An Evaluation of Shortest Path Algorithms on Real Metropolitan Area Networks

by

David Johansson

LIU-IDALITH-EX-A--08/059--SE

2008-12-18
Final Thesis

An Evaluation of Shortest Path Algorithms on Real Metropolitan Area Networks

by

David Johansson

LIUIDA/LITH-EX-A--08/059--SE

2008-12-18

Supervisor: Robert Sundberg, Netadmin Systems i Sverige AB
Examiner: Ulf Nilsson, Linköping University
Abstract

This thesis examines some of the best known algorithms for solving the shortest point-to-point path problem, and evaluates their performance on real metropolitan area networks. The focus has mainly been on Dijkstra’s algorithm and different variations of it, and the algorithms have been implemented in C# for the practical tests. The size of the networks used in this study varied between 358 and 2464 nodes, and both running time and representative operation counts were measured.

The results show that many different factors besides the network size affect the running time of an algorithm, such as arc-to-node ratio, path length and network structure. The queue implementation of Dijkstra’s algorithm showed the worst performance and suffered heavily when the problem size increased. Two techniques for increasing the performance were examined: optimizing the management of labelled nodes and reducing the search space. A bidirectional Dijkstra’s algorithm using a binary heap to store temporarily labelled nodes combines both of these techniques, and it was the algorithm that performed best of all the tested algorithms in the practical tests.

This project was initiated by Netadmin Systems i Sverige AB who needed a new path finding module for their network management system NETadmin. While this study is primarily of interest for researchers dealing with path finding problems in computer networks, it may also be useful in evaluations of path finding algorithms for road networks since the two networks share some common characteristics.
**Acknowledgements**
This thesis is dedicated to my grandmother Bernice Johansson who recently passed away. She always had a positive attitude and encouraged me to finish this report. I would also like to thank my family, and especially my wife Yui Johansson, for all support during my work.

I would like to thank Joel Nilsson at Netadmin for initiating the contact which led to this thesis project, and my supervisor Robert Sundberg at Netadmin who guided me through my work.

Finally I would also like to thank my examiner Ulf Nilsson at Linköping University for his help and advice, and Kim Nilsson for his valuable feedback on the report.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Objective</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Background</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>Problem description</td>
<td>1</td>
</tr>
<tr>
<td>1.4</td>
<td>Scope</td>
<td>2</td>
</tr>
<tr>
<td>1.5</td>
<td>Method</td>
<td>2</td>
</tr>
<tr>
<td>1.6</td>
<td>Structure of the report</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Analysis of the problem</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>The network</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>The current path finding module</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Primary path with least hops</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Path with minimal distance</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>Path with maximum bandwidth</td>
<td>7</td>
</tr>
<tr>
<td>2.6</td>
<td>Path between locations</td>
<td>7</td>
</tr>
<tr>
<td>2.7</td>
<td>Path constraints and requirements</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Survey of possible shortest path algorithms</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>The labelling method</td>
<td>9</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Label-setting vs. label-correcting shortest path algorithms</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Unweighted shortest path algorithm</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>Dijkstra’s algorithm</td>
<td>11</td>
</tr>
<tr>
<td>3.4</td>
<td>Bidirectional Dijkstra’s algorithm</td>
<td>12</td>
</tr>
<tr>
<td>3.5</td>
<td>Heap implementations of Dijkstra’s algorithm</td>
<td>14</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Binary Heap Implementation</td>
<td>15</td>
</tr>
<tr>
<td>3.5.2</td>
<td>4-Heap Implementation</td>
<td>15</td>
</tr>
<tr>
<td>3.6</td>
<td>Dial’s algorithm</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Previous evaluations of shortest path algorithms</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Cherkassky et al.’s evaluation</td>
<td>18</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Shortest path algorithms</td>
<td>18</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Network types</td>
<td>19</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Results of the evaluation</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Zhan and Noon’s evaluation</td>
<td>20</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Shortest path algorithms</td>
<td>20</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Networks</td>
<td>21</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Results of the evaluation</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Implementation aspects</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Data structures for the graph implementation</td>
<td>23</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Node-Arc Incidence Matrix</td>
<td>23</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Node-Node Adjacency Matrix</td>
<td>24</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Adjacency Lists</td>
<td>24</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Arc Lists</td>
<td>25</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Forward and Reverse Star</td>
<td>25</td>
</tr>
<tr>
<td>5.2</td>
<td>Choice of data structure for implementation</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Motivation for choice of algorithms for implementation</td>
<td>27</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Breadth-first algorithm</td>
<td>27</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Bidirectional Breadth-first algorithm</td>
<td>27</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Dijkstra’s algorithm</td>
<td>27</td>
</tr>
</tbody>
</table>
Figures and tables

Figure 1: Graph representation of a network .......................................................... 5
Figure 2: Transformation of network from hardware to city level .......................... 7
Figure 3: The Scan operation .................................................................................. 9
Figure 4: Pseudo code for unweighted shortest path algorithm, based on Weiss (1999) p. 303 ................................................................. 11
Figure 5: Pseudo code for Dijkstra's algorithm ...................................................... 11
Figure 6: Point-to-point Dijkstra's algorithm ....................................................... 12
Figure 7: Unidirectional vs. Bidirectional search space .................................... 13
Figure 8: Dreyfus example .................................................................................... 13
Figure 9: Pseudo code for Dijkstra's algorithm using a heap ............................ 15
Figure 10: Directed graph example ..................................................................... 23
Figure 11: Node-Arc incidence matrix representation of the directed graph example .............................................................................................. 23
Figure 12: Node-Node adjacency matrix representation of the directed graph example .............................................................................................. 24
Figure 13: Adjacency lists representation of the directed graph example .......... 25
Figure 14: Forward and reverse star representation of the directed graph example .............................................................................................. 26
Figure 15: Compact forward and reverse star representation of the directed graph example .............................................................................................. 26
Figure 16: Running time in relation to network size .......................................... 34
Figure 17: Scanned ports in relation to network size ........................................... 35
Figure 18: Labelled ports in relation to network size .......................................... 36
Figure 19: Queue iterations for Dijkstra's algorithm on Network 3 ................. 38
Figure 20: Comparison of running time for Dijkstra’s algorithm on Network 3 when using heap updates and duplicate heap insertions ...... 40
Figure 21: Number of extra Find-Min operations on Network 3 ....................... 41
Figure 22: Performance in relation to heap operation ratio (2-heap vs. 4-heap) ........................................................................................................ 43
Figure 23: Time delta of 2-Heap vs. 4-Heap in relation to heap operation ratio ........................................................................................................ 43
Figure 24: Search space on Network 1 ................................................................. 45
Figure 25: Search space on Network 2 ................................................................. 45
Figure 26: Search space on Network 3 ................................................................. 46
Figure 27: Running time on Network 1 ................................................................. 47
Figure 28: Running time on Network 2 ................................................................. 47
Figure 29: Running time on Network 3 ................................................................. 48
Figure 30: Queue iterations on Network 1 .............................................................. 48
Figure 31: Queue iterations on Network 2 .............................................................. 49
Figure 32: Queue iterations on Network 3 .............................................................. 49
Figure 33: Running time for the bidirectional heap version of Dijkstra’s algorithm on Network 3 compared to other versions .......................... 50
Figure 34: Running time for least-hops algorithms on Network 1 .................... 57
Figure 35: Running time for least-hops algorithms on Network 2 .................... 58
Figure 36: Running time for least-hops algorithms on Network 3 ................. 59
Figure 37: Scanned ports for least-hops algorithms on Network 1............ 60
Figure 38: Labelled ports for least-hops algorithms on Network 1 ........ 60
Figure 39: Scanned ports for least-hops algorithms on Network 2......... 61
Figure 40: Labelled ports for least-hops algorithms on Network 2........ 61
Figure 41: Scanned ports for least-hops algorithms on Network 3........ 62
Figure 42: Labelled ports for least-hops algorithms on Network 3........ 62
Figure 43: Shortest path running time on Network 1.................................. 63
Figure 44: Shortest path running time on Network 2.................................. 63
Figure 45: Shortest path running time on Network 3.............................. 64
Figure 46: Number of scanned ports for shortest path algorithms.......... 64
Figure 47: Number of labelled ports for shortest path algorithms........... 65
Figure 48: Number of heap operations on Network 1.............................. 65
Figure 49: Number of heap operations on Network 2.............................. 66
Figure 50: Number of heap operations on Network 3.............................. 66

Table 1: Scanned nodes in each iteration for Dreyfus example.............. 13
Table 2: Networks on which the algorithms were tested...................... 31
Table 3: Time complexity of algorithms............................................. 33
Table 4: Ranking of algorithms according to running time on each network.
.......................................................................................................... 37
1 Introduction

This report presents the author’s Final Thesis project at the Department of Computer and Information Science (IDA) at Linköping University. A large part of the work was carried out at Netadmin Systems i Sverige AB, the company that provided the subject that was studied.

1.1 Objective

The objective of the project was to develop a new path-finding module for the network administration tool NETadmin, and to examine which algorithm(s) that should be used in the new version of the module by an empirical comparison of their performance. The objective of this report is to present the results of the study.

1.2 Background

Netadmin System i Sverige AB was formed in 2004 and is located in Linköping, but before it became a standalone company it was started up as a project within Wasadata System AB in 1998. Netadmin provides a common platform for managing technology, services and end users in broadband networks. The company has expanded rapidly since its start.

The NETadmin system is used by network owners to manage their networks, and it provides four different user interfaces with functionality that are targeted at different users: network technicians, customer support, service providers and finally the end users. The path-finding module, referred to as PhysicalPath in this report, is primarily of interest to the technicians since they are the ones dealing with the hardware. PhysicalPath has two main functions: (i) to find the primary path between two nodes in the network and (ii) to find all paths between two nodes in the network. These nodes are specified as a hardware ID and, optionally, a port ID.

The paths calculated by PhysicalPath are not used for real time routing of network traffic. The paths are for example used when tagging a VLAN on the hardware along the path between two nodes in the network, or when displaying the route that connects two hardware units on the network map. A database containing all hardware equipment, ports and links is used to gain information about the network, so there is no need for real time probing.

1.3 Problem description

The current version of PhysicalPath that is deployed in NETadmin is not optimized for its tasks. Since NETadmin is used in larger networks today the performance has become an issue, and it also lacks several desired features. This has led to the need of a new version of PhysicalPath that scales better and provides more functionality.
The functional requirements for the new version are:

- Ability to find the primary path between two hardware units/ports in the network.
- Ability to find all paths or a given number of paths between two hardware/ports in the network.
- Ability to find paths between nodes in a higher layer, such as places, suburbs, cities, regions and countries.
- Ability to use different criteria for finding the primary path or sorting the found paths, such as least number of jumps, shortest path, highest bandwidth or highest priority.
- Ability to specify a filter that can be used to constrain the search result according to several criteria.

A tool for testing the algorithms and comparing them against each other was also needed for the study.

1.4 Scope
The scope of the project included the tasks of documenting the current module, producing a requirements specification for the new module, carrying out a survey of possible algorithms, designing and implementing the module and writing this report.

This report was restricted to only cover the algorithms for finding the primary path, although both a depth-first algorithm and Yen’s algorithm (Yen, 1971) were implemented and tested for solving the “all paths” problem. The performance studies do not include memory usage, only time usage is measured. The current module also includes functionality for finding which hardware equipment that is unreachable when a specific hardware unit goes down, but improvements to this functionality was left out due to time constraints.

1.5 Method
The project was conducted through studies of literature on shortest-path algorithms, analysis of the current path-finding module, implementation of a new module with several algorithms and tools for testing their performance. The code was written in C# for Microsoft’s .NET Framework 2.0, and the performance tests were conducted on mirror databases of real metropolitan networks.

1.6 Structure of the report
Chapter 2 gives an analysis of the problem and identifies the requirements that the new module should fulfill.
Chapter 3 gives a theoretical background for label-setting algorithms and describes some of the best-known path finding algorithms.

Chapter 4 presents the results from two previous evaluations of shortest path algorithms.

Chapter 5 covers the aspects to consider when implementing these algorithms and which adjustments that had to be made to fit the network model.

Chapter 6 outlines the performance metrics used for the tests.

Chapter 7 describes the computational tests that were done.

Chapter 8 provides an analysis of the test results.

Chapter 9 presents the conclusions of this project.
2 Analysis of the problem

2.1 The network

The NETadmin system contains a database with detailed information about all the hardware devices and connections in the network. Each connection holds information about source hardware, source port, target hardware, target port, connection type and connection priority. The different connection priorities are primary, secondary and so on. A patch connection is used if there is no primary connection from the device. Which outgoing connection to use depends on which incoming connection the network traffic arrives on.

The connections in the database are directed from the lower levels of the network (e.g. an end user’s switch port) towards the higher level of the network (the core network), but data can flow in both directions. A network may contain domains, which are ring topologies with redundant links that are used to ensure connectivity in case a link goes down. One port may have several connections, but there is only one connection between the same pair of ports. All ports on the same hardware are assumed to be connected to each other, unless the port is specified as a 1:1 patch port or if the incoming connection is a patch connection. A connection arriving at a 1:1 patch port may only continue out from a certain port connected to this port, and a patch connection may only continue through a connection from the same port as it arrives on.

Geographical information about each hardware unit is also stored in the database. A hardware unit is located in a defined place, which in turn is located in a specific suburb that belongs to a city. The cities are then located in a region, and regions are located in countries, which constitutes the highest geographical level. Hence there is a total number of six different levels in which you can see connections: countries, regions, cities, suburbs, places and finally at the hardware level. The hardware equipment in the network may also belong to several different companies, so each hardware has an attribute specifying which company it belongs to.

The network can be described as a directed graph $G = (N, A)$ consisting of a set of nodes $N$ and a set of arcs $A$ where the combination of a hardware and port is represented by a node and a connection by an arc; the arcs are pairs $(i, j)$ where $i, j \in A$. The total number of nodes and arcs are denoted by $n$ and $m$, respectively. Since there can be rings in the network, the graph may contain cycles. A path in the graph is specified as a sequence of nodes $n_1, n_2, n_3, ... , n_N$ such that $(n_i, n_{i+1}) \in A$ for $1 \leq i \leq N$. If there is an arc $(i, j)$ directed from $i$ to $j$ there is also an arc $(j, i)$ from $j$ to $i$ because data can flow in both directions of a network link. The arcs can be assigned several values to represent different attributes of the network connections, for example $c_{ij}$.
to represent the cost of the connection between $i$ and $j$. Figure 1 shows a simple network and its corresponding graph representation.

![Network and Graph](image)

**Figure 1: Graph representation of a network.**

It is important that the graph is up to date, so it has to be regenerated from the database each time a search is done since the network topology might have changed since the last search. This will unfortunately have a negative impact on the performance.

### 2.2 The current path finding module

The current path finding module in NETadmin is developed in C++. It contains functions for searching the primary path as well as all possible paths between two nodes in the network. Both of these load the network structure from the database into the memory before starting the search. The connections are stored in an arc list, which is space efficient. The drawback, however, is that a search for connections from a specific node requires an iteration over the whole list, which results in bad performance.

The algorithm that is used to find the primary path tries to find the path by doing four different tests. The first thing it does is to expand the list of hardware equipment along the primary connection path from the source hardware as far as possible, but stops if it finds the target hardware. Then it does the same thing in the opposite direction, from the target hardware to the source hardware. The algorithm will abort the search and return if it finds more than one primary path to take at any step. After this is done, it checks the two lists to see if there is any hardware equipment in the two lists that connect the primary path from the source hardware to the primary path from the target hardware. Finally it checks which one of these possible paths, if any found, that has the least number of hops, and returns this path.
The problem with this algorithm is that it only works when there is exactly one primary connection for every hardware unit along the path, which is not always the case.

The algorithm that returns all possible paths uses a depth-first search technique where a recursive function calls itself for every possible path to take at every node. The recursive function stops if it reaches the target node, if it detects a loop or if there are no more possible paths from the current node.

If more than one direct connection is found between the same two hardware equipment, it will only do a depth-first search for the first connection found and skip the rest. The resulting list of paths was sorted in order of increasing length before it was returned. The biggest problem with this approach is that it may take very long time to run on complex networks, and it did not terminate on some networks that were tested.

### 2.3 Primary path with least hops

The problem of finding a least-hops path between two equipments in a network can be transformed into the problem of finding a least-hops path between two nodes in the graph representation of that network. The objective is to find a valid path from the source $S$ to the target $T$, which contains a minimal number of arcs. A path is only valid if there exists arcs in the graph that connects $S$ to $T$ in the given order and all constraints on the path are fulfilled.

At first glance, this appeared to be equivalent to solving the shortest path problem in an unweighted graph (i.e. a graph where all arcs have a length equal to 1). However, in the case at Netadmin it is also important that these paths not only have the least number of hops but also the highest possible priority on the links. If two or more paths are found that have the same number of hops, the one with the highest priority must be chosen. Since the paths may contain several arcs, the sum of the priorities of these arcs is calculated in order to compare them against each other. This will effectively be a comparison of their average priority because the priority is only measured when two paths contain the same number of arcs.

A search for a primary path should by default only consider paths of primary or patch connections, but it may also be used to find a path that includes connections of lower priorities by explicitly specifying this in the search.

### 2.4 Path with minimal distance

This task is equivalent to finding the shortest valid path between two nodes in a weighted graph, where each arc has a defined length. The lengths of all arcs in the path from $S$ to $T$ are summed up to give the total distance, and it is this value that the algorithm will try to minimize.
The algorithm should use the number of arcs as a second metric to compare if two or more paths have the same distance, and if they are still equal it should choose the one with the highest priority. If more than one path would remain after all these comparisons, the algorithm will simply choose the first of them that it finds.

### 2.5 Path with maximum bandwidth

Finding a maximum bandwidth path in a weighted graph where each connection has a maximum bandwidth defined is a little bit different to the other two problems mentioned above. Instead of minimizing the sum of a metric, the algorithm must find the path that has the highest value on its lowest bandwidth arc since each path’s bandwidth is limited by its slowest link. Just as for the shortest path problem, the number of links and then priority should be considered when two or more paths have the same maximum bandwidth.

![Network at the hardware level](image1.png) ![Network at the city level](image2.png)

**Figure 2:** Transformation of network from hardware to city level.

### 2.6 Path between locations

A path between hardware equipment is rather straightforward to specify, but a path between two locations needs to be defined a little more precisely since there are no connections that directly connect one location to another. Instead, the connections between locations are dependent on the connections between the hardware equipments in these locations. A suggestion is to define a path between location $L_S$ and $L_T$ to be a path between hardware equipment such that only the first hardware belongs to $L_S$ and only the last
hardware belongs to $L_T$. The hardware to start from could therefore be any hardware in $L_S$ that has a connection to hardware outside of $L_S$ and the target hardware could be any hardware in $L_T$ that is reachable from hardware outside of $L_T$. The two locations $L_S$ and $L_T$ can be places, suburbs, cities, regions or countries. Figure 2 illustrates an example of a transformation of a network from the hardware level to the city level.

2.7 Path constraints and requirements

It is desirable to be able to constrain the search according to different criteria by specifying a filter. This filter can be applied when the graph is created before the search. The desired constraints in the filter that may be applied are:

- Allowed Path Types (Primary, Secondary etc.)
- Allowed Connection Types (Ethernet, Fiber, Wifi etc.)
- Allowed Hardware (by ID, Type, Function or Manufacturer)
- Allowed Companies (that the hardware belongs to)
- Allowed VLAN
- Allowed Locations
- Forbidden Hardware (by ID, Type, Function or Manufacturer)
- Forbidden Companies (that the hardware belongs to)
- Forbidden VLAN
- Forbidden Locations
- Minimum Bandwidth (downstream and upstream independently)

It is also desired to have the possibility to put requirements on the path that must be fulfilled, these are:

- Hardware/Ports that must be included in the path
- Domain Types that must be included in the path

These requirements are much harder to deal with and can not be filtered out at the time of graph creation. The problem relates to the well-known Travelling Salesman Problem in which a salesman has to find the optimal route for visiting all cities in a region just one time. To examine heuristics for efficiently solving this problem falls outside the scope of this project. Instead, a simple post-search filter has been implemented to facilitate this functionality. The filter examines all the found paths and tests them against the requirements, and it returns the path(s) that fulfill them.
3 Survey of possible shortest path algorithms

The shortest path problem is one of the fundamental problems in graph theory, and extensive research has produced many algorithms for solving it. The shortest path problem is actually relatively easy to solve efficiently (Ahuja et al. 1993), but it assumes that there are no cycles with negative distances in the network. If such cycles are present it results in an infinite loop since traversing the arcs in the negative cycle always reduces the cost.

Intuitively it may seem that finding the shortest path between two points in a network should be much faster than finding the shortest path from a source to all other points in a network, but there are actually no known algorithms that do that any faster than by a constant factor (Weiss, 1999).

3.1 The labelling method

Most shortest path algorithms are based on the labelling method (Cherkassky et al. 1993). The method keeps track of each node’s distance \(d(i)\), parent node \(p(i)\) and status \(S(i)\), where the status can be either unreached, labelled or scanned. Before the algorithm starts, all nodes have \(d(i) = \infty\), \(p(i) = \text{null}\) and \(S(i) = \text{unreached}\). At the start, the distance of the starting node is set to 0, \(d(s) = 0\), and it is marked as labelled, \(S(s) = \text{labelled}\). Then the Scan operation is applied to all labelled nodes until none exist and the method terminates.

![Figure 3: The Scan operation.](image)

The Scan operation that is applied to labelled nodes is given in Figure 3, where the length of an arc between \(i\) and \(j\) is denoted by \(l(i, j)\). In each iteration, some unreached and scanned nodes may become labelled. The distance label \(d(i)\) is an upper bound on the shortest path from node \(s\) to \(v\) during algorithm execution, and the Scan operation tries to lower the distance label for all successors of the node that it scans. Eventually the distance label \(d(i)\) represents the optimal distance for the shortest path to node \(i\) when the algorithm has terminated (which only happens if the graph doesn’t contain any negative length cycles).

Note that this algorithm produces a shortest path tree from node \(s\) to every other node \(i \in N\), where \(d(i)\) holds the shortest distance to \(i\) and the path can
be obtained by tracing backwards from $p(i)$. The two most important factors for the algorithm’s performance is how it selects the next labelled node to scan and which data structure that is used to maintain the set of labelled nodes (Zhan, 1997).

### 3.1.1 Label-setting vs. label-correcting shortest path algorithms

The algorithms are often divided into two groups: label-setting and label-correcting. The difference between them is how they update the cost for reaching a node in each iteration; label-setting algorithms sets a permanent cost each time while label-correcting algorithms can update the costs in later iterations. Label-setting algorithms are much more effective, but can only be applied to shortest-path problems in acyclic graphs (a graph without cycles) or graphs with non-negative arc lengths.

Only label-setting algorithms will be examined in this report since there are no negative costs associated with the links in the networks.

### 3.2 Unweighted shortest path algorithm

Although we observed in the analysis (2.2) that this algorithm cannot solve the least hops primary path problem because of the necessity to include link priorities, it is still of interest for this research since it provides a lower bound for the running time when comparing the performance of the other algorithms, and the algorithm has therefore been implemented.

The algorithm that solves the unweighted shortest path problem uses a strategy known as breadth-first search. It can be constructed by first setting the cost for reaching all unknown nodes to infinity ($\infty$). We then start with setting the distance for the node to start from to 0 and then put it in a queue. Then we loop while the queue is not empty, and in each iteration we collect the first node $i$ in the queue and for each node that is adjacent to $i$ and has a distance set to infinity, we set its distance to $d(i) + 1$, its parent node to $i$ and then put it in the queue. If the target node is found we can exit the loop, and the full path can be retrieved by recursively following each node’s parent from target to source. A pseudocode for this algorithm is given in Figure 4.

The running time for this algorithm is $O(m + n)$ if the data structure used for the graph enables a search for adjacent nodes in time proportional to the number of adjacent nodes (e.g. adjacency list implementation) (Weiss, 1999).
Figure 4: Pseudo code for unweighted shortest path algorithm, based on Weiss (1999) p. 303.

3.3 Dijkstra's algorithm

Dijkstra’s algorithm uses the labelling method, but always applies the scan operation to the labelled node with the total lowest distance from the start node. This will ensure that the algorithm always scans the nodes in order of increasing distance, and the scanned node can be marked as permanently labelled since its distance cannot be decreased by subsequent scans. Therefore, Dijkstra’s algorithm only needs to scan each node once.

```java
Path getUnweightedPath(Node source, Node target){
    Queue q = new Queue;
    Node i, j;
    source.dist = 0; q.enqueue(source);
    while(!q.isEmpty()){  
        i = q.dequeue();
        for each j adjacent to i{
            if(j.dist == INFINITY){
                j.dist = i.dist + 1;
                j.path = i;
                if(j = target))
                    return BuildPath(source, target);
            }
        }
    }
    return null; //No path found
}
```

Figure 5: Pseudo code for Dijkstra's algorithm.

```java
void Dijkstra(Node source){
    Queue labelled = new Queue;
    Node i, j;
    source.dist = 0; labelled.enqueue(source);
    while(!labelled.isEmpty()){ //Get node to scan
        i = RemoveMinNode(labelled);
        i.known = true;
        for each j adjacent to i{
            if(!j.known){
                if(i.dist + dist(i,j) < j.dist){
                    j.dist = i.dist + dist(i,j);
                    j.path = i;
                    if(!j.labelled){
                        labelled.enqueue(j);
                        j.labelled = true;
                    }
                }
            }
        }
    }
}
```

Figure 5: Pseudo code for Dijkstra's algorithm.
A simple implementation of the algorithm will loop through all labelled nodes in each iteration to find the one with the minimum distance and do at most one update per arc (which takes constant time), so the running time will be \(O(m + n^2) = O(n^2)\). Figure 5 shows pseudo code for Dijkstra’s algorithm.

When a point-to-point path is desired, as in the case of this study, terminating the algorithm as soon as it has picked the target node for scanning can reduce the practical running time. Pseudo-code for this version of Dijkstra’s algorithm is given in Figure 6.

![Path Dijkstra](image)

**3.4 Bidirectional Dijkstra’s algorithm**

Performing a bidirectional search with Dijkstra’s algorithm can often reduce the practical running time for finding point-to-point shortest paths. As the name implies, this algorithm performs Dijkstra’s algorithm in both directions simultaneously, from the source towards the target and from the target towards the source. It stops when a node has been scanned from both directions. Searching from both the source and target in a homogenous graph can reduce the search space to approximately half the size compared to only searching from the source, which is illustrated in Figure 7.
It is important to not assume that the shortest path always goes through the node where the forward and backward searches meet. Dreyfus (1969) points this out with an illustrative example given in Figure 8.

The bidirectional Dijkstra’s algorithm would scan the following nodes:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Forward Scan</th>
<th>F. Distance</th>
<th>Backward Scan</th>
<th>B. Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>0</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>2</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>4</td>
<td>C</td>
<td>4</td>
</tr>
</tbody>
</table>

The forward and backward scan would meet at C, and the path A-C-B has a distance of 8. However, the path A-D-E-B would have a distance of 7 which
is shorter so it is obvious that the shortest path not always goes through the first node that has been scanned in both directions.

The correct way of finding the shortest path using bidirectional Dijkstra’s algorithm is to consider which path that is the shorter of two options when the scan meets at a node $i$ (Ikeda et al. 1994):

1. The path from $Start$ to $i$ combined with the path from $i$ to the $Target$.
2. The path from $Start$ to a node $a$ combined with the arc $(a, b)$ and the path from $b$ to $Target$, where $a$ has been scanned by the forward search and $b$ has been scanned by the backward search and the arc $(a, b)$ is the arc that minimizes this total distance.

In Dreyfus example, the second option would give the path $A-D-E-B$, which yields the shortest path.

### 3.5 Heap implementations of Dijkstra’s algorithm

The operation of finding the next labelled node with minimum distance is a bottleneck operation in the basic implementation of Dijkstra’s algorithm. It spends $O(n)$ time in each iteration scanning down the list of labelled nodes. The algorithm can be made faster for sparse graphs, such as the networks in this study, by using a priority queue instead of a simple list for storing the labelled nodes.

A priority queue is an abstract data type, which supports the operations $insert$, $deleteMin$ and $decreaseKey$. The $deleteMin$ operation removes the object with the lowest value from the priority queue, and the $decreaseKey$ operation updates the value of an object in the priority queue to a lower value. A priority queue is often implemented using a heap data structure and there are several different types of heaps, all with different advantages and disadvantages.

To use a priority queue for storing the labelled nodes, Dijkstra’s algorithm would be modified to use the $deleteMin$ operation to find the next minimum distance node to be scanned. A node position in the priority queue must be kept in memory in order to update its distance label with the $decreaseKey$ operation. This can be done by assigning a $heap\ position$ to all labelled nodes and having the heap operations to update it whenever a node position is changed within the heap. Pseudo code for the heap version of Dijkstra’s algorithm is shown in Figure 9.
Weiss (1999) suggests a different method where a new insertion is done instead of updating a node distance label, since the smallest distance would always be picked first in the deleteMin operation. This will of course lead to multiple entries of the same node in the heap, so every node that is picked from the heap with the deleteMin operation must be checked so that it hasn’t already been scanned. It is simply discarded if it has been scanned and the next one is picked instead. The drawback of this approach is that the heap will grow larger and it requires $O(m)$ deleteMin operations instead of $O(n)$, so it might not be a better solution in practice.

3.5.1 Binary Heap Implementation
A binary heap stores the labelled nodes in a binary tree with the smallest distance label at the root (top). The insert, deleteMin and decreaseKey operations all take $O(\log n)$ time, where $n$ is the number of objects in the heap. This means that Dijkstra’s algorithm using a binary heap would run in $O(m \log n)$ time.

3.5.2 4-Heap Implementation
A 4-heap is very similar to a binary heap, but instead of storing two values at each level of the heap it stores four. This has the advantage that it only requires half the time compared to the binary heap for insert and decreaseKey operations, $O(\log_4 n)$. There is a drawback however, the deleteMin operation on a 4-heap requires $O(4\log_4 n)$ since it needs three
comparisons at each level instead of one to find out which of the child nodes that is the smallest when it rearranges the heap.

A 4-heap should have an advantage over a binary heap when the number of insertions is much greater than the number of deleteMin operations, and both Wiess (1999) and Goldberg (2001) suggests that 4-heaps may outperform binary heaps in practice.

3.6 Dial’s algorithm

Dial’s algorithm is a version of Dijkstra’s algorithm that uses buckets to store nodes with finite temporary labels in a sorted fashion. The buckets are numbered with a range from 0, 1, 2, …, nC, where n is the number of nodes in the graph and C is the largest length of an arc, so nC is the upper bound of any temporary labelled node’s distance. Each bucket stores all temporary labelled nodes with a distance that equals to the bucket’s number. This enables us to scan the buckets in increasing order from 0 and up until we find the first non-empty bucket when we select the next node to scan.

All temporary labelled nodes that are updated during a scan are moved to the bucket that corresponds to its distance. Note that we don’t need to begin from bucket 0 when we continue with the next selection of node to scan, we can continue from where we found the last one since there cannot be any nodes with lower distance than that.

The total time of distance updates in this algorithm is $O(m)$ and the number of scans for node selection is $O(nC)$, so the running time is $O(m + nC)$. This is not a very good upper bound if C is large and could easily become worse than the original implementation of Dijkstra’s algorithm, which runs in $O(n^2)$. In many practical cases C is modest in size, however, and the running time in reality is much better than its worst-case scenario (Ahuja et al. 1993).

The algorithm requires $nC + 1$ buckets, which produces a very high memory demand for large networks. Fortunately, it is possible to reduce the memory demand to $C + 1$ buckets if you use them in a wraparound fashion. If we have $d(i)$ as the smallest distance for a temporary labelled node at the beginning of an iteration, the largest distance $d(j)$ for a temporary labelled node at the end of the iteration must be less than or equal to $d(i) + C$ since C is the maximum length of an arc in the graph. This means that for each iteration, we have a maximum distance range from $d(i)$ to $d(i) + C$ and all finitely temporary labelled nodes can be stored in $C + 1$ buckets. By storing the temporary labelled node with distance $d(j)$ in bucket $d(j) \ mod (C + 1)$ we can search through them in a circular fashion, starting over with the first one when we’ve reached the end, and each bucket will only contain nodes with the same distance label.
Due to time constraints, Dial’s algorithm was not included in the practical tests in this study. Since the real networks that were used lacked the distance property (unweighted graphs) it would not be very useful to test it either, and for the least hops and maximum bandwidth cases there would be far too many nodes with equal “distance”, so most of them would go into the same bucket anyway.
4 Previous evaluations of shortest path algorithms

This chapter presents two previous evaluations of shortest path algorithms. The first one was done in 1993 by Cherkassky, Goldberg and Radzik and is an extensive empirical study on 17 shortest path algorithms on different types of simulated networks. The second study, done in 1996 by Zhan and Noon, tested 15 of those algorithms on 21 real road networks in the US.

4.1 Cherkassky et al.’s evaluation

4.1.1 Shortest path algorithms

The evaluation was done on a Sun Sparc-10 workstation and the algorithms were implemented in the C programming language. The main experiments were carried out with eight different algorithms:

- Bellman-Ford-Moore: With parent checking
- Dijkstra: Double buckets
- Dijkstra: k-array heap
- Incremental Graph: Pape-Levit implementation
- Incremental Graph: Pallottino implementation
- Threshold algorithm
- Topological Ordering: Basic implementation
- Topological Ordering: With distance updates

They also did a special comparison of different versions of Dijkstra’s algorithm on a subset of the problems in order to show their strengths and weaknesses. The algorithms that were tested were three heap versions and four bucket versions:

- Dijkstra’s Heap – k-array (k set to 3)
- Dijkstra’s Heap – Radix
- Dijkstra’s Heap – Fibonacci
- Dijkstra’s Buckets
- Dijkstra’s Buckets – Overflow Bag
- Dijkstra’s Buckets – Double
- Dijkstra’s Buckets – Approximate

The reader is referred to Cherkassky et al.’s (1993) paper for detailed information about these algorithms.
4.1.2 Network types

The experiments were done on several different types of networks in order to get an indication of which algorithms perform best on a certain problem domain.

Square grids

The first type of network is a square grid where each node corresponds to a point on a plane with integer coordinates \([x, y]\) where \(1 \leq x \leq X, 1 \leq y \leq Y\) and \(X = Y\). Each of these nodes are connected with an arc forward \(([x, y], [x + 1, y])\), up \(([x, y], [x, y + 1(\text{mod} Y)]\)) and down \(([x, y], [x, y - 1(\text{mod} Y)]\))

and there is also a special source node which is connected to each node in the first “column” \(([1, y], 1 \leq y \leq Y)\). All arc lengths are selected uniformly from the interval \([0, 10000]\).

Wide grids and long grids

The wide grid has exactly the same properties as the square grid except that the value of \(X\) is fixed at 16, so the width grows with the network size. The long grid on the other hand has the value of \(Y\) fixed at 16, which means that its length grows with the network size.

Harder grid problems

Two more complex networks were also tested, in which a collection of arcs are connecting randomly selected pairs of nodes within a “column” plus a collection of arcs from lower numbered columns to higher numbered columns. One uses non-negative arc lengths and the other uses non-positive arc lengths.

Random networks

These networks are created by constructing a Hamiltonian cycle with arc lengths set to 1 and then adding random arcs with a length in the range of \([0, 10000]\). The sparse graph had \(m = 4n\) and the dense graph had \(m = n^2 / 4\). Tests were also done on a graph where the number of nodes and arcs was fixed, but the arc length was selected from a range of \([0, U]\) where the upper bound \(U\) was set to 1, 10, 100, 10000 and finally 1000000. This experiment was done to find out which impact the arc length has on the different algorithms.

4.1.3 Results of the evaluation

The result of their evaluation concludes that there is no single algorithm that performs best on all types of networks. Dijkstra’s algorithm is robust and performs well on networks with nonnegative arc lengths, with the double bucket implementation being the best overall. For graphs with many negative arcs the Topological ordering algorithm with distance updates is suggested as the best choice, but these will not be discussed further since this report only deals with networks containing nonnegative arcs.
Variations of Dijkstra’s algorithm

All implementations except the approximate buckets algorithm do one scan per node. Therefore, their difference in performance is due to how they select the next labelled node with minimum distance. In the case of dense networks this work is relatively small compared to the node scans, so all algorithms yield very similar performance. On more sparse networks, the differences become more obvious.

The $k$-array heap implementation performs poorly when the number of labelled nodes is relatively large (e.g. the wide grid problem) because the heap operations are fairly expensive unless the number of nodes on the heap is small. On the other hand, it performs very well when the number of labelled nodes is small (e.g. the long grid problem).

The radix heap implementation generally performs better than the $k$-array heap implementation, the only exception being the long grid problem. This shows that a more advanced data structure may be worth implementing to improve the performance. On the other hand, the Fibonacci heap implementation, which has a better theoretical worst case bound, is generally slower than the $k$-array heap implementation. This shows that an algorithm that is good in theory does not always perform that well in practice.

Dial’s bucket implementation suffers from large memory requirements. Another problem is that it may examine a large number of empty buckets, for example on the long grid problem. The overflow bag variant of this implementation has the problem that the overflow bag may become very large in size and examined many times. The approximate buckets implementation works very well on most problems but suffers when there are many arcs of small length since the same node may be scanned many times.

The double buckets implementation, which uses two levels of buckets, was the best on most problems and considered to have the best overall performance of Dijkstra’s algorithms.

4.2 Zhan and Noon’s evaluation

4.2.1 Shortest path algorithms

Zhan and Noon used the same code for the implementations of shortest path algorithms as Cherkassky et al. and tested 15 of these on real road networks. The algorithms that they tested were the following:

- Bellman-Ford-Moore: Basic implementation
- Bellman-Ford-Moore: With parent checking
4.2.2 Networks

Unlike previous studies, which had used random networks of different types, this study used real road networks. These networks included road networks from ten different states in the USA as well as from the U.S. National Highway Planning Network which spans over the continental United States. Two sets of networks were created, one low-detail and one high-detail, with ten networks in each. The number of nodes in the low-detail networks ranged from 523 up to 2878 while the number of nodes in the high-detail networks ranged from 35793 up to 92792. The arc-to-node ratio was very similar for all these networks, ranging from 2.66 to 3.28.

Zhan and Noon points out two important factors that differentiate these real road networks from the random networks that have often been used in similar studies. The first is that their degree of connectivity (arc-to-node ratio) is rather low with an average value of around 3, which makes them very sparse graphs. This ratio is often set to a higher value for randomly generated networks, which in turn affects the algorithms’ performance.

The second difference is that the random networks are often created with random arc values that are independent of each other, both the connectivity and arc length are generated in a homogenous fashion across the whole network. Real road networks on the other hand are often built up by dense urban areas surrounded by suburban areas which are in turn surrounded by a rural road structure which is much sparser. This is a very interesting observation since it also applies to data network infrastructure, which is
often built up by denser “urban” networks connected together through a much sparser “rural” network.

4.2.3 Results of the evaluation

The results of their tests showed that the two versions of the Incremental Graph algorithm perform best on both small and large networks. Pallottino’s implementation has an advantage as its worst case complexity is polynomial compared to the Pape-Levit implementation which has an exponential worst case complexity. After these algorithms it was the Threshold algorithm that came closest, being around 40% slower than the fastest. They discourage from using Bellman-Ford-Moore and Dijkstra’s naive implementation using a simple queue since these were found to have far worse performance on big networks than the other algorithms.

When searching for a one-to-one or one-to-some path(s) instead of the shortest path to all nodes in the network, they recommend considering one of Dijkstra’s implementations using buckets. The reason for this is that Dijkstra’s algorithm has the advantage of being able to stop once the target node(s) is reached. The approximate buckets implementation is recommended when the maximum arc length is within a limit of around 1500, but for networks with a larger maximum arc length the double buckets implementation is recommended instead.
5 Implementation aspects

5.1 Data structures for the graph implementation

One of the main properties of a graph to consider when choosing a data structure for its representation is the relation between the number of arcs and the number of nodes. If the number of arcs $m$ is close to $n^2$, the graph is considered to be dense, but if $m = \alpha n$ where $\alpha$ is much smaller than $n$, the graph is said to be sparse. This chapter describes some of the ways to represent graphs. Note that some data structures are very space inefficient if the graph is sparse.

![Figure 10: Directed graph example.](image)

5.1.1 Node-Arc Incidence Matrix

This representation stores the graph as an $n \times m$ matrix which contains one row for every node and one column for every arc. In each column $(i, j)$ there is a value of 1 for the row corresponding to node $i$ and -1 for the row corresponding to node $j$, the rest of the column contains the value 0. This means that only $2m$ out of its $nm$ entries are used and hence the node-arc incidence matrix is very space inefficient. (Ahuja et al. 1993)

The graph in Figure 10 can be described with the following node-arc incidence matrix:

```
   (1, 3)  (2, 3)  (3, 4)  (3, 6)  (4, 6)  (5, 3)  (6, 5)  (7, 6)  (8, 6)
1   1    0     0     0     0    0     0    0     0
2   0    1     0     0     0    0     0    0     0
3  -1   -1    1     1     0    -1    0    0     0
4   0    0    -1    0     1    0    0     0     0
5   0    0     0    0     1   -1    0    0     0
6   0    0     0   -1    -1    0    1   -1    -1
7   0    0     0    0    0     0    1    0     0
8   0    0     0    0    0     0    0    0     1
```

![Figure 11: Node-Arc incidence matrix representation of the directed graph example.](image)
5.1.2 Node-Node Adjacency Matrix

This representation uses a $n \times n$ matrix $H = \{h_{ij}\}$ which contains a row and column corresponding to each node. If there is an arc from node $i$ to $j$, the entry $h_{ij}$ will be set to 1, otherwise it will be 0. If additional information is needed about an arc, such as cost or capacity, they can be stored in additional $n \times n$ matrices. The advantage of this model is its simplicity, to check up the existence or cost of an arc or to insert an arc is simply a matter of indexing the matrix and to find all outgoing or ingoing arcs for a node is done by scanning corresponding row or column. The drawback, however, is that the time for this scan is proportional to the number of nodes $n$, which can become big for large networks. Further, to add or delete a node requires reallocation and copying of the matrix which has a time complexity of $O(n^2)$. The node-node adjacency matrix is only space efficient if the graph is dense since it uses $m$ out of $n^2$ entries in the matrix. (Ahuja et al. 1993)

\[
\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

Figure 12: Node-Node adjacency matrix representation of the directed graph example.

5.1.3 Adjacency Lists

In the adjacency list representation, all the adjacent nodes to each node are stored in a linked list. Additional information such as cost or capacity for the arc can also be stored in the list. This means that the space requirement is only $O(n + m)$ (Weiss, 1999) which is a huge improvement for sparse graphs. For undirected graphs, each arc needs to be stored two times, which results in double memory usage. If this is the case, one can store a field, mate, for each arc $(i, j)$ that points to the arc $(j, i)$ so that updates of both arcs can be done easily. An advantage of this representation is that the time to find all outgoing or ingoing arcs for a node is proportional to the number of arcs connected to the node.
5.1.4 Arc Lists

The arc list contains only a list of all arcs \((i_1, j_1), (i_2, j_2), (i_3, j_3), \ldots, (i_M, j_M)\) such that \((i_k, j_k) \in A\) and \(i_k, j_k \in N\) for \(1 \leq k \leq M\).

The advantage of the arc list is its low memory use, \(O(m)\), but the drawback is the access time for a particular arc, \(O(m)\), which is quite bad. The example graph can be stored in an arc list as follows:

\(((1, 3), (2, 3), (3, 4), (3, 6), (4, 6), (5, 3), (6, 5), (7, 6), (8, 6))\)

5.1.5 Forward and Reverse Star

Forward star representation has a similar philosophy to adjacency lists but stores the arcs of each node in a single array instead of linked lists. A unique sequence number is first assigned to each arc to order them; the arcs from node 1 are numbered first, then those from node 2 and so on. The internal order of the arcs coming from the same node is not important. The arc list, which consists of source, target, cost and capacity of each arc, is stored in four arrays, and a pointer is stored for the first arc going out from each node. The arcs going out from node \(i\) will be stored at position \(\text{point}(i)\) to \((\text{point}(i + 1) – 1)\) in the arc-list. This provides an efficient way of determining all outgoing arcs from a node.

To find an equally efficient way of finding all arcs coming into a node, a reverse star representation can be used. This works much the same way as the forward star representation but lists all arcs going into each node instead. The pointer for the reverse star is stored in \(r\text{point}\).
Forward and reverse star representations can be compacted together by using a trace array of size \( m \) which stores the arc number from the forward star representation for each arc that is pointing towards each node (Ahuja et al. 1993).

From the three remaining structures that have been discussed, the arc list is easiest to implement and most memory efficient, but the long access time makes it slow and hence not suitable for the algorithms. The two candidates remaining are the adjacency list and forward (and reverse) star.
representations. The forward star representation’s main advantage is that it is more space efficient than the adjacency list, but the drawback is that the running time for inserting or deleting an arc or node is $O(m)$ whereas it is $O(1)$ for the adjacency list (Ahuja et al. 1993, p. 37).

Since there is already a class implemented for a linked list in the .NET Framework from version 2.0 and above (System.Collections.Generic.LinkedList), the adjacency list data structure was the representation chosen to use for implementing the graphs in this study.

5.3 Motivation for choice of algorithms for implementation

The algorithms that were chosen for implementation are listed here.

5.3.1 Breadth-first algorithm

The breadth-first algorithm was originally chosen for solving the least-hops path problem because of its simplicity and good performance. However, the algorithm was no longer a valid option when it was realized that the least-hops problem also had to consider the priority of the links. Despite this fact, it was still kept as a reference in this study because it provides a lower bound on the execution time to compare the other algorithms with.

5.3.2 Bidirectional Breadth-first algorithm

This algorithm was chosen to get a measure of how much the performance could improve by doing a simultaneous search from both directions.

5.3.3 Dijkstra’s algorithm

The traditional Dijkstra’s algorithm is probably the most well known algorithm for solving the shortest path problem for networks with non-negative arcs, and is included to compare its performance with the versions that use more advanced data structures.

5.3.4 Dijkstra’s algorithm using a binary heap

The binary heap implementation of Dijkstra’s algorithm has often been used as the benchmark to compare other algorithms against (Goldberg, 2001), and is also relatively straightforward to implement. Therefore it was a natural candidate in this study. The heap implementations in this study use the decreaseKey-operation when updating the distance label of labelled nodes. Each node needs to keep track of its position in the heap.

5.3.5 Dijkstra’s algorithm using a binary heap with duplicate insertions

The alternative approach of having multiple insertions instead of updates in the heap was tested to see how it performs in practice compared to the standard binary heap.
5.3.6 Dijkstra’s algorithm using a 4-heap

The 4-heap was tested since it has been suggested that it could produce better running times in practice than the binary heap (Weiss, 1999; Goldberg, 2001).

5.3.7 Bidirectional Dijkstra’s algorithm

The bidirectional Dijkstra’s algorithm is included in the tests to measure how much a bidirectional search may improve the performance compared to the standard version of Dijkstra’s algorithm.

5.3.8 Bidirectional Dijkstra’s algorithm using a binary heap

A bidirectional version of Dijkstra’s algorithm using a binary heap data structure was included to see the impact of using two different strategies together for improving the running time. This algorithm was implemented towards the end of the project and it was only tested on one of the three networks included in this study.

5.4 Algorithm modifications to fit the network model

Because the network treats a connection differently if it’s a patch, 1:1-patch or normal connection, some adjustments of the algorithms had to be done. One way of solving this would have been to adjust the graph at the time of its creation, so that internal arcs would be created between nodes on the same hardware unless they were in a patch or 1:1 patch connection.

The other option is to adjust it in the algorithms at run-time, which was the method chosen in this project. The reason for this was that it only needs to do the adjustment for those nodes that are scanned in the search, which in most cases should be far less than all nodes in the graph.

The algorithms were altered so that looking up adjacent nodes could be done in three different ways depending on the type of arc and node:

- **Patch connection**
  If the arc on which the algorithm arrived to the node on was a patch connection, the algorithm was only allowed to continue out from the same node. The adjacent nodes were therefore only those directly connected to this node with an arc.

- **1:1 patch connection**
  If the node that the algorithm arrived on was in a 1:1 patch connection, it was only allowed to continue out from the node that was listed as the corresponding node in the 1:1 patch connection list.

- **Normal connection**
  If it was not one of the cases above, it was a normal connection and all nodes on the same hardware, except 1:1 patch connections, are considered to be adjacent.
6 Performance metrics

To be able to compare the different algorithms with each other, it is necessary to specify one or more performance metrics that will be evaluated. The ones used in this study are presented in this chapter.

6.1 Time complexity

By analyzing the time complexity of the code for each algorithm, their theoretical worst-case running time can be determined. The advantage of this method is that it is possible to evaluate different algorithms even before they have been implemented, and the measure gives an upper bound for how the running time relates to the size of the problem.

If the time consumption of an algorithm is measured as a function of the problem size, \( f(n) \), we can define its time complexity as \( O(g(n)) \) if \( f(n) \) doesn’t grow faster than \( g(n) \) when \( n \) grows. For example, if we have \( f(n) = 3n^4 + 8n^2 + n \) we would have \( g(n) = n^4 \) as upper bound of the growth rate; the time complexity of the function is therefore \( O(n^4) \). It is only the highest ordered term that is counted, since it will dominate the growth rate, and constants are excluded since their impact will be irrelevant for large \( n \).

An algorithm is said to be polynomial if \( f(n) \) doesn’t grow faster than \( n^k \) for some constant \( k \), and it is exponential if \( f(n) \) grows faster than \( n^k \) for every constant \( k \). A pseudopolynomial algorithm has a polynomial growth rate function of the input size for a (largest) constant in the input data (Holmberg, 2002).

A lower bound is defined as \( \Omega(g(n)) \), which means that there are constants \( c \) and \( n_0 \) such that \( f(n) \geq cg(n) \) when \( n \geq n_0 \) (Weiss, 1999).

6.2 Running time of algorithms

Although the time complexity gives a good comparison of how different algorithms behave in their worst-case scenario, it doesn’t really give an accurate measure for how they perform in practice. An algorithms worst-case scenario is often rare in real tests, and normally an algorithm performs much better than what its time complexity suggests (Ahuja et al. 1993).

Empirical tests of running time are often carried out to get a better measure of different algorithms’ performance on different problem domains. By doing this, it is possible to compare two algorithms in practice to find out which one is normally faster, and although they may have the same time complexity their running time can differ substantially.
6.3 Representative operation counts

Ahuja et al. (1993) argues that many empirical researches over-rely on CPU time as the primary performance measure. Because estimated CPU time depends on many factors, such as computer, compiler, language, implementation style etc., it is often difficult to replicate the results, which is desired for scientific research. Moreover, estimating only the CPU time for the completion of the whole algorithm doesn’t give any information about how the algorithm works or which operation that might be a “bottleneck”.

To get a better performance measure for computational tests it is suggested that the number of times each bottleneck operation is executed should be counted instead of CPU time. The number of operations that needs to be counted is usually not more than three or four even for complex algorithms. These operations are called representative operations because they represent the operations that affect the performance of an algorithm.

Some of the advantages with this performance metric over CPU time are:

1. It allows identification of operations that progressively consume a larger share of computational time as the problem size grows.
2. It allows comparison of algorithms that are implemented in different computer languages and run on different computers.
3. It allows an estimation of a virtual CPU time, which enables comparisons of running times between experiments carried out on different computers.

The idea assumes that any algorithm consists of a finite number of lines of code, and that it is written so that each line of code requires $O(1)$ time units. With the exception of I/O-operations this is fairly reasonable. The CPU time of executing this algorithm would then be a constant multiplied with the total number of lines of code executed. However, it is not necessary to count each time every single line of code is executed since an algorithm often consists of groups of operations that are executed repeatedly. Instead we focus on finding the representative operations for the algorithm, and count how many times these are executed, which gives a good picture of how the whole algorithm behaves. As a simple example, if we have a for-loop in which three lines of code is executed, each taking $O(1)$ time units, it is not necessary to count each execution of these lines. Instead we just count the number of times the for-loop is executed.
7 Computational test

7.1 The networks used in the tests

The algorithms were tested on three networks of different size. These networks were replications of real metropolitan area networks, but their names have been replaced in order to protect Netadmin’s customers. These networks were selected because they represent three common network sizes. Their characteristics are listed in Table 2.

Table 2: Networks on which the algorithms were tested.

<table>
<thead>
<tr>
<th>Network</th>
<th>Hardware</th>
<th>Ports*</th>
<th>Connections**</th>
<th>Connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network 1</td>
<td>160</td>
<td>358</td>
<td>442</td>
<td>2.76</td>
</tr>
<tr>
<td>Network 2</td>
<td>624</td>
<td>1885</td>
<td>1896</td>
<td>3.04</td>
</tr>
<tr>
<td>Network 3</td>
<td>1220</td>
<td>2464</td>
<td>2596</td>
<td>2.13</td>
</tr>
</tbody>
</table>

*Customer ports on switches are not included **Directed (one-way) connections

The connectivity is a measurement of how many connections each node has, and in this case it has been calculated as the number of connections divided by the number of hardware since all ports on the same hardware are considered to be connected to each other implicitly, except for the small number of patch connections. As can be seen in the table above, the networks in these tests are very sparse, \( m \) is a lot less than \( n^2 \).

7.2 Computational analysis

Two types of problems were tested, least-hops paths and shortest paths. For each network, a test was done for paths with a size (number of links) from a range between 2 and 10. The reason for this range of path size is that it was difficult to find longer paths in the networks, due to their hierarchical architecture. The tests measured both the running time of the algorithms as well as their representative operation counts.

7.2.1 Running time

Each test was done 50 times to get an estimate of the running time using the median value. The reason for not using the average value is that a single execution that is slowed down 100 times due to another process interrupting will not affect the median as much as it would affect the average time.

All iterations of each test were done between the same pair of nodes, i.e. it was the same path that was calculated in each iteration. This does unfortunately increase the impact of the network topology on the running time, which the reader should keep in mind when studying the results.
7.2.2 Representative operation counts

The representative operation counts that were selected for all algorithms are the following:

- Number of scanned ports
- Number of labelled ports (temporary values)

For Dijkstra’s algorithm the following measurement was added:

- Number of ports that were updated with new values
- Number of queue iterations for finding the temporary labelled port with lowest cost (only for original Dijkstra, not applicable to the heap versions)
- Number of extra deleteMin-operations when duplicate entries of labelled ports were found (only applicable to Dijkstra’s algorithm with non-updating heaps)
8 Results and analysis

8.1 Comparison of time complexity

The theoretical time complexity of each algorithm is given in the table below.

Table 3: Time complexity of algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>Breadth-First (bidirectional)</td>
<td>$O(m + n)$</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Dijkstra (bidirectional)</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Dijkstra 2-heap (insert only)</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Dijkstra 2-heap (update)</td>
<td>$O(m \log n)$</td>
</tr>
<tr>
<td>Dijkstra 4-heap (update)</td>
<td>$O(m \log_4 n)$</td>
</tr>
<tr>
<td>Dijkstra 2-heap (bidirectional)</td>
<td>$O(m \log n)$</td>
</tr>
</tbody>
</table>

It is clear from this that the heap implementation of Dijkstra’s algorithm is faster than the standard queue implementation in the theoretical worst case scenario. Although the breadth-first algorithm has a better time complexity, it doesn’t solve the actual problem correctly and is therefore only included as a reference to compare against.

Note that the bidirectional versions of the algorithms don’t improve their time complexity since their worst-case scenario might be as bad as for their unidirectional versions. In fact, Dreyfus (1969) even suggests that the bidirectional Dijkstra’s algorithm may have a worse running time than the unidirectional version since a decreasing number of additions and comparisons are necessary to modify the temporary labels as more nodes become permanently labelled. According to him, it takes over $3/5$ of the total work to determine the first half of permanent labels for all nodes. Using a bidirectional algorithm should therefore be more effective only when the total number of permanently labelled nodes required to find the shortest path is less than half of all nodes.

8.2 Running time in relation to network size

By extracting the running time on each network for a path-size of two we get a comparison of the running time in relation to network size. The network size here is counted as the total number of ports the network contains.
Figure 16: Running time in relation to network size.

Although this is a very limited amount of data, it is easy to see that the running time for Dijkstra’s algorithm (without a heap) grows faster than the others, just as it is predicted by the time complexity given in 8.1. The reason for this rapid increase in running time for Dijkstra’s algorithm is that when the network size grows, there will be a larger number of possible paths to examine and this in turn increases the number of labelled ports that needs to be loop through in each scan operation. This becomes clear when looking at Figure 17 and Figure 18 that shows the number of scanned and labelled ports in relation to network size. Although the number of scanned ports on the largest network for Dijkstra’s bidirectional algorithm is lower than the heap versions of Dijkstra’s algorithm, the number of labelled ports actually exceeds all others which make the queue operations expensive.
Another thing to note is that the number of scanned and labelled ports is lower for some algorithms on the medium size network than on the smallest network, which may seem strange. The reason that this can happen is that the networks are not homogenous, which means that the path in the smallest network may have been in a denser network area than the path in the medium size network. The reader should keep in mind that the figures presented in this study do not take the network topology into account when measuring the network size, only the number of ports.
Most studies that compare shortest path algorithms compare their running time in relation to network size, but equally important for this study is the relation between running time and path size. The reason is that we are not interested in the time for finding the shortest path from a source node to all other nodes in the network; we just want the path to the target node. If the source and target are close to each other, it doesn’t affect the running time that much if one network is ten times the size of the other if their search space is of similar size.

What really affects the running time here is the path size, i.e. the number of nodes between the source and target. In general, with higher path size the search space grows; more nodes need to be scanned and even more are labelled. As can be seen from Figure 34, Figure 35 and Figure 36 in Appendix A.1, this greatly affects the running time and the difference between some of the algorithms becomes very clear already when the path size goes beyond 5 nodes.

When looking at the diagrams of the running time in comparison with the path size, the algorithms are divided into three groups of similar behaviour. The first and fastest group contains Breadth-First, Breadth-First Bidirectional and Dijkstra Bidirectional. The second group consists of the

![Figure 18: Labelled ports in relation to network size.](image-url)
three heap implementations of Dijkstra’s algorithm, which all perform with very similar results. And last, the simple implementation of Dijkstra’s algorithm is by far the slowest of all algorithms for every path size. The extended number of labelled nodes that needs to be looped through in each iteration has a big impact on the running time as the path size grows. It is clear that the queue data structure for storing temporary labelled nodes is very inefficient for large number of nodes.

The exception to this pattern is the bidirectional Dijkstra’s algorithm on Network 3 (see Figure 36), where it shows more similarity with group two than with group one. A possible explanation for this is that although the number of scanned nodes are far less than for the heap algorithms, the number of labelled nodes has grown to a “critical mass” due to the network size. The gain of reduced number of scans has been cancelled out by the workload of iterating through the labelled nodes for each scan.

Table 4: Ranking of algorithms according to running time on each network.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Network 1</th>
<th>Network 2</th>
<th>Network 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>BF – BiDirect</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Dijkstra - BiDirect</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Dijkstra - 2-Heap Insert</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Dijkstra - 2-Heap</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Dijkstra - 4-Heap</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

The ranking in Table 4 gives a very similar result for Network 1 and 2, but it differs a little bit when compared with Network 3. The big difference is that the bidirectional Dijkstra’s algorithm has lost ground to the others and now ranks as number 5 instead of number 2, mainly because of poor performance on shorter path lengths. When looking at the performance counts in Figure 42 we can see that the number of labelled ports for the bidirectional Dijkstra’s algorithm actually exceeds those for the unidirectional heap versions, which explains why its running time is slower than theirs. However, for the larger path sizes the order is restored and the bidirectional algorithm outperforms the others.

8.4 Dijkstra’s algorithm: Queue vs. Heap

The results show that using a queue for storing the temporary labelled nodes in Dijkstra’s algorithm is a bad idea. Its performance loses out to all heap implementations on all networks, and the difference grows rapidly with larger network and path size. On the smallest network, the queue implementation loses with a factor of around 2, on Network 2 it is around 5 and on Network 3 it is around 6. When looking at the representative operations count for the shortest path algorithms on Network 3, the reason becomes obvious. Although both the queue and heap implementations scans
and labels the same amount of ports, the number of iterations through the queue grows with an enormous rate as can be seen in Figure 19. For the path size of 2 it is a modest 101 iterations while it has grown close to 10,000 for the path size of 5. But for the path size of 8 it has nearly reached one million iterations! At these levels the queue iterations dominates the time consumption which can be seen by comparing it to the heap implementations.

One strange thing is that the time consumption for the queue implementation is higher for the path size of 8 than that for 10, even though the number of scans and total number of labelled ports are less. A reason for this is found when observing that the number of queue iterations is larger for the former, which may seem contradictory when the number of labelled ports is less. The explanation for this is that both paths reach the same target node but start at different points, which means that the paths may have taken different directions so that the longer path explores fewer nodes in the beginning than the shorter path. This has the effect that the number of iterations through the queue builds up quicker for the path size of 8 than for 10.

### 8.4.1 Comparison of different heap implementations

One of the objectives of this study was to identify if any of the heap implementations is superior to the others on real networks. The ones tested here is the standard 2-heap implementation of Dijkstra’s algorithm, both the version with updates to heap position and the other version which does a new insert into the heap instead of updating the position. The 4-heap

![Figure 19: Queue iterations for Dijkstra's algorithm on Network 3.](image-url)
implementation was also included to test if it actually has a performance advantage in practice as suggested by Weiss (1999) and Goldberg (2001).

**Heap with updates vs. Heap with duplicate inserts**

Weiss (1999) mentions two ways to handle the updates of labelled node distances when using a heap. The first option is to use the *decrease-key* operation for the node on the heap, but this requires that we know its position in the heap and therefore requires extra logic in the implementation so that every node keeps track of its heap position. This requires $O(n)$ extra memory space and requires an additional operation to update the heap position every time a node is moved within the heap.

The second option is to insert the labelled node into the heap again when its distance is updated, which means that there may be duplicates of a node in the heap. However, this does not cause any trouble since the node with the lowest distance will always be selected first from the heap. The only thing to keep in mind is to check that the selected node has not yet been scanned, otherwise it’s just to continue with the next one. The advantage of this solution is that we don’t need to keep track of the heap position for each labelled node, which makes it easier to implement and doesn’t require the extra position attribute for each node. But the size of the heap may now grow larger, $O(m)$ instead of $O(n)$, because of the duplicate inserts, and hence we also get $O(m)$ instead of $O(n)$ deleteMin-operations for finding the next node to scan.

If we compare these two methods in theory, we end up with the first one taking at most $m\log(n)$ time for updates and spend $n\log(n)$ time for finding the minimum distance label, which gives it a time complexity of $O(m\log(n))$. The second method with duplicate inserts requires instead $m\log(m)$ time for updates and $m\log(m)$ time for finding the minimum distance label because the heap may now have grown to a size of $m$ because of the duplicate inserts. This gives the second method a time complexity of $O(m\log(m) + m\log(m)) = O(m\log(m))$ which is worse than for the first method. The time gained by not performing the decrease-key operation on an update is lost by doing an insert instead, and because the heap may be larger this may take longer time. In addition it may also need to perform $m$ delete-min operations instead of $n$, where each operation may take $\log(m)$ time instead of $\log(n)$.

This difference between the methods can be seen in Figure 20, which shows a comparison of running time between the two different heap implementations on Network 3. For the path size of 2 they are nearly identical but when the path size grows their performance difference grows as well, and the version that updates the heap position is clearly better than the other version that uses duplicate inserts.
Figure 20: Comparison of running time for Dijkstra’s algorithm on Network 3 when using heap updates and duplicate heap insertions.

The number of extra Find-Min operations for the insert-only heap version of Dijkstra’s algorithm on Network 3 is presented in Figure 21. It is easy to see the correlation between the performance penalty and the extra operations when you compare it with Figure 20. The only exception is for the smallest path size where this insert-only heap version is slightly faster than the other, but since the running times we talk about here are so small it may be a measurement error. Another possible reason may be that the extra operations for keeping track of the heap position take more time than the 3 extra Find-Min operations that the insert-only version performs.
Figure 21: Number of extra Find-Min operations on Network 3.

2-Heap vs. 4-Heap
Both Weiss (1999) and Goldberg (2001) suggest that 4-heaps may have better performance than 2-heaps. Each node in a 4-heap may have up to 4 child-nodes, which makes the heap shallower than a 2-heap for the same amount of nodes. This improves the running time of inserts and updates (decrease-key) to $O(\log_4(n))$, but the drawback is that it takes more time to order the heap when deleting the minimum node since it requires 3 operations to find the minimum of 4 child-nodes, compared to 1 operation in a 2-heap. This gives the delete-min operation a running time of $O(4\log_4(n))$, and the 4-heap should therefore be more effective when the number of inserts and updates are much greater than the number of delete-Min operations.

For the heap version of Dijkstra’s algorithm, this means that the 4-heap should have an advantage over the 2-heap when the number of labelled nodes and updates are far more than the number of scanned nodes. The notion of “far more” here is important; remember that the time complexity above is for the worst case scenario. What really affects the running time in practice is the average case for these operations. It is important to understand how the heap works and how the network characteristics affect the heap structure in order to analyze this. For the insert operations, a newly
labelled node is inserted at the end of the heap and then percolated up until the correct location is found. This node may be percolated all the way up, giving a running time of $O(\log_d(n))$ for a $d$-heap, but it only happens when this new node has the minimum distance of all nodes in the heap. This is very unlikely unless the network graph contains a shortest path consisting of many short arcs with a **total distance** that is smaller than any of the other arcs. On average, the running time for the insert operation is likely to be much less than the worst case scenario.

The same logic can be applied to the decrease-key operations in which a labelled node is also percolated up in the heap to a new position; it is not that likely that it will have to travel through the whole depth of the heap. On the other hand, when a delete-min operation occurs, the root node of the heap is removed and the hole that is left has to be percolated down until it reaches a level where the last element of the heap can be inserted. It is very likely that this hole has to be percolated all the way down to the bottom of the heap because that is where the last element of the heap was before. This means that the delete-min operation’s average time is most likely the same as the worst case time, which is $O(d\log_d(n))$ for a $d$-heap. The delete-min operation is therefore, on average, much more expensive than the insert and decrease-key operations, which is why the latter two must be used “far more” in an algorithm than the delete-min operation in order to get a performance advantage with the 4-heap over a 2-heap.

To test what this “far more” should be in practice, the number of heap operations was measured on the different networks, and the results are found in Appendix A.5. A **heap operation ratio** was calculated as the sum of insertion and decrease-key operations divided by the number of delete-min operations. This ratio turned out to be between 1.2 and 3.7, depending on network and path size. Figure 22 shows which of the algorithms that performed best for the different heap operation ratios, with the value ‘1’ indicating the faster algorithm and ‘0’ the slower algorithm. Just as the theory predicted, we do see an advantage for the 2-heap on the lower ratios and an advantage for the 4-heap on the higher ratios. On the ratios in the middle, 1.78 up to 3.38, the results are mixed.
Figure 22: Performance in relation to heap operation ratio (2-heap vs. 4-heap).

To further analyze the results, a *time delta* was calculated as the running time of the 2-heap minus the running time of the 4-heap for every heap operation ratio, and the result is presented in Figure 23 together with the heap size. From this it seems like the 2-heap wins by a larger margin than the 4-heap does when it wins, and it is probably due to the fact that the heap size is generally larger for the lower heap operation ratios.

Figure 23: Time delta of 2-Heap vs. 4-Heap in relation to heap operation ratio.
The tests done in this research suggests that the 2-heap should be used instead of the 4-heap unless the number of insertion and decrease-key operations exceeds the number of delete-min operations with a factor over 3. For the real metropolitan area networks in this test the 2-heap appears to be more attractive and Weiss (1999) and Goldberg’s (2001) suggestions of the 4-heap’s superiority cannot be verified.

8.5 Dijkstra’s algorithm: Unidirectional vs. Bidirectional

Another objective of the study was to find out what improvement a bidirectional version of Dijkstra’s algorithm could achieve when finding a point-to-point shortest path. As described in chapter 3.4, the bidirectional version of Dijkstra’s algorithm should be able to reduce the search space to about half the size in a homogenous network.

The number of labelled ports can be used as a measurement of the search space and is presented for the two versions of Dijkstra’s algorithm in Figure 24, Figure 25 and Figure 26. On Network 1, the unidirectional version scans between 1.4 and 2.6 times more ports, with the average search space being the double size of the bidirectional version, just as expected. On Network 2, this factor has grown to 4.4 on average, but on Network 3 it has dropped to just 1.4! The reason for the relatively large search space for Dijkstra’s unidirectional algorithm on Network 2 is probably due to the network structure, it could be that the algorithm reaches a very dense part of the network which is never encountered in the bidirectional version because it terminates before it reaches that far.

On Network 3 the search space is actually smaller for the unidirectional version on path size 2 and 5! This may seem contradictory to what one expects, but again this is due to the network structure. For example, the target node has 32 ports whereas the source node only has 1 port when the path size is 2, which means that the bidirectional version that scans from both directions will label a much larger number of nodes in the beginning than the unidirectional version. But when the path size grows to 8 and 10, the search space for the unidirectional version of Dijkstra’s algorithm is about twice the size of the bidirectional version, which is what we would expect.
Figure 24: Search space on Network 1.

<table>
<thead>
<tr>
<th>Path size</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>95</td>
<td>152</td>
<td>221</td>
<td>245</td>
</tr>
<tr>
<td>Dijkstra - BiDirect</td>
<td>37</td>
<td>106</td>
<td>118</td>
<td>118</td>
</tr>
</tbody>
</table>

Figure 25: Search space on Network 2.

<table>
<thead>
<tr>
<th>Path size</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>147</td>
<td>1170</td>
<td>1462</td>
<td>1755</td>
</tr>
<tr>
<td>Dijkstra - BiDirect</td>
<td>26</td>
<td>237</td>
<td>455</td>
<td>473</td>
</tr>
</tbody>
</table>
The difference in search space between the two versions of Dijkstra’s algorithm is reflected in the running time, but it is not linear with the search space since the number of operations grows faster than the search space. As can be seen from Figure 27, Figure 28 and Figure 29, the running time of the bidirectional version is generally more than twice as fast as the unidirectional version, in some cases up to nearly 29 times faster (Network 2, path size 4)! The exception is for path size 2 on Network 3 where the search space is smaller for the unidirectional version and hence the running time is slower for the bidirectional version.

![Figure 26: Search space on Network 3.](image)
Figure 27: Running time on Network 1.

Figure 28: Running time on Network 2.
The great improvement in running time for the bidirectional version of Dijkstra’s algorithm is mainly due to having two small queues of labelled nodes to iterate through before each scan operation instead of one large that requires a lot more iterations. The number of queue iterations for each algorithm on the three networks is shown in Figure 30, Figure 31 and Figure 32.

Figure 29: Running time on Network 3.

Figure 30: Queue iterations on Network 1.
8.6 Combining improvements to Dijkstra’s algorithm for point-to-point paths: A bidirectional heap version

We have already seen the impact of two different ways for improving the practical running time of Dijkstra’s algorithm; the operations for managing labelled nodes can be done more effectively by using a heap, and the search
space can be reduced by performing a bidirectional search. A natural step is therefore to combine these two techniques in an attempt to further improve the running time. This was only tested for the least hops problem on Network 3, and the result compared to other versions of Dijkstra’s algorithm is presented in Figure 33.

The new version greatly improves the running time and is several times faster than the others. The only exception is for the shortest path size, where its bidirectional nature in this special case happens to increase the search space instead of reducing it. But this loss is very small compared to the huge performance gain for the larger path sizes. This algorithm is clearly the fastest algorithm in this study that solves the least hops and shortest path problems correctly.

<table>
<thead>
<tr>
<th>Path size</th>
<th>Dijkstra</th>
<th>Dijkstra - 2-Heap</th>
<th>Dijkstra - BiDirect</th>
<th>Dijkstra - 2-Heap BiDirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.070</td>
<td>0.077</td>
<td>0.120</td>
<td>0.112</td>
</tr>
<tr>
<td>5</td>
<td>0.830</td>
<td>0.523</td>
<td>0.825</td>
<td>0.315</td>
</tr>
<tr>
<td>8</td>
<td>22.900</td>
<td>4.446</td>
<td>3.507</td>
<td>0.912</td>
</tr>
<tr>
<td>10</td>
<td>35.172</td>
<td>5.766</td>
<td>5.501</td>
<td>1.727</td>
</tr>
</tbody>
</table>

Figure 33: Running time for the bidirectional heap version of Dijkstra’s algorithm on Network 3 compared to other versions.
9 Conclusions and Recommendations

9.1 Impact of network complexity on algorithm performance

There are a lot of factors to consider when choosing an algorithm for solving a shortest path problem. The decision should not only be based on a simple comparison of their theoretical time complexity because they are often overly pessimistic, although they give important information on the worst case scenario in relation to network size. The $O(n^2)$ implementation of Dijkstra’s algorithm, which had the worst time complexity of the algorithms that were tested, was clearly the worst performing algorithm for large problem sizes.

The practical running time for an algorithm does not only depend on the network size, it is much more complex than that. One important factor is a network’s degree of connectivity, which is the arc-to-node ratio. A dense network has a high degree of connectivity and for completely dense networks, where $m = \Omega(n^2)$, Dijkstra’s basic algorithm with the time bound $O(n^2)$ is the best possible. This is not common for real networks, which are often very sparse networks; the metropolitan area networks used in this study had an average arc-to-node ratio of just 2.6. The basic version of Dijkstra’s algorithm does not perform very well on these sparse networks and was evidently slower than all the other algorithms in the test, especially for larger network and path sizes.

Many studies on shortest path algorithms focus on performance in relation to network size, but in this study we have seen that the path size affects the running time just as much, if not more, for the point-to-point shortest path problem. The path size is more directly related to the search space, which for a point-to-point path is usually a lot smaller than the whole network size. But the tests have also shown that network topology has an important impact on the search space, since an algorithm may label more nodes in a short path size than in a large path size if the short path is within a much denser part of the network. The metropolitan area networks in this study are not homogenous; instead they consist of dense city networks connected together by more sparse rural networks, much in the same way as real road networks.

9.2 Test results

The results showed a clear disadvantage to the queue implementation of Dijkstra’s algorithm for the running time in relation to network size. The data was, however, very limited and the path size that the networks was compared with each other for was very short, which makes the network size less important to the result. It is therefore not possible to draw any clear conclusions from this.
In relation to path size, the performance of the algorithms in this study can be divided into three groups which perform similarly. The first and fastest group contains the Breadth-First algorithm, Bidirectional Breadth-First algorithm and Bidirectional Dijkstra’s algorithm. The second group contains the heap versions of Dijkstra’s algorithm, which all perform very similar. And lastly we have the queue implementation of Dijkstra’s algorithm which is by far the slowest of them all.

The use of representative operation counts has been of great value for this study since it allows a deeper analysis of how the different algorithms operate. For example, the poor performance of the queue implementation of Dijkstra’s algorithm is easy to understand when realizing how many queue iterations it does for large problems. It has also been fundamental to the analysis of different heap versions as well as the study of search space reduction for the bidirectional search method.

9.3 Improvements to the simple queue version of Dijkstra’s algorithm

The computational study confirmed what the time complexity predicted: Dijkstra’s algorithm using a queue data structure performs very poorly when the problem size increases. Two different techniques were tested in order to improve its performance: optimizing the management of labelled nodes and reducing the search space.

By using a heap data structure for storing the temporary labelled nodes, the operation of finding the next node to scan can be performed much faster than in a simple queue, with the time complexity of $O(\log(n))$ instead of $O(n)$. The improvement in running time when using a heap instead of a queue was easily seen in the computational study; it was up to seven times faster. Three different heap versions of Dijkstra’s algorithm were tested: 2-heap, 4-heap and a 2-heap which performed duplicate inserts instead of updates. They all performed very similar on all networks, but although the difference is small between them, some conclusions can be drawn.

The duplicate insert method was found to be slower in practice since it requires extra “find-min” operations for duplicated nodes. The 4-heap was faster than the 2-heap when the number of insertions and decrease-key operations where more than 3.38 times as many as the number of delete-min operations, but slower when it was lower than 1.78. In general, the 2-heap was faster than the 4-heap in this study and is therefore recommended for use in similar applications. This is very interesting since this contradicts suggestions by Goldberg (2001) and Weiss (1999) that 4-heaps are often better. Further studies should be carried out on random paths on several networks in order to minimize the possible impact the network topology might have had in this study.
The search space can be reduced to roughly the half, and sometimes even more, by using a bidirectional search when finding a point-to-point path. The actual gain of a bidirectional search depends on the network structure and the specific problem. For some special cases, as when the path size was small and the target node contained a large number of ports, it was even found to increase the search space! In general, however, the running time for the bidirectional version is over two times faster and sometimes even up to 29 times.

The reduction of search space improves the running time for the queue version of Dijkstra’s algorithm the most, but also reduces the running time for the heap version significantly.

9.4 Comparison to previous studies
Both Cherkassky et al. (1993) and Zhan and Noon (1998) found in their studies that no single algorithm was superior on every problem. This is supported by the tests in this thesis; for example the queue implementation on Dijkstra’s algorithm was faster than the heap implementation in some occasions. They also found that the queue implementation of Dijkstra’s algorithm shows very poor performance, except when the number of labelled nodes is very small. That has also been the case in this study where the queue implementation was significantly slower than the other algorithms on the larger problem sizes.

9.5 The new path finding module
A new path finding module for the NETadmin system has been developed, and it contains all the algorithms covered in the practical tests of this study. Besides, it also contains algorithms for finding the highest bandwidth path as well as corresponding algorithms for finding paths between locations. Both a depth-first search and Yen’s algorithm has been implemented for finding the K shortest paths. The module also includes a filter for constraining the search according to the criteria listed in 2.7.

9.6 Recommendations
The results from this study have shown that a significant performance gain is possible by using a bidirectional search method and storing the temporarily labelled nodes in a heap when using Dijkstra’s algorithm for finding a point-to-point shortest path. The recommendation to Netadmin is to use a bidirectional version of Dijkstra’s algorithm and to use a 2-heap instead of a queue. It is also recommended to filter out any unwanted hardware or connection types when creating the network graph since reducing the search space prior to executing the algorithm greatly reduces the running time.
9.7 Improvements to this study

This study has not adjusted the result of the tests for the impact of the network topology. In order to minimize this effect on the results, it is suggested that further studies are carried out on a larger number of networks and that several random paths of each size are selected.

A limitation of this study is that it hasn’t tested the bucket versions of Dijkstra’s algorithm. Previous studies from Cherkassky et al. (1993) and Zhan and Noon (1998) suggests that some of these algorithms perform even better than the algorithms tested here; it is therefore recommended that further computational studies are carried out in order to evaluate their performance on metropolitan area networks. Another improvement to this study would be to include label-correcting algorithms in the evaluation of point-to-point shortest paths. Zhan and Noon (2000) found that Pallottino’s graph growth algorithm implemented with two queues was generally faster in finding a point-to-point path than Dijkstra’s algorithm using approximate buckets.

Further improvements to the path finding module may be done by caching the network structure in the memory between searches and let updates be triggered by changes in the database. This would also open up for the possibility of preprocessing the network and store some distance bounds in memory. Goldberg and Harrelson (2005) use this technique in their algorithm which employs $A^*$ search combined with landmarks and the triangle inequality. They show very promising results from their tests and it may well be worth to consider using their algorithm for Netadmin.
References and further reading

References


Holmberg, Kaj (2002). Kombinatorisk optimering med linjärprogrammering, Matematiska Institutionen, Linköpings tekniska högskola


Further reading

The following papers cover shortest path algorithms that haven’t been included in this study, but which may be of interest to the reader.


Appendix A – Test Results

A.1 Running time for least hop algorithms

![Figure 34: Running time for least-hops algorithms on Network 1.](image-url)
Figure 35: Running time for least-hops algorithms on Network 2.
Figure 36: Running time for least-hops algorithms on Network 3.
A.2 Representative operation counts for least hop algorithms

Figure 37: Scanned ports for least-hops algorithms on Network 1.

<table>
<thead>
<tr>
<th>Path size</th>
<th>BF</th>
<th>BF - BiDirect</th>
<th>Dijkstra</th>
<th>Dijkstra - BiDirect</th>
<th>Dijkstra - 2-Heap Insert</th>
<th>Dijkstra - 2-Heap</th>
<th>Dijkstra - 4-Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>29</td>
<td>6</td>
<td>36</td>
<td>36</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>22</td>
<td>6</td>
<td>94</td>
<td>27</td>
<td>63</td>
<td>59</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>85</td>
<td>27</td>
<td>192</td>
<td>42</td>
<td>199</td>
<td>165</td>
<td>184</td>
</tr>
<tr>
<td>7</td>
<td>113</td>
<td>58</td>
<td>235</td>
<td>42</td>
<td>230</td>
<td>234</td>
<td>229</td>
</tr>
</tbody>
</table>

Figure 38: Labelled ports for least-hops algorithms on Network 1.

<table>
<thead>
<tr>
<th>Path size</th>
<th>BF</th>
<th>BF - BiDirect</th>
<th>Dijkstra</th>
<th>Dijkstra - BiDirect</th>
<th>Dijkstra - 2-Heap Insert</th>
<th>Dijkstra - 2-Heap</th>
<th>Dijkstra - 4-Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>28</td>
<td>13</td>
<td>95</td>
<td>37</td>
<td>68</td>
<td>62</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>93</td>
<td>29</td>
<td>152</td>
<td>106</td>
<td>127</td>
<td>119</td>
<td>107</td>
</tr>
<tr>
<td>5</td>
<td>178</td>
<td>76</td>
<td>221</td>
<td>118</td>
<td>241</td>
<td>203</td>
<td>211</td>
</tr>
<tr>
<td>7</td>
<td>209</td>
<td>136</td>
<td>245</td>
<td>118</td>
<td>261</td>
<td>245</td>
<td>243</td>
</tr>
</tbody>
</table>
Figure 39: Scanned ports for least-hops algorithms on Network 2.

Figure 40: Labelled ports for least-hops algorithms on Network 2.
Figure 41: Scanned ports for least-hops algorithms on Network 3.

Figure 42: Labelled ports for least-hops algorithms on Network 3.
A.3 Running time for shortest path algorithms

The non-updating 2-heap with duplicate inserts was only tested on Network 3.

Figure 43: Shortest path running time on Network 1.

<table>
<thead>
<tr>
<th>Path size</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>0.101</td>
<td>0.926</td>
<td>0.665</td>
<td>0.729</td>
</tr>
<tr>
<td>Dijkstra - 2-Heap</td>
<td>0.106</td>
<td>0.567</td>
<td>0.420</td>
<td>0.428</td>
</tr>
<tr>
<td>Dijkstra - 4-Heap</td>
<td>0.106</td>
<td>0.566</td>
<td>0.401</td>
<td>0.473</td>
</tr>
</tbody>
</table>

Figure 44: Shortest path running time on Network 2.

<table>
<thead>
<tr>
<th>Path size</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>27,766</td>
<td>30,072</td>
<td>5,265</td>
<td>35,436</td>
</tr>
<tr>
<td>Dijkstra - 2-Heap</td>
<td>4,674</td>
<td>5,487</td>
<td>1,804</td>
<td>6,820</td>
</tr>
<tr>
<td>Dijkstra - 4-Heap</td>
<td>4,798</td>
<td>5,662</td>
<td>1,759</td>
<td>6,954</td>
</tr>
</tbody>
</table>
Figure 45: Shortest path running time on Network 3.

A.4 Representative operation counts for shortest path algorithms

The number of scanned and labelled ports is the same for all shortest path algorithms, except for the insert-only 2-heap version of Dijkstra’s algorithm, which is not included here.

Figure 46: Number of scanned ports for shortest path algorithms.
A.5 Heap operations

The heap operations have been counted for Dijkstra’s algorithm on the shortest path problems.

![Figure 48: Number of heap operations on Network 1.](image-url)
Figure 49: Number of heap operations on Network 2.

<table>
<thead>
<tr>
<th>Path size</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanned ports (delete-min)</td>
<td>916</td>
<td>1179</td>
<td>295</td>
<td>1614</td>
</tr>
<tr>
<td>Labelled ports (inserts)</td>
<td>1403</td>
<td>1525</td>
<td>853</td>
<td>1714</td>
</tr>
<tr>
<td>Port distance updates (decrease-key)</td>
<td>189</td>
<td>199</td>
<td>147</td>
<td>238</td>
</tr>
</tbody>
</table>

Figure 50: Number of heap operations on Network 3.

<table>
<thead>
<tr>
<th>Path size</th>
<th>2</th>
<th>5</th>
<th>8</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scanned ports (delete-min)</td>
<td>9</td>
<td>115</td>
<td>1719</td>
<td>1762</td>
</tr>
<tr>
<td>Labelled ports (inserts)</td>
<td>28</td>
<td>258</td>
<td>2006</td>
<td>2019</td>
</tr>
<tr>
<td>Port distance updates (decrease-key)</td>
<td>4</td>
<td>131</td>
<td>988</td>
<td>1054</td>
</tr>
</tbody>
</table>
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.
For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© David Johansson