twice. The restructuring that takes place inside \(S_c\) at first can follow the ordering of \(K^c_i\) in some areas and the reordering of \(K^p_i\) in others, as if the weight changes of \(e_p\) and \(e_c\) were acting independently. When \(n_c\) finally obtains its optimal distance, the nodes in the
latter area are reordered using branches $K_{i,c}$. The total sequence of operations is at worst the same as if all the nodes in $S_c$ were first restructured by $K_{i}^p$ and then by $K_{i,c}^p$. This is what would happen if the algorithm were executed independently for each change.

When there are many of such edge weight changes in series, we can apply the same reasoning recursively. As each end node of a changed edge receives its optimal distance information, it will initiate a new wave of restructuring into its descendent nodes in the new tree.

### 3.5.3 Complexity

From the proof of correctness, the complexity of the algorithm follows directly. For the single edge change, as well as well for case 1 and 2 of the multiple edge changes, the algorithm follows an easily determined number of steps.

Every edge departing from a node in non-root branches is visited exactly once. Furthermore, for every branch, its mini-root needs to be extracted from $Q$. The maximum size of $Q$ is the total number of nodes in the non-root branches. The maximum number of insertions and key decrements in $Q$ is the total number of edges whose source is a node in a non-root branch.

We let $\delta_c$ be the number of branches defined by the old tree $T$ and the new graph $G'$ (excluding the root branch). Let $\delta_n$ be the total number of nodes in these branches, and $\delta_e$ be the total number of edges whose sources correspond to these nodes. The number of operations of the algorithm can be expressed as follows:

- Algorithm iterations: $\delta_c$
- Extractions from $Q$: $\delta_c$
- Node visits: $\delta_n$
- Maximum size of $Q$: $\delta_n$
- Maximum number of insertions into $Q$: $\delta_n$

98
- Maximum number of removals from $Q$: $\delta_n - \delta_c$

- Maximum number of key decrements in $Q$: $\delta_c$

The exact complexity of the algorithm depends on how the underlying priority queue $Q$ is implemented. The simplest way of implementing it is using a linked list. A more efficient, as well as practical, way involves using binary heaps. In theory, using Fibonacci heaps should yield the lowest asymptotic complexity; however, in practice they are inefficient because of the large overhead involved in using them.

The following are the total resulting asymptotic complexities:

- Linked List: $O(\delta_c + \delta_c\delta_n)$
- Binary Heap: $O(\delta_c\log(\delta_n))$
- Fibonacci Heap: $O(\delta_c + \delta_n\log(\delta_n))$

Note that the best asymptotic complexity of existing dynamic algorithms is that of dynamic Dijkstra and its variations. For linked list structure, the above complexity is better than for dynamic Dijkstra ($((\delta_n)^2$). For the heap structures, the above complexity matches that of Dynamic Dijkstra. Nevertheless, the number of extractions $\delta_c$ performed by our algorithm is smaller than for any known algorithm that satisfies the above upper bounds (all algorithms that satisfy this bound, have extraction operations from a priority queue). Even though in the worst-case the smaller number of extractions does not improve the upper bound, in practice a lower number of extractions means that less nodes need to enter the priority queue, and the number of operations performed on the queue is smaller. The number of insertions can be as small as $\delta_c$, while for all the known algorithms that satisfy the above bound, the number of insertions is at least $\delta_n$.

Furthermore, the highest key (in this case $\Delta$) used in the priority queue of the new algorithm is at most the maximum weight of an edge. For dynamic Dijkstra, the maximum key can be as much as the length of the path from one side of the network to the other. Because the keys are much smaller, it is much easier to implement the priority queues (for example, by using bucket search).
For the third case of multiple edge changes (multiple decrements in series), it harder to calculate a bound on the maximum number of steps that will be executed by the algorithm. As explained earlier, every time that the end node of a changing edge weight obtains a better distance attribute, it will propagate a new wave of re-structuring into its descendents. Therefore, in the worst case, the algorithm performs as if the algorithm were executed once for each weight change. Each hypothetical execution takes places after an edge weight is changed, starting from the top-most to the bottom most. Therefore, when there are edge weight decrements in series, the worst case complexity is the same as if the changes were done independently, one by one. Unlike the case where the decrements are in parallel, there is no worst-case saving by grouping the changes together before the algorithm execution.

This anomaly in measuring the worst-case complexity can be illustrated once again by the ball-and-string model. It is easy to construct an example of a model where two string in series have to decrease their length, and no matter how the shortening is done, some balls will move, then stop, then move again, until finally stopping. In such an example, there is no way of decreasing the distance of the strings so that all balls will only move at most once. In the same way that the physical model has to reposition the balls more than once, our algorithm derived from this physical model needs to visit the corresponding nodes more than once during its execution.

3.6 Simulation Results

In this section we investigate via simulations the complexity of our new shortest path algorithm. The algorithm is compared with the other dynamic SPT algorithms derived in Chapter 2. These algorithms, which we use for comparison purposes, are the dynamic versions of the Bellman-Ford, D'Esopo-Pape, and Dijkstra algorithms (using the first incremental method). For a full treatment of these algorithms and on how to create dynamic SPT algorithms based on static ones, see Chapter 2.

The simulations in this chapter are performed using the same random network generator as described in Section 2.9. Please refer to that section for the details on
how the simulation was performed.

### 3.6.1 Event Generation

Once a random network is obtained, we simulate link failures and recoveries on that network. In every network, \((25 \cdot D)\) edges are randomly selected (with uniform probability). First, a link failure is simulated (the edge weight becomes infinity) on each of the edges, and later, each of the edges recovers its previous weight.

After each event, the shortest path tree is recomputed using our new algorithm as well as the three known algorithms described earlier. During the computation, we keep track of the number of operations that the algorithm performs. We increment the observed complexity of the algorithm every time any one of the following occurrences takes place:

- A comparison is made between the distances of two nodes sharing a common edge.
- A comparison is made between two elements in the queue.

### 3.6.2 Node Based Complexity

During the simulation, we compare the average observed complexity of the different dynamic SPT algorithms as the network size increases. For each network size, 40 different networks of that size are randomly generated \((L = 0.5, D = 7)\). For each network, 175 link failures and 175 link recoveries were simulated. After each simulation the observed complexity of that single-failure event is recorded. The average observed complexity over the \(40 \times 350\) events of the same network size is then computed.

Figure 3-4 first illustrates the average observed complexity of conventional static SPT algorithms as the network size increases. These results are obtained from Chapter 2 where they are analyzed in detail. It is worth noticing the really large number of comparisons needed as the network increases its size. It is clear that using linear
search for the priority queue in Dijkstra’s algorithm quickly becomes inefficient. It is also interesting to note that the algorithm with the worst worst-case bound (D’Esopo-Pape) actually performs the best on average. This is because of some clever heuristic that it uses (revisited nodes are given higher priority - see Chapter 2). However, this heuristic will prove to be counter-productive when using the dynamic version of these algorithms.

When we perform the same experiment using the various dynamic algorithms discussed, we notice a tremendous reduction in the algorithmic complexity. Both dynamic Dijkstra and our new algorithm use binary heaps for their priority queues. Figure 3-5 compares the average observed complexities, which as expected, are orders of magnitudes lower than for static algorithms.

The dynamic version of the D’Esopo-Pape algorithm now becomes quite inefficient with large networks. Dynamic Dijkstra performs slightly better than Dynamic Bellman-Ford because of its improved node selection method. Our new algorithms clearly performs much better on average than the other three dynamic algorithms.

We now give an example of what happens inside each algorithm when some important link failure occurs. Table 3.1 shows the events that took place during the
Figure 3-5: Complexity of dynamic algorithms.

execution of each algorithm after a typical such failure occurs.

The first column contains the number of edges visited by each algorithm. Recall that each time an edge is visited a comparison is made between the nodes at each end of the edge. Dynamic Bellman-Ford and D'Esopo-Pape end up visiting the same edges more than once. Dynamic D'Esopo-Pape seems to be particularly inefficient at this. On the other hand, both Dynamic Dijkstra and our new algorithm visit only the minimum number of edges necessary, and each edge is visited only once. This number (2627) is equivalent to what we previously defined as $\delta_e$, the number of edges departing from the nodes of the non-root branches (nodes affected by the change).

The second column shows the number of extractions or iterations performed by each algorithm. This is the number of times that a new node has to be chosen from the list $Q$. The number of extractions is by far lower with the new algorithm. In fact, the new algorithm performs the minimum number of extractions. This number (13) is equivalent to $\delta_c$, the number of non-root branches (clusters of nodes that need to be reordered).

The third column shows the number of comparisons performed by the search functions in each algorithm. Since Dynamic Bellman-Ford and D'Esopo-Pape have
Table 3.1: Observed algorithm behavior after a link failure.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Edges Visited</th>
<th>Extractions</th>
<th>Search Comparisons</th>
<th>Parent Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Bellman-Ford</td>
<td>5582</td>
<td>646</td>
<td>0</td>
<td>347</td>
</tr>
<tr>
<td>Dynamic D’Esopo-Pape</td>
<td>30676</td>
<td>4880</td>
<td>0</td>
<td>2375</td>
</tr>
<tr>
<td>Dynamic Dijkstra</td>
<td>2627</td>
<td>186</td>
<td>1876</td>
<td>19</td>
</tr>
<tr>
<td>New Algorithm</td>
<td>2627</td>
<td>13</td>
<td>602</td>
<td>13</td>
</tr>
</tbody>
</table>

no priority queues, they don’t have to perform any such comparisons. However, the extra cost suffered by Dynamic Dijkstra and our new algorithm is compensated by the smaller number of edges needed to be visited. Since our new algorithm needs to do less extractions and searches than Dynamic Dijkstra, less search comparisons are performed.

Finally, the last column shows the number of times that the parent attributes are changed during the execution of the algorithms. Dynamic Bellman-Ford and D’Esopo-Pape change the parent attributes of a large number of nodes more than once. Dynamic Dijkstra only changes once the parent attribute of any given node. However, it changes more parent attributes than necessary. The resulting SPT has 6 nodes unnecessarily changing parent. On the other hand, our new algorithm only makes the minimum number of changes in the SPT structure. There is only one edge change for each of the 13 non-root branches.

### 3.7 Chapter Summary

In this chapter, we have presented a new algorithm to recompute the shortest path tree (SPT) in a network after some edges have changed weights. The algorithm is heavily based on the dual linear programming technique and on its physical interpretation. This physical interpretation uses balls and strings to model the system of constraints involved in computing an SPT. Our new algorithm can be seen as a simulation of
how such a physical model would naturally reconfigure itself.

The new algorithm uses a new search criterion (that of minimum distance change) to determine in which direction the information should be propagated. Unlike existing dynamic SPT algorithms which select one node at a time, the new algorithm can select entire branches at a time. This allows the algorithm to use the intact structure of the SPT to spread information in the right direction without having to perform as many searches. The new algorithm also makes the minimum number of changes to the SPT structure.

The algorithm visits each affected node only once. It makes fewer extractions from the priority queue (one per branch) than any other algorithm of comparable complexity (one per node). As a result, when using underlying linear search or bucket search, the new algorithm has a lower asymptotic complexity than any other known algorithm. When using a more complicated ordered structure (such as a heap), the asymptotic complexity is the same as for the best known algorithms. Nevertheless, simulations indicate that on average, the new algorithm has the lowest observed complexity.
Chapter 4

Local Restoration Algorithm for Link-State Routing Protocols

Link-state protocols such as OSPF are the dominant routing technology in today’s Internet. Despite their many advantages, these protocols require the flooding of new information across the entire routing area after changes in any link state (e.g. link failures). As the routing area grows or the frequency of link-state changes increases, the overhead (in terms of bandwidth and processing cost) of flooding becomes prohibitive. Furthermore, such flooding over a large area will cause temporary inconsistency of link states among many routers, potentially creating many transient routing loops that can last for a long time. This limits the scalability of the routing protocols to large routing areas.

To overcome such problems, we present in this chapter a novel algorithm that minimizes the amount of information distributed by link-state routing protocols. Upon a link failure, our algorithm will distribute the link-state changes to the minimum number of routers that are needed to ensure loop-free routing. Moreover, implementing our algorithm requires only a simple extension to any existing link state protocol.
4.1 Background and Motivation

Link-state routing protocols, such as OSPF[13, 20, 23, 32] or IS-IS[14, 26], are becoming the dominant Internet technology for routing packets within Autonomous systems. Under such protocols, an IP router will make its forwarding decision based on the entire description of the topology of the routing area.

In link-state protocols the process of collecting topological information from the network is separated from the actual process of computing the correct routes. The former is performed distributively by all the routers in an area who share information with each other on the state of the links. Once a given router has the topological information on the routing area, it will construct a routing table based on the shortest path tree (SPT) that it computes for that area.

The main advantage of these link-state protocols is that the routing computation is done exclusively inside the router. This means that the computation can be performed quickly and without relying on what other routers are doing. When there is a change in the cost metric of some link, the new information needs to be sent only once to every router in the area; the routing tables will be updated to their correct values immediately. This is in sharp contrast to distance-vector protocols, such as RIP, where multiple routing packets may have to be sent many times between the same routers in order for their routing tables to converge to a steady-state and correct value.

However, even though the amount of information that link-state protocols need to transmit is relatively small (compared to distance-vector protocols), it might still be too large when the cost metrics of the links vary too quickly or when the number of links in an area is too large. In principle, every router in an area should have at all times the latest information on the topology of the area. This topology information should be identical in all routers in order to avoid routing loops. This means that every time there is a cost metric change in one of the links, all the routers in the routing area must be notified. Flooding an entire area after a single link-state change is often unnecessary and is inefficient in terms of bandwidth and computation overhead.
required. Furthermore, flooding a large routing area might require a large amount of
time. During the time that the new information is being propagated, different routers
will have different link-state information, and therefore, transient routing loops are
possible. Many of the Internet's problems with routing instability are associated with
the long delays required to propagate routing information[16, 25, 27].

One way to reduce the total amount of routing information being transmitted
is to have the router responsible for a given link to simply ignore changes in that
link's metric. The new information will simply not be propagated and routing will
continue along the old paths in a sub-optimal manner. After several metric changes,
the router can broadcast an update packet summarizing all the topological changes
that have taken place. By limiting the frequency of such updates, the amount of
network information traffic can be limited at the cost of some sub-optimal routing
between update intervals.

However, when the link change that takes place is that of a link failure (cost metric
becomes infinity), it is no longer possible to withhold this information from the rest
of the network. Doing so would cause routing failure and packet loss. Therefore,
information on the failure of a link must be propagated to some routers in the network
in order to restore the paths that were traversing that link.

In this chapter we will present a routing algorithm to limit the routing information
that needs to be propagated in a link-state protocol when any network link fails. The
scheme will restore all the paths traversing the failed link by only performing local
updates on the neighboring routers. The object of our scheme is to restore loop-free
routing after a link failure, while spreading information about that failure to as few
routers as possible. This is particularly important for very large routing areas where
it is not desirable to disturb a huge number of routers when just a single link fails.

This approach is also useful to divert traffic when a link becomes particularly
congested. In current link state protocols such as OSPF, it is not practical to change
the metric of a link every time its congestion state changes. Because of the large
amount of time required to flood a routing area with new information, changes in
link metrics only take place in OSPF when absolutely necessary. Since our approach
does not require flooding, a link that is congested can temporarily advertise itself as being down and thus can temporarily divert traffic through a local alternative path. Because in our scheme very few routers need to be informed, this operation can be done quickly and as frequently as needed.

Section 4.2 will first state some of the lower bounds associated with this problem. Section 4.3 then gives some intuition on the problem and introduces some preliminary ideas on how to solve it. Sections 4.4 and 4.5 will introduce the two main algorithms that we propose their properties are proven.

4.2 Necessary Conditions

The object of this chapter is to provide a scheme that will restore loop-free routing in a link-state hop-by-hop routing protocol environment with minimum communication overhead. Currently, in regular link-state protocols such as OSPF, this is accomplished by broadcasting the information about that link failure to all the routers sharing the same area. We would like our scheme to achieve the same objective (correct routing) while saving some bandwidth by informing only some of the routers in the area. Informing less routers could lead to a degradation in routing optimality but should not affect the correctness of the routing scheme. Before describing how such a scheme can work, we will show the limits of how much communication complexity we can save by not informing all the routers about the link failure.

We will prove the necessity conditions for the routing recovery to function correctly. This will be in terms of the minimum number of routers that need to be informed of the link failure. When we present the algorithms in Section 4.4 and 4.5, we will also see that this necessity condition will match the sufficiency condition. We begin by defining the different types of restoration paths.

Definition 5 For a given failure of some link \((A, B)\), a restoration path is defined as any set of nodes that includes \(A\) (but not necessarily \(B\)) and forms a single path between \(A\) and \(B\). A shortest restoration path is a restoration path which minimizes
the distance between $A$ and $B$ along that path. A *minimal* restoration path is a restoration path with the minimum number of elements (nodes or routers).

Now we can state the lower bound on the number of routers that need to be informed after a link failure.

**Theorem 8** Given any network topology and any failure of some link $(A, B)$ (the network is still connected after the failure), if the number of routers that are informed of the failure is less than the size $M$ of a *minimal* restoration path, then there is always a set of metrics for the links of the network that will cause *any* routing scheme to fail.

This means that no matter what scheme we use for path restoration, we need to inform at least a number $M$ of routers that is equivalent to the size of a minimal restoration path. If we inform less routers than $M$, then no matter how careful we are in designing the area topology and no matter which link fails, there is always some set of link metrics that will cause some packets to enter routing loops or get dropped. Therefore, any algorithm that wants to restore loop-free routing for all sets of link metrics must inform at least $M$ routers of the link failure. It follows that any algorithm that wants to work for all network also has to comply with the requirement of informing at least $M$ routers. We will now prove Theorem 8.

**Proof:** We assume that we have an arbitrary topology, and an arbitrary link $(A, B)$ in that topology fails. We let $L$ be the total number of nodes in the network. We now construct a set of link metric where every incoming link into $B$ except $(A, B)$ has a cost metric of $L$. Every other link has a cost metric of 1. We let $M$ be the size of a minimal restoration path (all minimal restoration paths have the same size). We assume on the contrary that the number of routers that are informed of the link failure is less than $M$.

We observe the path of a packet $P$ that starts at router $A$ with destination $B$. We define as $P_h$ the router where packet $P$ is located after traveling for $h$ hops. We define $D(n_1, n_2)$ to be the minimum hop count from node $n_1$ to node $n_2$. Because of the high
costs of the links going into $B$, any router in the network with connectivity to $A$ that is uninformed of the link failure will calculate a shortest path from itself to $B$ that must pass through $A$. Therefore, such uninformed routers will forward the packet to a next hop which is one hop closer to $A$. In our terminology, if $P_h$ is uninformed, then $D(A, P_{h+1}) = D(A, P_h) - 1$. On the other hand if $P_h$ is an informed router, the packet can be forwarded at most one hop further away from $A$, $D(A, P_{h+1}) \leq D(A, P_h) + 1$.

Now we assume on the contrary that the routing algorithm is capable of moving packet $P$ from $A$ to $B$ in $k$ hops while traversing $i$ informed routers. It follows that the packet must traverse $k - i$ uninformed routers. After $k$ hops, we know that

$$D(A, P_k) \leq D(A, P_0) + i - (k - i) \quad (4.1)$$

$$\leq 0 + 2i - k \quad (4.2)$$

Since by assumption the number of informed routers is less than $M$, $i < M$. Since the number of traversed hops has to be at least the minimum hop count between $A$ and $B$, $k \geq M$. Therefore, $D(A, P_k) < M$. This contradicts our assumption that the packet reaches $B$ after $k$ hops which would imply that $D(A, P_k) = M$. \qed

In the scenario constructed by the proof, the only way in which a packet starting at $A$ might arrive at $B$ is if there are at least $M$ informed routers. Later, we will also see how when using the appropriate algorithm, in any network, it is sufficient to inform only $M$ routers to achieve loop-free routing.

### 4.3 Preliminary Ideas

In this section, we will present some basic ideas about how routing can be restored with only local updates. While these ideas are either inefficient or do not work in all cases, they provide insight to understanding our local route restoration scheme.
4.3.1 Tunneling

The simplest way to restore packet routing in a network after a link failure is to use tunneling. As illustrated in Figure 4-1, when the link between nodes A and B goes down, a new path is constructed between nodes A and B through nodes N₁, N₂, and N₃. All the traffic that should have traveled through the broken link is now encapsulated and tunneled into this new path, which acts as a virtual link. As with this example, if the chosen path is the minimal restoration path, such a tunneling scheme will inform the minimum number of routers.

![Figure 4-1: Tunneling packets from A to B.](image)

This scheme effectively limits the update information that needs to be broadcast after a link failure. Only the nodes that are part of the new path from A to B are informed of the changes in the topology. Even though the rest of the network does not know about the change, global routing continues to function correctly, although it can now be suboptimal.

The major drawback of a tunneling scheme is that every single data packet that goes through the new path has to be encapsulated at node A. This requires the router
to be able to generate a new packet for every data packet that must be diverted. This operation can greatly limit the efficiency of high-speed routers in their packet forwarding functions.

It would be much more desirable to come up with a mechanism that could leave intact the packet forwarding mechanism of the router. We will now show how our objectives can be met by simply modifying the routing tables of a small number of nodes without resorting to encapsulation or tampering with the per-packet forwarding mechanisms.

4.3.2 Local Routing Table Adjustment

Loop-free routing can still be achieved without tunneling and without global updates. In fact, loop-free routing can happen by only informing and adjusting the routing tables of the same routers as in the case of tunneling (the nodes in the restoration path). The advantage of adjusting the routing tables over tunneling is that the informed nodes need only modify their routing table once and let the forwarding engine function normally, while in the case of tunneling, these nodes need to perform a separate function on a per-packet basis.

Simple Updates

At first sight, it might seem that our objectives can be met by simply recomputing the routing tables in the restoration path. In other words, along some restoration path, the new topological information is broadcast and the routing tables are recomputed as usual, whereas anywhere else, the information is not sent, and these other nodes continue using the old routing tables. In our example, if routers $A$, $N_1$, $N_2$, and $N_3$ simply recomputed their routing tables using the new information about the link failure, global routing would still function correctly. Even though node $S$ still uses an old routing table, a packet from $S$ to $D$ would still be routed correctly, although possibly sub-optimally.

However, it turns out that this partial update scheme is not restrictive enough
in the way packets are forwarded. Because the actual path that a packet takes is
determined on a hop-by-hop basis by nodes with different topology information, it is
possible for routing loops to occur.

For example, in Figure 4-2 we can see how a routing loop can be created. The
number next to each link is the cost metric of that link. When the link between
nodes A and B breaks, a restoration path is established between nodes A, and B,
via node C. According to this scheme, the routing tables of A and C are updated
and all the others are left intact. When a packet with destination D arrives at C,
it is forwarded to E since it is along the shortest path. However, when the packet
arrives at E, node E does not know about the link failure, and the packet is forwarded
back to C which is along the shortest path which existed before the link failure took
place. Because node E is not informed, it continues to route packets according to the
outdated routing table. Therefore, we have a routing loop between routers C and E.

![Figure 4-2: Topology subject to routing loops.](image)

The problem with this erroneous scheme is that packets can leave the restoration
path (in the example, A-C-B) too soon. When this happens, the packet will enter
a region where nodes no longer have updated routing tables, and these nodes can
forward the packet back to an earlier part of the restoration path causing routing
looms.

**Uniform Updates**

One attempt at trying to solve this problem is to force all the packet that would have had to traverse the failed link \((A, B)\) to travel through the entire restoration path. This can be achieved by modifying the routing tables in the nodes belonging to the restoration path in such a way that all packets that would have had to traverse \((A, B)\), now have to be forwarded to the new next hop for \(B\). In other words, all the packets that router \(A\) would have forwarded to \(B\) before the link failure will now all be forwarded along the restoration path until they reach \(B\). However, it turns out that this scheme does not work either. Unlike its predecessor, this scheme is too restrictive in the way next hops are selected, and this might lead to routing loops because the packet might not be able to exit the restoration path on time.

For example, if we go back to Figure 4-1, let us assume that before the link failure, a packet traveling from node \(S\) to \(E\) would traverse nodes \(A\), \(B\), and \(N_3\). After the link failure, the packet is forced to traverse nodes \(N_1\), \(N_2\), and \(N_3\) until it reaches \(B\). At that point, \(B\) will forward the packet back to \(N_3\), leading to a routing loop.

### 4.4 Branch-Update Algorithm

The problem with the two routing table adjustment schemes proposed in the last section is that a packet might leave the restoration path at the wrong time. In one scheme, the packet can leave the restoration path too soon, and in the other, it might leave too late.

In this section, we will propose an algorithm, which we will refer to as the *Branch-Update algorithm*, that finally solves our local routing restoration problem. The algorithm still uses the idea of a restoration path and limits the update information to the nodes in that path. However, unlike the schemes presented in the last section, this algorithm is capable of forcing the packet to leave the restoration path at the right time so that no routing loops can ever occur.
4.4.1 Description

Similarly to the uniform update scheme, the Branch-Update algorithm modifies only some of the entries of the routing table. Similarly to the simple update scheme, all computation is performed locally.

The algorithms functions as follows. After a link failure \((A, B)\), router \(A\) informs all the routers that are in some shortest restoration path of the link failure. Each informed router will then record all the nodes \(D\) that are descendents of \(B\) in its existing (outdated) SPT data structure. Essentially, this branch \(D\) of nodes is the set of destinations that are affected by the link failure. A new shortest path is then calculated only for the node \(B\) following the link failure. The next-hop for all the nodes in the branch \(D\) becomes the same as the newly calculated next-hop for \(B\). A more detailed description of this algorithm is presented in the following pseudocode.

**Branch-Update Algorithm**

1. Link \(L\) between nodes \(A\) and \(B\) fails.
2. In Node \(A\):
   a. The set \(D_0\) is defined as all the nodes that are descendents of \(L\) in any current shortest path tree rooted at \(N_0\).
   b. The link-state database is modified to incorporate the change of state of link \(L\) (the link is down).
   c. The SPF engine recomputes the next-hop for node \(B\) only. The new next hop for \(B\) is now some other node \(N_1\).
   d. The next hop for all the destination nodes in \(D_0\) is set to \(N_1\).
   e. A special packet containing \(L\) and \(B\) is sent to \(N_1\).
3. In Node \(N_i\) after receiving the special packet:
   a. The set \(D_i\) is defined as all the nodes that are descendent of \(L\) in any current shortest path tree rooted at \(N_i\).
   b. The link-state database is modified to incorporate the change of state of link \(L\) (the link is down).
   c. The SPF engine recomputes the next-hop for node \(B\) only. The new
next hop for \( B \) is now some other node \( N_{i+1} \).

d. The next hop for all the destination nodes in \( D_i \) is set to \( N_{i+1} \).

e. If \( N_{i+1} \) is not equal to \( B \), send a special packet to \( N_{i+1} \), containing \( L \) and \( B \). Step 3 will be repeated when the packet reaches \( N_{i+1} \).

The intuition behind this algorithm is that for each router only the branch of nodes in the SPT that is affected by the link failure needs to be considered. Routing to all the other nodes can proceed as before.

This solves the problem seen with the uniform update scheme where packets were leaving the restoration path too late. Furthermore, by assigning the same next-hop to all the considered nodes, it forces all the packets destined to such nodes to follow the same next-hop. This prevents the packet from leaving the restoration path too soon as in the case of the simple update scheme.

Note that the restoration path used by the algorithm is a shortest restoration path. The algorithm would also function correctly if the set of informed routers is any restoration path.

### 4.4.2 Correctness

We will now state the results formally and prove the correctness of the algorithm.

**Theorem 9** When a network \( \mathcal{N} \) (that uses a link-state routing protocol with shortest path routing) suffers from a single link failure, the Branch-Update algorithm can guarantee that loop-free routing will continue in \( \mathcal{N} \) after the link failure.

Before we prove the main result we introduce the following definitions and lemmas:

**Definition 6** Let \( P_d \) be a packet which has router \( d \) as its destination. Let \( R(P_d, h) \) be the router where packet \( P_d \) is located \( h \) hops after being first transmitted. After a given link \( L \) failure, packet \( P_d \) is said to be affected at hop \( h \) if and only if node \( d \) is a descendent of link \( L \) in some SPT that was rooted at \( R(P_d, h) \) before the link failure.
The notion of a packet being affected reflects whether or not a packet has to worry about the effects of the link failure. If a packet is affected at a given hop $h$, something should be done to correct its routing, whereas if a packet is unaffected, routing can proceed as before. We now make the following statements about how the affected state of a packet evolves with time.

**Lemma 16** After a link $L$ failure and a recovery based on our algorithm, if a packet $P_d$ at hop $h$ is not affected, then the same packet will not be affected at future hops $h'$, where $h' > h$.

**Proof:** Let $P_d$ be a packet that is not affected at hop $h$. If $R(P_d, h)$ is not an informed router, then $P_d$ will be forwarded in the same manner as before the link failure. If $R(P_d, h)$ is an informed router, then recall that our algorithm will only change the routing table entries for the destinations that are descendents of $L$ in any SPT computed at $R(P_d, h)$. Since $P_d$ is not affected at hop $h$, destination $d$ cannot be one of the routing tables entries that was modified. Therefore, $P_d$ will also be forwarded in the same manner as before the link failure. In either case $P_d$ is forwarded along the SPT that existed before the link failure. By the definition of affected packet, there is no shortest path between $R(P_d, h)$ and $d$ that traverses link $L$. Therefore $P_d$ will also not affected at hop $h + 1$. By repeating this argument, we conclude that $P_d$ will not be affected at any hop $h'$, where $h' > h$. $\square$

We now know that all packets that are not affected at a given hop count will remain so and will safely reach their destinations using the same paths as before the link failure. We now look at what happens when a packet is affected at a given hop count.

**Lemma 17** After a link $(A, B)$ failure and a recovery based on our algorithm, if a packet $P_d$ is affected at hop $h$, then at some hop count $h' \geq h$, $P_d$ will either (i) reach the destination at $d$, (ii) not be affected, or (iii) $R(P_d, h')$ will be an informed router in the restoration path. The path taken between hops $h$ and $h'$ is loop-free.
Proof: If \( R(P, h) \) is an informed router, then the lemma is true with \( h = h' \).
If \( R(P, h) \) is not an informed router, then packet \( P \) will be forwarded towards \( d \) following some path in the old SPT. The shortest distance between \( R(P, h+1) \) and \( d \) according to the old topology (without the link failure) will therefore be smaller than the shortest distance between \( R(P, h) \) and \( d \). Therefore, while \( P \) remains affected and is inside uninformed routers, \( P \) will be traveling in a loop-free manner towards \( d \). At some point in time, one of the three cases outlined above will have to take place.

Now the only case that is left to be covered is what happens to \( P \) when it is both affected and inside an informed router.

Lemma 18 After a link \((A, B)\) failure and a recovery based on our algorithm, if a packet \( P \) is affected at hop \( h \) and \( R(P, h) \) is an informed router, then at some future hop count \( h' \geq h \), \( P \) will not be affected. The path traversed between hops \( h \) and \( h' \) is loop-free.

Proof: According to our algorithm, all of the routing table entries for destinations that were descendents of \((A, B)\) in some old SPT are modified in the same way. Therefore, destination \( d \) is one of the entries that is modified in routing table of router \( R(P, h) \). According to the algorithm, \( P \) will be forwarded to the next hop of the restoration path (we do not require the restoration path to be a shortest restoration path). If at the next hop \( P \) is not affected, the statement is satisfied. If at the next hop \( P \) is still affected, \( P \) will continue to be forwarded along the restoration path until it becomes unaffected and/or it reaches router \( B \). Since the old SPT computed at \( B \) could not have contained any loops, at router \( B \), \( P \) must be unaffected.

Finally, we can paste all the different trajectories that a packet might take and complete the proof of correctness.

Proof of Theorem 9: According to Lemma 16, if a packet starts traveling unaffected, it will continue to be unaffected until it reaches its destination in a loopless manner. If it doesn't start unaffected, according to Lemma 17, it will either reach
its destination in a loop-less manner, become unaffected while traveling loop-less (in which case the destination will be reached in a loop-less manner), or will arrive at an informed router in a loop-less manner. If it reaches an informed router, according to Lemma 18, the packet will become unaffected and therefore reach its destination in a loop-less manner. Therefore, for any combination of circumstances (topology, metrics, link failure, starting router, destination router), a packet will be able to reach its destination without traveling in any loops.

\[ \Box \]

4.4.3 Optimality

As discussed in the previous section, the routers that are informed by our algorithm about the the link failure can be any restoration path. The specific restoration path chosen by the pseudo-code in Section 4.4.1 is a shortest restoration path. This is attractive from an implementation point of view because this restoration path can be computed using the same SPF engine used to compute the routing table. A different restoration path can be chosen by varying the next-hop search function. For example, a minimal restoration path can be computed if the informed routers use a minimum-hop tree rather than a shortest-path tree to compute the next hop of the restoration path.

If the restoration path chosen by our algorithm is a minimal restoration path, then our algorithm satisfies the the lower bound stated in Section 4.2. Our algorithm will restore correct loop-free routing while informing the minimum number \( M \) of routers.

4.4.4 Limitation

The major limitation of our algorithm is that it can only guarantee loop-less free routing when there is only a single failure in the network. While it is quite possible that our algorithm might function correctly with multiple link failures, under extreme circumstances, routing loops might still occur.

Figure 4-3 gives an example of when our algorithm will create routing loops. First both links \((X, A)\) and \((D, E)\) fail. For the first failure a restoration path consisting
of $X$ and $B$ is formed. For the second failure, the restoration path consists of $D$ and $C$. Now, a packet starts at $X$ and has destination $Y$. Since $(X, A)$ is broken, $X$ will forward $P_Y$ to $B$. At $B$, $P_Y$ is no longer affected by the $(X, A)$ failure. Therefore, the packet will be forwarded towards what $B$ thinks is the shortest path to $Y$ ($B - D - E - Y$). When the packet reaches $D$, it is forwarded along the restoration path to $C$. At $C$ the packet is no longer affected by the $(D, E)$ failure and the packet is forwarded along the direction of what $C$ thinks is the shortest path ($C - X - A - Y$). The packet then returns to $X$ and initiates an infinite routing loop $X - B - D - C$.

![Figure 4-3: Topology subject to routing loops when there are 2 link failures.](image)

As can be seen in this example, the problem with two link failures is that a packet that is unaffected by a given link failure can become affected because of the interference of the other link failure.

### 4.5 Vector-Metric Algorithm

The problem with the two routing table adjustment schemes proposed in the last section is that a packet might leave the restoration path at the wrong time. In one scheme, the packet can leave the restoration path too soon, and in the other, it might
leave too late.

In this section, we will propose an algorithm, which we will refer to as the Vector-Metric algorithm, that finally solves our local routing restoration problem. The algorithms still uses the idea of a restoration path and limits the update information to the nodes in that path. However, unlike the schemes presented in the last section, this algorithm is capable of forcing the packet to leave the restoration path at the right time so that no routing loops can ever occur.

4.5.1 Description

The main novelty of the Vector-Metric algorithm is that the metric of each link is represented inside each router by a vector of values rather than by a scalar. In the link-state database of each router, each link has an associated vector $V = (v_0, v_1, v_2, \ldots, v_n)$ representing the cost metric of that link. The first element of the vector is referred to as the zero-order metric, the second element is the first order metric, and so on. Lexicographic ordering is used to compare the values of two vectors. In other words, an $i^{th}$ order metric is infinitely greater than an $i + 1$ order metric. During initialization, the zero-order metric is set to the original cost of the link, and every other metric is set to zero. During the execution of the algorithm, the vector metric of a link can be downgraded. When this happens, the zero-order metric for that link is set to zero, and all the other order metrics are shifted by one ($v_i \leftarrow v_{i-1}$). More formally, if $V = (v_0, v_1, v_2, \ldots, v_n)$ is the vector metric of a link, after it is downgraded, the new vector metric will be $V' = (v'_0, v'_1, v'_2, \ldots, v'_n)$, where $v'_0 = 0$ and $v'_i = v_{i-1}$.

The algorithms functions as follows. After a link failure $(A, B)$, router $A$ informs all the routers that are in some shortest restoration path about the link failure. The set of links $\mathcal{P}$ that connect the nodes in the restoration path are recorded. For each of the links in $\mathcal{P}$, the informed routers will downgrade the metric vector of that link. Note that these vector metrics are only downgraded in the link-state database of the informed routers (those in the restoration path). All the other routers are not aware of the link state change. Once these vector metrics are downgraded, the routing tables
are recomputed in the informed routers using the regular shortest path first (SPF) engine. The only novelty is that the SPF engine uses the vector metrics rather than just the scalar metrics. Vector metrics are added like any two vectors; comparisons are done using lexicographic ordering. A more detailed description of this algorithm is presented in the next page.

**Vector-Metric Algorithm**

1. Link $L$ between nodes $A$ and $B$ breaks.
2. In Node $A$:
   a. The link state database is modified to incorporate the change of state of link $L$ (the link is down).
   b. The SPF engine computes the entire path to reach node $B$. The set of links in this path is $\mathcal{P}$.
   c. The vector metrics of all the links in $\mathcal{P}$ are downgraded.
   d. The SPF engine recalculates all the next-hops using the vector metrics as modified in C. These next-hops are used in $A$’s new routing table.
   e. A special packet containing $L$ and $\mathcal{P}$ is sent along path $\mathcal{P}$.
3. In Node $N_i$ after receiving the special packet:
   a. If $N_i = B$, stop
   b. Otherwise, the link state database is modified to incorporate the change of state of link $L$ (the link is down).
   c. The vector metrics of the links in $\mathcal{P}$ are downgraded.
   d. The SPF recalculates all the next-hops using the vector metrics as modified in $B$. These next-hops are used in $N_i$’s new routing table.
   e. A special packet containing $L$ and $\mathcal{P}$ is sent along path $\mathcal{P}$.

The intuition behind this algorithm is that the informed routers should consider any distance within the restoration path to be infinitely smaller than any distance outside the restoration path. In this way the entire restoration path acts like a macro-node. Any packet that enters the restoration path will exit it using the most optimal
link, just like before the link failure, a packet entering a regular router would have also exited it using the most optimal link. Whenever there is a link failure, the algorithm will simply group all the routers in the restoration path to behave like a macro-node. This solves the problem seen with the two schemes presented in the last section. Each packet will exit the restoration path at the optimal point: at the router which is closest to destination.

Note that the restoration path used by the pseudocode is a shortest restoration path. The algorithm would also function correctly if the set of informed routers is any restoration path. The shortest restoration path will provide the most optimal routing, but any other restoration path will also provide loop-less routing.

4.5.2 Correctness

We will now state the results formally and prove the correctness of the algorithm.

**Theorem 10** When a network \(\mathcal{N}\) (that uses a link-state routing protocol with shortest path routing) suffers from any link failure, the Vector-Metric algorithm can guarantee that loop-free routing will continue in \(\mathcal{N}\) after the link failure.

Before we prove the main result we introduce the following definitions and lemmas:

**Definition 7** Let \(P_d\) be a packet which has router \(d\) as its destination. Let \(R(P_d, h)\) be the router where packet \(P_d\) is located \(h\) hops after being first transmitted. After a given link \(L\) failure, \(D(P_d, h)\) is the shortest vector distance between router \(R(P_d, h)\) and \(d\) using the new vector metrics stored in \(R(P_d, h)\). Furthermore, let \(D'(P_d, h)\) be the the same shortest distance, but using the vector metric that existed in \(R(P_d, h)\) before the link failure.

The distance \(D(P_d, h)\) is essentially the shortest distance to destination \(d\) for packet \(P_d\) seen locally by the router in which \(P_d\) is located at hop count \(h\). We now prove the correctness of the algorithm by showing that for any given packet \(P_d\), its local distance to destination \(D(P_d, h)\) monotonically decreases after every hop.
Proof: We first look at two possible cases:

Case 1 - If \( R(P_d, h) \) is not an informed router, then \( R(P_d, h) \) has not downgraded the vector metrics for the restoration path links. Because the router is using shortest path routing, it will forward packet \( P_d \) to a route which is closer (using the non-downgraded vector metrics) to destination \( d \), \( D'(P_d, h+1) < D(P_d, h) \). If this next router \( R(P_d, h+1) \) happens to be informed, then the shortest distance to destination using the downgraded vector metrics might be even lower (but never greater). Whether the next hop router is informed or not, \( D(P_d, h+1) \leq D'(P_d, h+1) \). Either way, \( D(P_d, h+1) < D(P_d, h) \).

Case 2 - If \( R(P_d, h) \) is an informed router, then it has downgraded the vector metrics for the restoration path links. We let \( m \) be the informed router with the smallest shortest distance to destination \( d \) (using the downgraded vector metrics). If router \( R(P_d, h) \) is \( m \), then the shortest path between \( R(P_d, h) \) and \( d \) does not use any downgraded vector metrics. Therefore \( D(P_d, h) = D'(P_d, h) \), and \( R(P_d, h) \) will forward \( P_d \) in the same manner as if the link \( L \) failure had not occurred. The packet will be forwarded to a non-informed router and \( D(P_d, h+1) < D'(P_d, h) = D(P_d, h) \).

On the other hand, if \( R(P_d, h) \) is not \( m \), then the shortest path between \( R(P_d, h) \) and \( d \) (using the downgraded vector metrics) must pass through a set of informed routers until it reaches \( m \). This is because the distance of any path consisting only of links with downgraded vector metrics is infinitely less than any path containing some link whose vector metric has not been downgraded. Therefore, the packet will be forwarded to another informed router. Since both routers share the same vector metrics and the packet is forwarded along the shortest path, \( D(P_d, h+1) < D(P_d, h) \).

Therefore, whether \( R(P_d, h) \) is an informed router or not, the packet will be forwarded to a router whose shortest distance to destination is always smaller. Therefore, for any \( h^* > h \), \( D(P_d, h^*) < D(P_d, h) \). Since there is a unique shortest distance to \( d \) for every router, \( P_d \) cannot visit any router more than once. Therefore, our vector metric algorithm can guarantee loop-free routing. \( \square \)
4.5.3 Optimality

As in the case of the Branch-Update algorithm, if the restoration path chosen by our Vector-Metric algorithm is a minimal restoration path, then our algorithm satisfies the lower bound stated above. Our algorithm will restore correct loop-free routing while informing the minimum number $M$ of routers.

4.5.4 Extensions

The Vector-Metric algorithm can also inform any extra number of routers that are outside the restoration path. These extra informed routers as well as the original restoration path form a restoration set. After a link failure, all the routers in such a restoration set will downgrade the vector metrics of all the links that connect any two routers in the restoration set. All these routers will then recompute their routing tables with modified metrics. In other words, the macro-node created after a link failure can be as large as we want as long as it contains at least a restoration path. Note that the proof of correctness also applies for the case where there the set of informed routers is larger than the restoration path.

Providing the option of informing a variable number of nodes gives us the ability to control the optimality of the global routing. In the extreme case, when all the routers in a routing area are notified, we simply fall back to the normal OSPF behavior. All the links in the network are downgraded and all the zero-order metrics have become first order metrics. At the cost of having to perform global flooding, we have optimal global routing once again. On the other hand, when only a minimal restoration path of routers is informed, the global routing can be sub-optimal. In all cases, routing is always loop-free, but a larger number of informed routers implies better routing optimality. Furthermore, the size of the restoration set can be increased any time after the link failure. When the set eventually includes all the area routers, we are again routing optimally. All the zero-order metrics have zero value and we can reuse this memory slot to store the metric of a lower order.

On the other hand, when a failed link becomes functional (or uncongested) once
again, the set of informed routers is notified of the recovery. The corresponding vector metrics that were previously downgraded are now upgraded (values are shifted upward, $v_i \leftarrow v_{i+1}$ and $v_n \leftarrow 0$). In these routers, the routing tables are then recomputed as usual.

Another important feature of the Vector-Metric algorithm is that it is capable of guaranteeing loop-less routing even in the presence of multiple link failures. In the cases where the multiple failures are sequential or independent, the execution of the multiple link failure protocol is a simple extension of the case where there is a single link failure. For each link that fails, $L_1, L_2, L_3, \ldots$, a new restoration set is found. As usual, each restoration set $R_1, R_2, R_3, \ldots$, must include at least a restoration path for the corresponding failed link. The information about each link $L_i$ failure needs to be propagated only to the corresponding restoration set $R_i$. Each router with new information will just downgrade the vector metrics of all the links connecting the restoration set of which it is part. If a router is part of more than one restoration set, it will downgrade the vector metric of a given link as many times as that link is part of a different restoration set. For example, if the restoration sets $R_1$ and $R_2$ overlap, then the routers in $(R_1 \cap R_2)$ will downgrade twice all the links that connect the nodes in $(R_1 \cap R_2)$. Once the appropriate links are downgraded, every router with new information recomputes its routing table. If new failures occur in the future, the same procedure is repeated once again. The number of levels in which we can apply this scheme on the same router is only limited by the size of the vector metric.

In cases where multiple failures occur simultaneously and failures are not independent (one failure disrupts the restoration path of another failure), the restoration protocol must be able to adjust the shape of each restoration set to circumvent the new failures. Such readjustments must be done reliably so that every router in a restoration set sees the same set. The implementation of the distributed algorithms needed to ensure this reliability is beyond the scope of this thesis and will be addressed in future works.
4.6 Chapter Summary

In this chapter, we have presented the vector metric algorithm that can restore routing in a link-state network after multiple link failures. The main novelty of the algorithm is that the link state metric used internally by the SPF engine at each router is represented by a vector rather than a scalar. The extra information allows the algorithm to operate only in the local neighborhood of the failed link. Our algorithm needs to inform the minimum number of routers about any given link failure. Later on, if we desire to increase routing optimality, the size of the set of informed routers can increase at any time. Likewise, when a link recovers, only the same set of informed routers needs to be notified.

By limiting the amount of information broadcast after a link failure, our vector metric algorithm effectively reduces the amount of bandwidth and computational overhead required for loop-less network routing to be restored. Furthermore, the recovery time is considerably shortened, reducing to a minimum the transient routing loops that might occur while the new information propagates.

When using this algorithm, temporarily congested links can also quickly divert traffic to alternative paths by pretending to have failed. Since the number of routers that need to be informed is small, such updates can happen more quickly and frequently than conventional link-state protocols would allow. After a short time, when there is no more congestion, the link can pretend to recover, and routing will proceed as before.
Chapter 5

Efficient Algorithms for
Multi-Path Link-State Routing

A primary function in today’s Internet router is to determine the next hop to which a packet should be forwarded. The algorithm used by each router to obtain the next-hop information for a packet should always result in a loop-free routing path to the packet’s destination. An example of such an algorithm is the link-state protocol OSPF. In an OSPF-based router, all packets with the same destination are forwarded to a next hop that lies on a path of shortest distance to the destination. Given a set of link states, all packets between the same source and destination are forwarded along one or more paths of the same (shortest) distance.

This chapter presents a novel Multiple Path Algorithm (MPA) that enables a packet to be forwarded to its destination along multiple alternate loop-free paths, which are not necessarily of shortest distance. The key idea of MPA is to maintain an efficient data structure in a router that helps compute for each destination alternate viable next hops from the router in order to ensure loop-free routing for each packet. Each viable next hop is also specified with a degree of optimality, which enables a router to perform QoS routing and fault-tolerant routing efficiently. Moreover, the data structure used in MPA can be implemented as an add-on software to existing routing protocols, such as OSPF. The MPA is totally interoperable with existing protocols in the sense that loop free routing is still guaranteed in a network in which
only some of the routers use MPA.

5.1 Background and Motivation

In packet switching networks such as the Internet, a packet from a source traverses intermediate network routers before reaching its destination. When a packet arrives at a router, a routing protocol is used to determine the next hop and the corresponding outgoing link (port) to which the packet should be forwarded.

Link-state routing protocols, such as OSPF [13, 20, 23, 32], compute this next-hop information in two phases. First, given the current link states in the network, each router $S$ keeps track of a shortest path tree (SPT) rooted at $S$ to every other node. Since the SPT determines a unique path from $S$ to each router $R_i$, the first hop from $S$ along this path will then be used as the next hop to forward any packet that has $R_i$ as its destination.

Although a router using OSPF has global information about the regional network topology, it determines only the local route of a packet – the next hop. When each router computes such next-hop information based on an SPT, then each packet is guaranteed to be forwarded along a loop-free path to its destination. If multiple SPTs rooted at router $S$ co-exist in a network, there can be multiple choices of next hops from $S$ to forward a packet, each of which will lead to a different shortest path of the same minimum distance. In this case, the packet can be forwarded to any of these next hops, which would also result in a loop-free path.

Likewise, routers that use other types of routing protocols (distance vector protocols such as RIP or path vector protocols such as BGP) also employ next-hop routing. In these cases, the shortest paths are computed distributively so that no single router knows the entire shortest path to a destination. Nevertheless, each router knows the first hop of the shortest path to any destination $D$. This is the next hop used to forward a packet with destination $D$. Therefore, although the information on shortest paths may not be explicit (as in OSPF), practically all routing protocols in the Internet cause packets to travel along a shortest path to their destination. A pack-
et is always forwarded to a next hop that is the first link of the shortest path to destination.

This chapter presents a new Multiple Path Algorithm (MPA) which generalizes this idea by allowing a router to forward packets to multiple viable next hops that are not necessarily part of a shortest path from the router to the packet's destination. The essence of MPA is the implementation of an efficient data structure in the router that helps compute alternate viable next hops for each destination. When each router forwards its packets to any of the viable next hops, all packets will be routed to their destinations on loop-free paths. Moreover, a degree of optimality is determined for each viable next hop in the MPA. Different degrees of optimality yield paths of different distances.

An advantage of maintaining multiple viable next hops in a router for each destination is to speed up recovery from link failure. A router supporting MPA can recover from link failure much faster since the recovery mechanism using alternate paths can be implemented locally by the forwarding engine in the router. Since there is no need to wait for the routing tables to be recalculated, there is almost no delay incurred by the link failure.

On the other hand, without supporting MPA, after detecting its link failure, the router would need to inform other routers about the status of the link failure by flooding the information throughout the network. The router would then recompute the shortest path tree to determine a new alternative path. At the very least, the SPT algorithm in the router needs to recompute a new SPT.

Another reason for supporting MPA is that it allows a router to perform load balancing along different paths. Since the MPA produces extra next-hops, packets for the same destination can be sent to multiple viable next hops. Traffic being forwarded to a congested port can be partially diverted to another port that is still viable but is not as congested. Furthermore, the optimality metric can be used to determine what share of the traffic should go through each different next hop. Nevertheless, some additional schemes may need to be implemented to preserve the FIFO arrival order of the packets if such property is assumed by the upper layer protocol at the
destination. Otherwise, the destination may need a larger buffer and more processing to reassemble the packets delivered by different paths.

Yet another important application of MPA is Quality-of-Service (QoS) routing. It is known that finding an optimal path that satisfies certain QoS constraints is in general computationally difficult (NP-complete). Since some QoS parameters that need to be optimized may depend on the network traffic load and thus can change dynamically over a short period of time, this makes finding an optimal path infeasible even for a moderate-size network. Moreover, even if some QoS parameters do not change frequently, determining the optimal path from a router may require global knowledge of these parameters in the network, and a local change in these parameters would therefore need to be disseminated to every node in the network. Hence in most cases, only suboptimal paths can be computed.

The MPA can be used to determine good approximate solutions to QoS routing by decoupling the underlying optimization problem into local and global computations. First, multiple viable paths that satisfy the global or less frequently changed QoS constraints are computed. Then, among these viable paths, the one that is optimized with respect to the local or more frequently changed QoS parameters can be determined. Although such a heuristic approach does not always yield the optimal path, in many cases a satisfactory path can be found.

Another desirable property of our MPA scheme is that the algorithm and its data structure can be efficiently implemented as an add-on component to existing routing protocols such as OSPF. It only adds a small constant overhead to the shortest path tree computation of the underlying routing protocol. Each MPA-enhanced router computes multiple paths based on the topology information exchanged by the router protocols.

Conventional routers based on shortest path routing can co-exist with MPA-enhanced routers in the same routing area and still guarantee loop-free routing for each packet. As we will discuss in more detail later, such interoperability is a consequence of the loop-free routing policy employed by our MPA scheme. More specifically, if we define the distance between two routers to be the length of the shortest path
from one to the other, then our scheme only forwards a packet to next-hop routers that are closer to the destination, a property which is also shared by conventional shortest path routing, thereby ensuring the resulting path to be loop free. In general, an MPA-enhanced router can interoperate with any router whose routing protocol uses the loop-free routing policy.

5.2 Notations and Definitions

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote a directed graph where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges in the graph. Let $N$ denote the total number of nodes in $\mathcal{V}$ and $E$ the total number of edges in $\mathcal{E}$. We will assume that every node in $\mathcal{G}$ is reachable from every other node by a directed path. Each edge $e_i$ ($i = 0, ..., E - 1$) has an associated weight (cost) of $w_i$, which is assumed to be positive. The weight of an edge connecting node $a$ to node $b$ is denoted by $w(a, b)$.

A rooted tree $T$ is a subgraph of $\mathcal{G}$ such that a particular node is specified as the root and every node in $\mathcal{G}$ is reachable from the root through a unique directed path using only edges in $T$. A node $n_j$ is called the next hop of node $n_i$ along a directed path if $n_i$ is directly connected to $n_j$ via an edge of the path. The length of a path is the sum of the weights of the edges in that path. The distance of a node $n_i$ ($i = 1, ..., N$) in $T$ is the length of the directed path in $T$ connecting the root to $n_i$. The shortest distance $d(a, b)$ from node $a$ to node $b$ is the length of the shortest path from node $a$ to node $b$. A tree $T$ is said to be a shortest path tree (SPT) for $\mathcal{G}$ if the shortest distance from the root node to every node $n_i$ using only the edges in $T$ is also the shortest distance from the root node to $n_i$ using any edge in $G$. Note that there can be multiple SPTs in a graph, but the shortest distance of any node is unique.

A network will be modeled as a directed graph where each router corresponds to a node and each link corresponds to an edge in the graph. Given any two routers ($S$ and $R$) in the network, there are many different loop-less paths that a packet can take to travel between $S$ and $R$. However, conventional routing algorithms always
force packets to travel between nodes $S$ and $R$ only along a shortest path between the two points.

In this chapter, we will relax this constraint and allow packets to follow other loop-less paths that are not necessarily the shortest paths. For this purpose, routers must be able to select next-hops that are not necessarily the first hop of a shortest path, but can be the first hop of some other loop-less path. We use the following definitions to classify the different types of loop-less paths between a source router and $S$ and some destination router $R$:

- The set $\mathcal{P}(S, R)$ includes all loop-less paths between the routers $S$ and $R$.

- The set $\mathcal{D}(S, R) \subseteq \mathcal{P}(S, R)$ consists of all paths from $S$ to $R$ such that the shortest distance from each consecutive hop to the destination $R$ decreases as the path is traversed, i.e., if $R_j$ is the next hop of $R_i$ along the path, then $d(R_j, R) < d(R_i, R)$.

- The set $\mathcal{S}(S, R) \subseteq \mathcal{D}(S, R)$ consists of all paths from $S$ to $R$ such that the shortest distance from each consecutive hop to the destination $R$ decreases and the distance from $S$ to each consecutive hop increases as the path is traversed, i.e., if $R_j$ is the next hop of $R_i$ along the path, then $d(R_j, R) < d(R_i, R)$ and $d(S, R_i) < d(S, R_j)$.

- The set $\mathcal{M}(S, R) \subseteq \mathcal{S}(S, R)$ consists of all shortest paths from $S$ to $R$.

As seen in Figure 5.2, set $\mathcal{P}$ gives the broadest set of paths to go from a source router $S$ to a destination $R$. Each successive subset limits the number of possible paths.

## 5.3 Framework

Recall that a conventional routing algorithm chooses the next-hops for a given destination $R$ by first (i) computing explicitly (in link-state protocols) or implicitly (in distance-vector protocols) the shortest paths to $R$ and then (ii) sets the next-hops...
to be the first segments of these paths. Using our terminology, this is equivalent to finding all the paths in the set $\mathcal{M}$ and choosing the union of their first segments as the set of next-hops the router uses for forwarding packets with the given destination.

We extend this conventional method by making use of a wider set of paths than those defined by $\mathcal{M}$. A routing algorithm first considers some initial set of paths to destination. The algorithm will then use the union of the first segments of the the wider set of paths to choose the next-hops available for forwarding. The wider the set of loop-less paths we initially consider, the more next-hops we will compute.

Figure 5-2 illustrates how this principle can be applied. The figures shows six nodes representing six routers. The source router is $S$ and the destination router is $R$. The dashed lines between the nodes are the links connecting the nodes. The numbers next to them are the distance metrics associated with each link.

To forward a packet with destination $R$, in a common OSPF implementation, router $S$ will calculate a shortest path from $S$ to $R$ which may result in path $A$. Since router $X$ is the first hop along $A$, according to the routing table, $X$ becomes the next
hop for all packets with destination $R$.

In some implementations of OSPF, both paths $A$ and $B$ might be initially selected since they are both shortest paths. In such a case, both the first hop of $A$ and $B$ will be first hops for packets with destination $R$. In this particular example, the first hop for both paths is always $X$.

We can extend the number of paths in the example of Figure 5-2 by including path $C$. Path $C$ is in set $S$ but not in set $M$ ($C$ is not a shortest path). Since node $Y$ is the first hop of path $C$, node $Y$ also becomes a next hop for packets with destination $R$. Therefore, if node $S$ receives a packet with destination $R$, the packet can be forwarded to either node $Y$ or node $X$ since they are both next hops.

Ideally, we would like extend the initial set of considered paths as much as possible in order to obtain as many next hops for a given destination as possible. However, if the initial set of considered paths is too wide, a routing loop might occur at a later hop. For example, if router $S$ (as well as routers $W$ and $Y$) initially consider path $D$, a packet with destination $R$ would travel from $S$ to $Y$ to $W$, and then it would be forwarded to $X$. However, since $X - W - R$ is a shortest path from $X$ to $R$, it
is possible that router \(X\) might send the packet back to \(W\) causing a packet loop. Note that path \(D\) while being inside \(\mathcal{P}\) is outside \(\mathcal{D}\). In the next section we will see in greater detail why choosing an initial path outside of \(\mathcal{D}\) might lead to a routing loop.

The general procedure for obtaining the set of next hops and thus building the routing table can be summarized in the following two steps:

**Forwarding Table Construction**

1. Some unspecified set of loop-less paths from source router \(S\) to destination router \(R\) is computed. The computation can be centralized in \(S\) (if we are using a link-state protocol) or can be done distributively (in the case of distance-vector or path-vector protocols).

2. Router \(S\) determines the first segments of the paths computed in step 1. The union of these first segments become the next hops for destination \(D\). Note that (as in the case of distance-vector protocols) router \(S\) never needs to know the entire paths computed in step 1.

Conventional routing can be seen a special case of this general procedure where the initial set of computed paths is restricted to be in \(\mathcal{M}\). In the next section, we will see what other sets can be used for this purpose.

### 5.4 Viable Next Hops

Recall that the objective of the MPA is not only to determine multiple next hops from a router, but to guarantee that each of those next hops will lead to global loop-free packet routing. A router cannot forward a packet to a next hop that might lead to a routing loop later on. We refer to the next-hops that guarantee loop-free routing as *viable* next hops. More formally:

**Definition 8** Let \(S\) be any router directly connected to another router \(R_i\). Given the routing algorithm used on the other router \((\neq S)\) and a destination router \(D \neq S\),
router $R_i$ is a viable next hop for $S$ if any packet departing $R_i$ with destination $D$ will never return to $S$.

The definition of viable next hop is destination as well as routing algorithm dependent. Different destinations lead to different viable next hops. Likewise, different routing algorithms employed in the other router will force a given router $S$ to have different viable next hops. For example, if the other routers in the network use a very loose routing policy (packets may be forwarded in many different direction), router $S$ might have to be very careful about where it forwards its packets, and only a few next hops can guarantee that the packet will not return back to $S$. On the other hand, if the other routers have a very strict routing policy (packets may only be forwarded in very few directions), there might be more viable next hops at $S$.

To simplify this scenario, we will assume that each router in the network uses only one of two possible forwarding table construction schemes:

1. A router uses a conventional routing scheme. This is the same as our general forwarding table construction with the initial path set restricted to be within $\mathcal{M}$.

2. A router uses the new MPA routing. It also uses the general forwarding table construction scheme, but the initial set of paths is restricted to be within some other set $\mathcal{Q}$.

As we discussed earlier, the basic requirement of any routing protocol is that any packet should be routed to its destination along a loop-free path. The widely used OSPF protocol guarantees loop-free routing to each destination by imposing that each router chooses the next hop based on an SPT. In other words, every router in the network is forced to use routing scheme 1. If the SPT is unique, then corresponding to each destination the next hop from the router is also unique. In addition, the routing of each packet is optimized with respect to the total cost associated with the traversed links.
However, determining the next hop by using scheme 1 is but one way of guaranteeing loop-free routing. If the set $Q$ is chosen correctly, other routers using scheme 2 can co-exist in the same area along with conventional routers using scheme 1.

There might be many types of sets $Q$ that are large enough (producing many next hops) and can guarantee that no packet will return to a router it has previously visited. It turns out that the previously defined set $D$ is one of them. That is, if all the routers using MPA compute initial paths that are within $D$, no packet will return to a previously visited router.

This loop-less behavior can be derived from the progressive distance changes as a packet is forwarded from one router to another. Notice that if router $R_2$ is the next hop of router $R_1$ along a shortest path from $R_1$ to destination router $R_3$, then we must have $d(R_2, R_3) < d(R_1, R_3)$. If we replace $R_2$ by any router $R_i$, such that $d(R_i, R_3) < d(R_1, R_3)$, as the next hop for $R_1$, $R_i$ is still a viable next hop. In other words, it can be shown that loop-free routing can be guaranteed if the shortest distance from a router to its destination is always larger than the shortest distance from the next hop of the router to the destination. More formally, this general routing policy can be stated as follows:

**Loop-free routing policy**  If a packet with destination $R_2$ is forwarded by router $S$ to its next hop $R_1$, then we must have

$$d(R_1, R_2) < d(S, R_2).$$

If a packet is always forwarded using this loop-free routing policy, its distance to destination is monotonically decreasing. Since the distance to destination is unique at each router, this implies that a packet packet will never return to a previously visited router.

If a forwarding table is constructed using scheme 2, where the initial computed paths are restricted to be in $D$, then all the next hops derived from such paths will conform to the loop-free routing policy. It follows that given a destination, any next hop of the router computed in this manner is a viable next hop. As we will discuss
later, the MPA makes use of a data structure to help compute alternate viable next hops whose shortest distances to the destination are less than the shortest distance from the router to the destination.

In practice, many Internet routers compute next-hop information based on shortest path routing. Since \( M \subset D \), conventional shortest path routing (table construction scheme 1) is a subset of the MPA with \( Q = D \) (table construction scheme 2). Therefore, MPA routers and conventional routers based on shortest path routing can interoperate in the same network and guarantee a loop-free route for every packet. For the rest of this work, we assume that the initial set of paths computed by the MPA must be within \( D \).

5.5 Data Structure

A conventional SPT algorithm (e.g., in Dijkstra, Bellman-Ford, etc.) during execution usually keeps track of certain state information about each node in the network[2, 3, 7]. Typically, such information consists of a distance attribute, the next hop attribute from the router, and perhaps a parent attribute for each node. At each instant of the algorithm execution, the distance attribute of a node usually represents the distance of the shortest path found so far from the source router to that node, while the next hop attribute and the parent attribute respectively identify the first hop and the last hop of that path.

Distributed SPT algorithms used in distance-vector and path-vector protocols also make use of the same information, although this information is distributed over the network, and not necessarily found at a single router. For example, with distributed Bellman-Ford, during the computation, a router knows the distance of the shortest path found so far to every destination along with the next-hop for that path. However, the router has no knowledge of the entire path. It only knows its first hop and the length. To discover the path, one would have to probe all the routers along the path.

To compute alternate viable next hops, the MPA uses a data structure that obtains more state information about each node in the network than those described above.
Table 5.1: MPA data structure for each destination node.

<table>
<thead>
<tr>
<th>Shortest distance from router S to node X</th>
<th>Hop A</th>
<th>w(S, A)</th>
<th>d_A(S, X)</th>
<th>Hop B</th>
<th>w(S, B)</th>
<th>d_B(S, X)</th>
<th>Hop C</th>
<th>w(S, C)</th>
<th>d_C(S, X)</th>
<th>Hop D</th>
<th>w(S, D)</th>
<th>d_D(S, X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_opt(S, X)</td>
<td>:</td>
<td>:</td>
<td></td>
<td>:</td>
<td>:</td>
<td></td>
<td>:</td>
<td>:</td>
<td></td>
<td>:</td>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>

without incurring significant computational complexity.

More specifically, the MPA in a given router S maintains for each destination node X the data structure shown in Table 5.1.

For each node X, the first entry in the data structure will contain the shortest distance \(d_{opt}(S, X)\) from the router S to X discovered so far. This information is computed by existing SPT algorithms such as Dijkstra or Bellman-Ford. In addition, the cost \(w(S, p)\) for each outgoing link or port connecting the router S to a potential next hop \(p\) is also maintained. Furthermore, the data structure keeps track of the length of the shortest path found so far that uses \(p\) as the next hop from S, which is denoted by \(d_p(S, X)\). When the execution of the MPA terminates, \(d_p(S, X)\) should be more than or equal to the sum of \(w(S, p)\) and the shortest distance \(d(p, X)\) from \(p\) to X.

By maintaining this data structure, the MPA identifies viable next hops for destination X by making use of the following test:

**Viability Test**  If \(d_p(S, X) < (d_{opt}(S, X) + w(S, p))\), then \(p\) is a viable next hop.

This correctness of this test can be easily derived algebraically from our loop-free routing definition. If some path through some potential port \(p\) has been found with distance \(d_p(S, X)\), it means that the the shortest distance from the next hop to destination is \(d_p(S, X) - w(S, p)\). If port \(p\) satisfies the viability test, this implies that the shortest distance from the next router following \(p\) to the destination is strictly smaller than the shortest distance from S to the destination. Since this satisfies our loop-free policy, port \(p\) leads to a viable next-hop.

Once the MPA has obtained information about the distance attribute for each potential next hop, it can easily determine which next hops are certainly viable for
a given destination by applying the viability test. If the distance information in the MPA data structure is not complete (the computation has not terminated or the computation cannot determine all paths in \( D \)) then not all the next hops derived from the paths in \( D \) will be discovered by passing the viability test. However, all the next hops that do satisfy the viability test result from next hops derived from paths in \( D \) and therefore are loop free.

By searching through different paths, the data structure at every node can be updated and improved by decreasing the distance attribute for each port as new paths are discovered. Improving these distances will increase the number of next-hops that pass the test and thus can be proven to be viable. As long as the shortest optimal distance attribute \( d_{opt} \) is computed correctly, the resulting selected next hops will be viable.

### 5.6 Algorithms

The construction and update of the data structure is achieved by the MPA in conjunction with an SPT algorithm. In the following discussion, an underlying SPT algorithm is assumed to be available to compute the shortest path tree from a router to every other node in the network. This SPT algorithm could be centralized (in the case of link-state protocols) or distributed (in the case of distance-vector protocols).

While it is essential that the underlying SPT algorithm calculates the *exact* shortest distance from a router to every other node, the MPA does not necessarily have to compute all viable alternate paths to each destination. These alternate paths only provide for a greater routing choice and performance but are not essential for the correctness of the routing procedure. The only constraint is that the selected paths must not violate the loop-free routing policy.

#### 5.6.1 Basic Algorithm

The basic form of the MPA can be implemented as a simple extension to existing routing algorithm. The extension adds the extra information of the MPA data struc-
ture (Table 5.1 to each node and attaches some extra comparisons to the computation already being done by the routing algorithm.

During its execution, the underlying SPT algorithm will constantly compare the distance attributes of two neighboring nodes. All commonly used SPT algorithms such as Bellman-Ford, Dijkstra, or D'Esopo-Pape are comparison-based algorithms, whose basic operation is to compare attributes between two neighboring nodes. Under any of these SPT algorithms, the optimal distance attribute of a node $d_{opt}$ will eventually converge to the shortest distance from the source to that node.

In the data structure of the MPA, the distance attribute of node $X$ (from $S$) computed by the SPT algorithm is contained in the corresponding field $d_{opt}(S,X)$. This field is initialized and updated by the underlying SPT algorithm, independently of other operations of the MPA. The other distance attributes $d_p(S,X)$ are initialized to infinity if $X \neq p$. Otherwise, $d_p(S,X) = w(S,p)$.

Every time the underlying SPT algorithm makes a comparison between nodes $a$ and $b$, $d_{opt}(b)$ is set to the minimum of its old value and $d_{opt}(a) + w(a,b)$. This comparison takes place in practically all known shortest path algorithms. In the case of the link-state protocols, the comparison is performed internally in the router using the link states. In distance-vector protocol the comparison is done by neighboring routers that exchange distance information.

At the same time, while the comparison between nodes $a$ and $b$ is being made by the underlying SPT algorithm, the MPA is triggered and starts making its own comparisons. For every next hop $p$ from $S$, $d_p(S,b)$ is set to the minimum of its old value and $d_p(S,a) + w(a,b)$. In this manner, $d_p(S,X)$ represents the best distance found so far of a path between $S$ and $X$ that passes through $p$.

After the execution of the shortest path algorithm is completed, the data structure in Table 5.1 is now updated. If the entry for a next hop $p$ satisfies the viability test, then it can be used as a viable alternative hop to reach the destination.

The number of viable next hops discovered by the basic MPA depends on which underlying SPT algorithm the router is using. The most commonly used SPT engine is Dijkstra's algorithm. For example, it is the one used in OSPF. If Dijkstra is used as the
underlying SPT algorithm, then all the paths in $S(S, X)$ will be probed by the MPA. We can see this by following the execution of Dijkstra's algorithm. The underlying Dijkstra algorithm will grow an SPT progressively by successively attaching nodes with monotonically increasing shortest distances from the source node. Therefore, any path whose shortest distance from the source is monotonically increasing will also be probed by the MPA extension via the parallel comparisons. If two such paths with the same first hop merge at a particular node $X$, the MPA selects the best of the two (shortest path) and the probing continues along the better of the two paths, since only the better of the two paths is needed to perform the viability test. On the other hand, a path $P$ that is in $D$ but not in $S$ will not be probed by the MPA when the underlying algorithm is Dijkstra. This is because at the link in $P$ where the distance from the source decreases, the child node will be visited by Dijkstra before the parent node, preventing the MPA extension from propagating the probe along this link.

Therefore, since the paths considered are all those contained in $S$ (and not those outside), at the end of the execution of the basic MPA, $d_p(S, i)$ corresponds to the distance of the shortest path in $S(S, i)$ that traverses the next hop $p$. However, this algorithm might not probe the paths that are in $D(S, X) \supseteq S(S, X)$ but not in $S(S, X)$. Note that the first hop of any path in $D(S, X)$ is always a viable path, but this algorithm might not be able to find it. Nevertheless, the set $S(S, X)$ might comprise almost all of the paths in $D(S, X)$ and might be sufficiently large to generate enough viable next hops. Furthermore, the next hops generated by the paths in $D(S, X)$ but not in $S(S, X)$, while guaranteeing loop-free routing, might not be desirable. This is because they correspond to paths that do not steadily move away from the source but instead are moving towards the source at some point of the trajectory. Forwarding a packet through the corresponding next hop might cause the packet to travel an unnecessarily long distance. If we still want the MPA to be able find the next hops associated with the paths in $D(S, X)$ as well as with those in $S(S, X)$, a more complex search is required, as shown in the next subsection.

Note that when other underlying SPT algorithms such as Bellman-Ford are used
with the MPA, it may not possible to know exactly what sets of paths will be probed by the MPA and how many viable next hops will be discovered. The lower bound on the number of paths probed by the MPA when using Dijkstra as the underlying SPT algorithm is a result of the orderly way in which Dijkstra progressively builds an SPT.

Once the MPA has terminated, its data structure will contain a list of viable next hops. The distance attribute for each next hop \( d_p(S,X) \) can be used by the forwarding engine to determine the desirability of the each viable next hop. The smallest \( d_p(S,X) \) corresponds to the next hop \( p_1 \) of the shortest path to destination \( D \). In general, the smaller \( d_p(S,X) \) is, the more desirable \( p \) is as a next hop for \( X \), since it is most likely to lead to a shorter path to \( X \).

To measure the complexity of the basic MPA we introduce the term \( \#P \) which is the number of ports on a router. For simplicity, we assume this to be the same in all routers. The space complexity for each router is clearly \( O(\#P \cdot N) \). To determine the computational complexity, note that we are performing the same steps as the underlying SPT algorithm, except that when an edge is visited (two nodes are compared) the MPA is triggered and \( \#P \) parallel comparisons take place. Thus, the computational complexity is the same as for the underlying SPT algorithm, except that the number of edges \( E \) is multiplied by a factor of \( \#P \). These are the computational complexities for some common SPT algorithms

- **Bellman-Ford**: \( O(\#P \cdot NE) \)
- **Dijkstra with binary heaps**: \( O(\#P \cdot E \log N) \)
- **Dijkstra with Fibonacci heaps**: \( O(\#P \cdot E + N \log N) \)

### 5.6.2 Exhaustive Algorithm

In order to obtain the highest possible number of viable next hops, one needs to probe all the paths in \( D \). This is can be done by using a more exhaustive version of the MPA.
Like the basic version, the exhaustive MPA works along with any underlying SPT algorithm. However, the underlying SPT algorithm requires some slight modification since it needs to be run more than once. The computational complexity might also be slightly higher, although it can be easily parallelized.

The more exhaustive search for next hops can be obtained by running a set of independent SPT algorithms in parallel. For every next hop \( p_i \), any shortest path algorithm (including dynamic ones) is executed using only the distance attribute \( d_{p_i}(S, X) \) in the data structure associated with the destination \( X \). At the end of the executions, the data structure will be thoroughly updated. All of the next hops \( p_i \) that can satisfy the viability test are obtained. These next hops constitute the set of all first hops of the paths in \( D(S, X) \). The distance \( d_p(S, X) \) is the distance of the shortest path in \( D(S, X) \) that traverses next hop \( p \).

The space complexity is once again the same as with the basic MPA: \( O(#P \cdot N) \). The computational complexity is simply \( #P \) times the computational complexity of the underlying SPT algorithm used.

### 5.6.3 Distributed Algorithm

Both the basic and exhaustive MPA work well with centralized SPT algorithms used in link-state protocols. However, for distributed SPT algorithms used in distance-vector or path-vector protocols, a much simpler approach can be used.

The distributed MPA consists of simply using the information provided by the neighboring routers to complete the information in the MPA data structure (Table 5.1). In such protocols (e.g., RIP, BGP), every router advertises to its neighbors the shortest distance between itself and every other router. For example router \( p \) will inform router \( S \) about what are the shortest distances between \( p \) and any other router \( X \). When \( S \) receives this information, it can use it to update the entry for \( d_p(S, X) \). As before, based on the completed MPA data structure, router \( S \) will then use the viability test to determine which next hops are viable for destination \( X \).

Since no extra information is being processed and no extra comparisons are being made, the asymptotic computational complexity of distributed MPA is the same as
that of the original routing protocol. The space complexity is once again $O(\#P \cdot N)$.

5.7 Chapter Summary

The multiple path algorithm (MPA) is an addition to a regular shortest path algorithm used in routers. Its purpose is to obtain alternative routing paths. The MPA is interoperable with most existing shortest path algorithms, including the Bellman-Ford and Dijkstra algorithms used in routing protocols such as RIP and OSPF. These shortest path algorithms share the common mechanism that computes for each node a distance attribute, which keeps improving and converges to the shortest distance when the algorithm terminates. Our MPA can also work over dynamic shortest path algorithms.

The extra next hops obtained by the MPA are guaranteed to forward the packet in such a way that there will never be a loop in the packet’s route even if other routers don’t use the MPA. The loop-less property is guaranteed even in the presence of conventional routers in the area that do not use the MPA. The data structure contained in every node determines how much the packet is advancing toward its destination. If the advance is positive, there clearly can be no loop. This metric can also be used to decide on the desirability of the path. Applications of the MPA include quick forwarding restoration after a link failure, load balancing, and QoS routing with multiple constraints.
Chapter 6

Summary

This thesis deals with the problem of IP routing reconfiguration from three different perspectives in the context of link-state routing protocols.

Firstly, we examine how the next-hop calculation can be speeded up and made more robust by developing new dynamic shortest path algorithms that reduce the computational complexity involved. Then, we analyze different ways in which the communication overhead between the different routers can be minimized by performing routing restoration at a local level. Lastly, we examine how reconfiguration can be speeded up locally and in a very small time scale by improving and adding more information to the routing tables that a router constructs.

6.1 Dynamic SPT Algorithms

We have presented new dynamic algorithms for computing a shortest path tree (SPT) in a directed graph. Our dynamic algorithms are derived using a unified algorithmic framework, under which the best known SPT algorithms (static or dynamic) can also be derived as special instances. Within this algorithmic framework, we have proposed two specific approaches that transform known static SPT algorithms to new dynamic ones. These new dynamic algorithms have theoretical asymptotic complexity bounds that match the best known results. We have also performed extensive simulations and discussed the implications of the simulation results on the average-case as well
as worst-case complexity of our algorithms. Among all the dynamic algorithms we studied, the algorithms using the first incremental method give rise to the best theoretical worst-case time bounds, while those using the second incremental method give rise to the best practical or average running times. The most promising algorithm for practical purposes is the Second Incremental Dijkstra algorithm. Its novelty is that it provably minimizes the number of link changes in the SPT, while exhibiting in our simulations a faster average performance than all the other incremental algorithms.

In Chapter 3, we have presented a new algorithm to recompute the shortest path tree (SPT) in a network after some edges have changed weights. The algorithm is heavily based on the dual linear programming technique and on its physical interpretation. This physical interpretation uses balls and strings to model the system of constraints involved in computing a SPT. Our new algorithm can be seen as a simulation of how such a physical model would naturally reconfigure itself.

The new algorithm uses a new search criterion (that of minimum distance change) to determine in which direction the information should be propagated. Unlike existing dynamic SPT algorithms which select one node at a time, the new algorithm can select entire branches at a time. This allows the algorithm to use the intact structure of the SPT to spread information in the right direction without having to perform as many searches. The new algorithm also makes the minimum number of changes to the SPT structure.

The algorithm visits each affected node only once. It makes fewer extractions from the priority queue (one per branch) than any other algorithm of comparable complexity (one per node). As a result, when using underlying linear search or bucket search, the new algorithm has a lower asymptotic complexity than any other known algorithm. When using a more complicated ordered structure (such as a heap), the asymptotic complexity is the same as for the best known algorithms. Nevertheless, simulations indicate that on average, the new algorithm has the lowest observed complexity.
6.2 Local Routing Restoration

In chapter 4, we have presented two algorithms that can restore routing in a link-state network after a link failure. The branch-update algorithms, while being simpler, is only capable of handling one failure at a time. On the other hand, the vector metric algorithm that restore routing even after multiple link failures.

The main novelty of the vector-metric algorithm is that the link state metric used internally by the shortest path engine at each router is represented by a vector rather than a scalar. The extra information allows the algorithm to operate only in the local neighborhood of the failed link. Our algorithm needs to inform the minimum number of routers about any given link failure. Later on, if we desire to increase routing optimality, the size of the set of informed routers can increase at any time. Likewise, when a link recovers, only the same set of informed routers needs to be notified.

By limiting the amount of information broadcast after a link failure, algorithms effectively reduce the amount of bandwidth and computational overhead required for loop-less network routing to be restored. Furthermore, the recovery time is considerably shortened, reducing to a minimum the transient routing loops that might occur while the new information propagates.

When using this algorithm, temporarily congested links can also quickly divert traffic to alternative paths by pretending to have failed. Since the number of routers that need to be informed is small, such updates can happen more quickly and frequently than conventional link-state protocols would allow. After a short time, when there is no more congestion, the link can pretend to recover, and routing will proceed as before.

6.3 Multiple Path Algorithms

The multiple path algorithm (MPA) is an addition to a regular shortest path algorithm used in routers. Its purpose is to obtain alternative routing paths. The MPA is interoperable with most existing shortest path algorithms, including the Bellman-
Ford and Dijkstra algorithms used in routing protocols such as RIP and OSPF. These shortest path algorithms share the common mechanism that computes for each node a distance attribute, which keeps improving and converges to the shortest distance when the algorithm terminates. Our MPA can also work over dynamic shortest path algorithms.

The extra next hops obtained by the MPA are guaranteed to forward the packet in such a way that there will never be a loop in the packet’s route even if other routers don’t use the MPA. The loop-less property is guaranteed even in the presence of conventional routers in the area that do not use the MPA. The data structure contained in every node determines how much the packet is advancing toward its destination. If the advance is positive, there clearly can be no loop. This metric can also be used to decide on the desirability of the path. Applications of the MPA include quick forwarding restoration after a link failure, load balancing, and QoS routing with multiple constraints.
Bibliography


