Combining unobtainable shortest path graphs for OSPF

Department of Mathematics, Linköping University

Erik Haraldsson

LITH-MAT-EX--2008/07--SE

Examensarbete: 45 hp

Level: D

Examiner: Kaj Holmberg
Department of Mathematics, Linköping University

Supervisor: Kaj Holmberg
Department of Mathematics, Linköping University

Linköping: May 27, 2008
Abstract

The well-known Dijkstra's algorithm uses weights to determine the shortest path. The focus here is instead on the opposite problem, does there exist weights for a certain set of shortest paths? OSPF (Open Shortest Path First) is one of several possible protocols that determines how routers will send data in a network like the internet. Network operators would however like to have some control of how the traffic is routed, and being able to determine the weights, which would lead to the desired shortest paths to be used, would be a help in this task.

The first part of this thesis is a mathematical explanation of the problem with a lot of examples to make it easier to understand. The focus here is on trying to combine several routing patterns into one, so that the result will be fewer, but more fully spanned, routing patterns, and it can e.g. be shown that there can't exist a common set of weights if two routing patterns can't be combined.

The second part is a program that can be used to make several tests and changes to a set of routing patterns. It has a polynomial implementation of a function that can combine routing patterns. The examples that I used to combine routing patterns, showed that this will increase the likelihood of finding and significantly speed up the computation of a “valid cycle”.

Keywords: Internet Protocol, OSPF, combining SP-graphs, valid cycle, subpath consistency
Acknowledgements

I would first and foremost like to thank my supervisor and examiner Kaj Holmberg for all support, discussion and help during this thesis.

I would also like to thank Marianne Kratz.

Finally, a big thanks to the people close to me.
# Table of Contents

Chapter 1: Introduction

1.1 Background ............................................................................................................. 1
1.2 Problem/Purpose ..................................................................................................... 1
1.3 Outline .................................................................................................................... 1

Chapter 2: Theoretical Description

2.1 Definitions .............................................................................................................. 3
2.2 Mathematical model ............................................................................................... 4
2.3 Valid cycle ................................................................................................................ 6
2.4 Subpath consistency ............................................................................................... 8
2.5 Restrictions to combining SP-graphs ..................................................................... 10
2.6 Summary .................................................................................................................. 15

Chapter 3: Mechanisms of the Program

3.1 Introduction ........................................................................................................... 17
3.2 Running Times ....................................................................................................... 17
3.3 The structure of my program .................................................................................. 18
3.4 Storage .................................................................................................................... 20
3.5 Startup .................................................................................................................... 21
3.6 The use of DFS ....................................................................................................... 22
3.7 How to divide SP-graphs ...................................................................................... 26
3.8 How to check in order to combine SP-graphs ....................................................... 27
3.9 How to combine SP-graphs ................................................................................... 28
3.10 Remove extra SP-graphs ..................................................................................... 30
3.11 Check for subpath consistency ........................................................................... 31
3.12 Changing SP-graphs ........................................................................................... 33
3.13 Check valid cycles ............................................................................................... 35
3.14 Summary ............................................................................................................... 35
3.15 List of complexities ............................................................................................. 36

Chapter 4: Mechanisms of the Program

4.1 Menu ....................................................................................................................... 37
4.2 Arguments ............................................................................................................... 38
4.3 Information .............................................................................................................. 39

Chapter 5: Computational tests and results

5.1 Generating modified SP-graphs .......................................................................... 41
5.2 Information about the examples used ................................................................... 41
5.3 Introduction to running the program .................................................................... 42
5.4 Results from running the program ....................................................................... 43

Erik Haraldsson, 2008
5.5 About Valid Cycles.................................................................46
5.6 Conclusions........................................................................47

Appendix A: The Source Code..................................................53
    A.1 graftest.h.......................................................................53
    A.2 control.cpp.................................................................54
    A.3 text.cpp.......................................................................58
    A.4 startup.cpp.................................................................59
    A.5 addOrSplit.cpp.............................................................60
    A.6 DFS.cpp.......................................................................65
    A.7 algorithms.cpp.............................................................68
    A.8 arcFunc.cpp..................................................................73
    A.9 pathFunc.cpp...............................................................74
    A.10 printOrEnd.cpp............................................................77

Appendix B: More Computational Results..................................79
    B.1 The Commands used.....................................................79
    B.2 More data for the RAND examples.................................79

Appendix C: Information about valid cycles...............................81
    C.1 Examples that has a valid cycle and are also subpath consistent........................................................................81
    C.2 Examples that lacks a valid cycle....................................81
Index Of Figures

Figure 1: A first example of an SP-graph (here an out-graph from node 3).................................4
Figure 2: An example of Dijkstra’s algorithm........................................................................4
Figure 3: A valid cycle (that is subpath inconsistent).................................................................8
Figure 4: Two SP-graphs with one set of paths each, that are subpath consistent and have a valid cycle..................................................................................................................9
Figure 5: A subpath consistent example with a valid cycle..........................................................9
Figure 6: Another subpath consistent example with a valid cycle.............................................10
Figure 7: An example of how combining SP-graphs would create incorrect information.............11
Figure 8: Two SP-graphs that can't be combined......................................................................12
Figure 9: Two SP-graphs that can be combined.........................................................................12
Figure 10: The resulting combined SP-graph............................................................................12
Figure 11: Another example of two combinable SP-graphs..........................................................13
Figure 12: Two SP-graphs........................................................................................................13
Figure 13: The parts that can be combined................................................................................13
Figure 14: Two SP-graphs to show how the program get the information in the graphs............22
Figure 15: SP-graph used to illustrate DFS...............................................................................25
Figure 16: A SP-graph with two sets of paths............................................................................26
Figure 17: The resulting two divided SP-graphs .......................................................................26
Figure 18: Illustrating the steps to combine SP-graphs...............................................................28
Figure 19: An example when one of the two SP-graphs doesn't add to the total amount of information........................................................................................................................30
Figure 20: One SP-graph is a part of the other, but has extra information..................................30
Figure 21: Two SP-graphs that are subpath consistent...............................................................33
Figure 22: Illustration of the four cases for changing an SP-graph..........................................34
Figure 23: Where changing the SP-graph causes a cycle to appear..........................................35
Definitions

Mathematical Model, General:
G = The graph
N = All nodes in the graph
A = All arcs in the graph
o_k = The origin of an arc or path
d_k = The destination of an arc or path
P_k = The shortest paths from o_k to d_k
Q_k = The paths from o_k to d_k that aren't a part of P_k
p, q, r = A specific path
A_l = SP-graph (Shortest-Paths Graph) l (one or several P_k)
w_{ij} = The weight from node i to node j, variable in the WFP
π_l = node potential for node i and SP-graph l.
V_l = All node-pairs (s,t), where s and t are both connected to A_l
T_{st} = Those paths from s to t, that are completely outside of A_l (uses no arc or node that is a part of A_l, except nodes s and t)
γ_l = Flow of commodity l in arc (i,j), dual variables for the WFP

Mathematical Model, Cycles:
C = A cycle, normally created from two SP-graphs
F = The arcs that are used in the normal direction, when creating a cycle
B = Arcs used backwards/with negative flow in a cycle.
S(s,e) = A subpath from s to e in SP-graph A_i
γ(B) = All arcs where both the start- and endnode belongs to set B
N(C) = Nodes that are connected by the arcs in set C
δ+(B) = All arcs that enters node set B (the endnode of the arc belongs to B but not the startnode)
δ-(B) = All arcs that leaves node set B (only the startnode of the arc belongs to B)

Complexity:
m = Total number of incoming SP-graphs
m(out) = Total number of resulting SP-graphs
|A_i| = The highest number of arcs in a SP-graph
|N| = The total number of nodes that is a part of a problem
#paths = The number of paths that a random SP-graph contains
#end (#start) = The number of endnodes (startnodes) for the paths in a random SP-graph
#CN = The number of nodes connected to a SP-graph
Chapter 1: Introduction

1.1 Background

The common thing for all protocols and programs that use the internet is the ability to send data from one computer to another. The data is divided into small packages and these packages will be controlled and combined once they reach their destination. Each package has a specific delivery address, that tells where the intended receiver is. A machine called a router is used at every junction to guide the package to the correct address. The routers need to send the packages in a manner that will make sure that they arrive fast, check if a specific line to another router is working, and divide traffic so that there aren't any traffic jams.

To do this there are several different routing protocols that a router can use. I will here assume that the OSPF (Open Shortest Path First) protocol is used, with the general ECM (Equal Cost Multipath) condition. OSPF uses Dijkstra's algorithm in order to calculate the shortest paths, and each router then saves a list of where a specific package should be sent in some kind of database. A new updated list must be created each time a specific link in the network is changed (a new link is added or an old link breaks down) and the updated information is then sent to each router. This list will, in a big network like the internet, only contain information of where to send the data in some neighbourhood of the router, in order to make the list relatively small and to update the list fewer times. Each link between two routers needs a designated weight in order for Dijkstra's method to do an optimization of the best route to send the package. The ECM condition says that there can be several valid options to get from one router to another (i.e. Dijkstra's method can find several paths with the same lowest sum of weights) and that the traffic, in these cases, will be divided evenly between all shortest routes.

1.2 Problem/Purpose

The problem of finding the shortest paths when we already have the corresponding weights for each arc is an easy problem. A harder problem is finding the weight if we know which paths should be the shortest. One difference between the two problems is that it is always possible to find a shortest path when the weights are known, but that the weight finding problem sometimes lacks a valid solution. If a set of weights doesn't make all pre-determined paths the shortest, then the weights aren't a valid solution. My student thesis, and especially the theory chapter, is primarily based on Peter Broström's dissertation [2] and deals with the weight-finding problem. The theory chapter was written with the intention of explaining the main parts of the problem in an easier way, with helpful examples and without any lengthy sidetracks.

There is an important structure called a 1-valid cycle. If it is possible to find a 1-valid cycle in a problem then the problem can't have a set of valid weights. There are also cases that has neither a 1-valid cycle nor a valid set of weights. My main purpose here was to create an algorithm that would combine several smaller SP-graphs to one larger SP-graph and observe how this affected the speed of the algorithm that looks for 1-valid cycles and the number of problems where a 1-valid cycle was found. The theory for combining SP-graphs was stated in [2], but there was no definite was of how to interpret it. Another part of my program can modify some of the shortest paths, in order to find more examples that only has 1-valid cycles after several paths has been combined.

1.3 Outline

I will in the second chapter first introduce the mathematical background in general, and then some interesting characteristics, that I will discuss in more detail and use in my program.
The third chapter is about how the program is built and which choices I made.

Then there is a brief fourth chapter that describes how to use the program.

I will finally discuss the results from running the program on some examples, and what conclusions all this leads to, in chapter five.

The complete code to my program is in Appendix A.

Appendix B has some additional results and appendix C notes which of the examples has a 1-valid cycle and other conditions.
Chapter 2: Theoretical Description

2.1 Definitions

I will here need the notion of a directed graph $G = (N, A)$ in order to effectively be able to describe a network. First $N$ is the nodes, here naturally representing the routers, and second $A$ is the arcs between the nodes. As you should remember an arc is directed, so that it has one node that it starts in and another that it ends in, and it is only possible to travel from the startnode to the endnode. Here the arcs are all the direct links between routers. One advantage with (directed) arcs is that it allows for different paths in different directions. However, this model only allows for one arc from router A to router B, so it is necessary to either introduce fake routers if there are several direct links between A and B or to see all direct routes as one. Two other terms that I will use are in-degree and out-degree for a specific node $i$. The first one defines the number of arcs that has their endpoint in node $i$, while out-degree is the number of arcs that starts in node $i$. I will use the term $(a, b)$ or $<a, b>$ to note an arc from $a$ to $b$.

Now I need to define a path. A path is one or several arcs that has a startnode $o$ (origin) and an endnode $d$ (destination) and the arcs are chosen in such a way that node $o$ has a in-degree of 0 and an out-degree of 1, node $d$ has a in-degree of 1 and a out-degree of 0 and the rest of the nodes that the arcs connects has an in-degree of 1 and an out-degree of 1, relative to the arcs in the path. An easier way to say this is that a path is a specific way to get from node $o$ to node $d$. A set of paths is all possible paths (or some part of all possible paths) from node $o$ to node $d$. A special sort of paths are those called Shortest Path. As the name suggest this is the paths that should be the shortest when the routers use Dijkstra's method on the defined weights. Here ECM (Equal Cost Multipath) comes into the picture. If a router uses ECM, then it means that we are allowed to have a set of shortest paths instead of just one shortest path between node $o_k$ to node $d_k$. We use $P_k$ to note the set of desired shortest paths from node $o_k$ to node $d_k$ where $k$ is one of all possible origin-destination pairs, $k=1,...,K$. The alternative to ECM is to only allow one path to be cheapest. ECM will be a more general case than when we have a one-path-rule, since it is still possible for a set of shortest paths to consist of only one path. So anything that is stated in this paper is also true when the one-path-rule is applied, but some of the code can be made easier and/or faster in that case.

An SP-graph (Shortest Path-graph), $A_l$, $l=1,...,m$, is a subset of the arcs that describes the desired shortest paths between a number of nodes ($A_l \subseteq A$). When it is as simple as possible an SP-graph is just a set of shortest paths from one node to another. However several sets of shortest paths can also be combined into a larger SP-graph. The most common combined structures are out-graphs and in-graphs. A out-graph are several sets of shortest paths that all starts in the same node (one set from node $o$ to node $d_1$ and another set from node $o$ to node $d_2$...) while an in-graph is the opposite, several sets of shortest paths from different startnodes that all has the same endnode. In the case when all sets contains only one path each, are they also known as out-tree and in-tree.

I will use the terms origin and startnode as interchangeable when I talk about SP-graphs and simply mean any node that has an in-degree of 0 and an out-degree of at least 1 and similarly for destination and endnode. I will however also use the term startnode when I talk about an arc and then mean the node where that specific arc starts, but I hope that the context of the term should make the meaning clear (the same naturally also applies to the term endnode). In most of both the theory and programming are there two “opposite” situations where something is true. One example of this is when you get to the part about combining SP-graphs, where they can then either be combined to out-graphs or in-graphs, but the theory is (practically) the same in both cases. I will then write out-graphs (in-graphs) the first time, and then leave out the opposite word in the parentheses in the rest of the discussion to make it easier to read, unless it is something that isn't obvious.
The picture above is an SP-graph, where the numbered rings represent nodes and the lines represent arcs and the arrows pointing out the endnode for each arc. It has two sets of paths, both sets starts in node 3 and one set ends in node 1, while the other one ends in node 8, which makes this an out-graph. Also the first set of paths from node 3 to node 1 is just one path, while the set from node 3 to node 8 contains two separate paths, one through node 5 and one through node 6. Node 3 has an in-degree of 0 and a out-degree of 2, since two arcs leave node 3. but no arc enters, and node 4 has a in-degree of 1 and a out-degree of 2, as one arc enters and two leaves. The in- and out-degree for the rest of the nodes can then be found similarly, by counting the number of incoming and outgoing arcs.

### 2.2 Mathematical model

Here is an example of how Dijkstra's algorithm works:

![Diagram of Dijkstra's algorithm]

It is used here to find the shortest path from node 1 to node 7, and the number above (or to the left of) each arc is the weight for that arc. The unbroken lines are the arcs that the algorithm finds as shortest and the dotted ones are the more expensive ones. In the parentheses above each node, is the total minimum weight that is needed to get to that node, and the previous node in the shortest path. Dijkstra's algorithm will always give the shortest path to each node, and will thus create an out-graph (here even out-tree), or alternatively an in-graph if it is set to find all the shortest paths to a specific node. It starts in node 1 and marks that the shortest path to node 3 this far is (1,1), to node 4 (4,1) and (3,1) to node 2. Then node 2 is checked, and the shortest path to node 6 is found, while the path to node 4 is more expensive than the one already there. When node 3 is searched, the path to node 5 is found, and so is a new shorter path to node 4, that replaces the old direct one. Node 4 gives the shortest path to node 7, and a more expensive path to node 5. Node 6 gives a more expensive path to node 5 and node 5 gives a more expensive path to node 7 and Dijkstra's algorithm is thus finished.

The opposite problem, that I will focus on here, is when we have the shortest paths, here one from node 1 to node 7, one from 1 to 5 and one path from 1 to 6. Each path can be in its own SP-graph, or they can all be part of the same SP-graph. An SP-graph is just a container for one or more sets of paths, and each path above is also a set of paths, since there is only one shortest path from node 1 to each of the endnodes.

A company can of course just decide the weights at random, but then most of the packages could end up being sent on slow routes or detours. A better alternative would be to just give low weights to the best connections and higher weights to slower connections, but it would still not give as much control of the routes used as deciding the weights based on these predefined preferred shortest
paths.

Now assume that the arc from node 3 to 4 has became inoperable for some reason. In that case the path from node 1 to node 7 will consist of the arcs (1,4) and (4,7), since it is the second shortest path. One important part of the internet is that the data should always be able to get from one node to any other node, even if the preferred link is broken, which can be accomplished by using weights.

We now have the necessary tools to start describe the problem of finding feasible (and minimum) weights. The easiest way to formulate this problem can be seen below:

$$
\min \sum_{(i,j) \in A} w_{ij}
$$

s.t. \( \sum_{(i,j) \in q} w_{ij} - \sum_{(i,j) \in p} w_{ij} \geq 1 \forall p \in P_k, \forall q \in Q_k, \forall k \)

$$
\sum_{(i,j) \in r} w_{ij} - \sum_{(i,j) \in p} w_{ij} = 0 \forall p \in P_k, \forall r \in P_k, \forall k
$$

where \( w_{ij} \) is the weight for the connection from node \( i \) to node \( j \), \( P_k \) is the desired set of shortest paths between node \( o_k \) and node \( d_k \). \( Q_k \) is the set of paths between node \( o_k \) and node \( d_k \) that should be more expensive than the paths in \( P_k \), and \( p,q,r \) each represents a single path.

Let's look back at the last example. There \( p=\{(1,3)(3,4)(4,7)\} \) for the path from node 1 to node 7 and for example \( q=\{(1,4)(4,7)\} \) would be a more expensive path, that is it would be a part of \( Q \). The total weight for the arcs in \( p \) is here 7 and the total weight for \( q \) is 8. And thus the first condition is fulfilled for these two paths.

This formulation does describe the problem very clear and concise. It looks for weights that are feasible, and even gives the smallest possible sum of weights. All the paths that should be shortest have the same total sum of weight (the second condition) and the rest of the paths between any origin and destination should be more expensive than the paths that has in advance been marked as the cheapest (the first condition). There is also a third set of conditions that makes sure that the weights either are positive integers or can be made integers by some multiplier, see [3]. If this problem has a valid solution, then we say that it has compatible weights. This formulation of the weight-finding problem does however have the problem that \( Q_k \) can become very big, even in a small problem, and a related problem is that it will be very hard to list all the possible more expensive paths from \( o_k \) to \( d_k \) in advance. [5] proves that this problem can instead be stated as:

$$
\min \sum_{(i,j) \in A} w_{ij}
$$

s.t. \( w_{ij} + \pi^l_j - \pi^l_j = 0 \forall (i,j) \in A_l, l=1,...,m \)

$$
\sum_{(i,j) \in q} w_{ij} + \pi^l_i - \pi^l_i \geq 1 \forall q \in T^l_{st}, \forall (s,t) \in V, l=1,...,m
$$

$$
w_{ij} \geq 1 \forall (i,j) \in A
$$

\( A_l \) is here an SP-graph, that contains one or several sets of paths, and \( \pi^l_i \) is the node potential for node \( i \) and SP-graph \( l \). If a node either has a in-degree or an out-degree that is higher than zero for the SP-graph \( A_l \), that is at least one arc in \( A_l \) either starts or ends in the node, then the node is connected to \( A_l \). \( V_l \) contains of all the node-pairs (s,t), such that s and t are both connected to \( A_l \). \( T^l_{st} \) is here all the paths from s to t, where no arc is a part of \( A_l \) and no node in the path is connected to \( A_l \) (except obviously s and t). Another way to explain it is that if a path \( q \) starts in s and then follows an arc to a node that is not a part of \( A_l \), then to another node outside \( A_l \) and going on like that until the path reaches t, then \( q \) is one of the paths in \( T^l_{st} \).

We say that \( A_l \) is fully spanned if it is connected to all nodes. In that case the problem above can be...
simplified to the following form, as showed in [3]:

$$\min \sum_{(i, j) \in A} w_{ij}$$

s.t. \(w_{ij} + \pi^l_i - \pi^l_j = 0 \forall (i, j) \in A, l = 1, ..., m\)

$$w_{ij} + \pi^l_i - \pi^l_j \geq 1 \forall (i, j) \notin A, l = 1, ..., m$$

If all three paths are part of the same SP-graph in the example above (figure 2), then the SP-graph is fully spanned and we can look at what it means for the latest formulation of the problem. First if the node price for node 1 is zero, and the node price is equal to the lowest cost to get there from node 1 for the rest of the nodes, then the weights will unsurprisingly be a valid solution to the problem above.

If only the path from node 1 to node 7 had been in the (first) SP-graph, then there would have been other paths in \(T_{st}\). One example would then be when \(s=4\) and \(t=7\), then \(q=\{(4, 5), (5, 7)\}\) would be a part of \(T_{47}\) (it would in fact be the only path), since node 5 is in this case not connected to the SP-graph. The condition would look like \(3+4+3-7 \geq 1\), so it would be true for the weights given.

The optimization problems above are all variations of what is called the Weight Finding Problem (WFP). It is possible to learn more about the problem by looking at the dual problem to the last version (where all SP-graphs had to been fully spanned). By reducing the number of variables and re-writing the objective function (the details are in [3]), the dual problem get the following form:

$$\min \sum_{l=1}^{m} \sum_{(i, j) \in A} y^l_{ij}$$

s.t. \(\sum_{l=1}^{m} y^l_{ij} \leq 1 \forall (i, j) \in A\)

$$\sum_{j: (i, j) \in A} y^l_{ij} - \sum_{j: (j, i) \in A} y^l_{ji} = 0 \forall i \in N, l = 1, ..., m$$

$$y^l_{ij} \geq 0 \forall (i, j) \notin A, l = 1, ..., m$$

This is a variation of a problem known as the multicommodity network flow problem (MNFP). The only difference from the standard formulation is that some variables (the ones corresponding to a shortest path in SP-graph \(A_l\)) are allowed to have unlimited negative values.

It is shown in [3] that WFP has a feasible integer solution if and only if MNFP has a limited optimal solution. In other words if we can find a flow for MNFP that gives an unlimited solution, then this is enough to show that it is impossible to find any weights that satisfies all desired shortest paths. Let's now define a flow for MNFP with unlimited solution as a valid cycle.

### 2.3 Valid cycle

It is now time to take a closer look at the MNFP. The easiest thing to spot is that the second constrains means that if a certain amount of a commodity gets into a node, then the same amount of that commodity will also leave the node, and this is true for all nodes and all commodities. This means that a commodity must be sent in cycles, from node a to some other part and then back to node a, if it is sent at all. The first conditions sets a upper limit on the total sum for all commodities, that can be sent on a specific arc. The use of \(A_l\) in the third conditions and the objective function means that each commodity corresponds to a specific SP-graph. The interpretation here is that the flow for a specific commodity must be non-negative, unless the arc is a part of the corresponding network.
SP-graph, and that only the flow that is on an arc in the corresponding SP-graph will be counted towards minimizing the objective function. The goal is to get the objective function to be \(-\infty\) and this can be done by making the flow on some arcs for specific commodities (where the arc is a part of the corresponding SP-graph) to be \(-\infty\). A negative flow from a to b can be seen as a positive flow from b to a.

Can this be done by just one commodity? Not really, since this would mean that there would have to be a cycle in the corresponding SP-graph (since only arcs in the SP-graph can have an unlimited negative flow and this unlimited negative flow must be sent in a cycle), and this would correspond with that we would want to send information from router a and then get it back to router a and only after that send it towards the destination, which would make no sense. I will from now on assume that no SP-graph has a cycle of its own, because that would be a very unrealistic case, and is just not interesting to study.

Let's now consider the case with two commodities. The first conditions will always be fulfilled by sending equal amounts of both commodities in a path that is a cycle (a path that both starts and ends in the same node), but in opposite directions (one commodity is sent from node a to node b while the other is sent from b to a alternatively that a negative flow is sent from a to b). This will also guarantee that the second condition is fulfilled. Condition 3 means that if an arc has a negative flow of a specific commodity, then it must also be a part of the corresponding SP-graph. The result is that all arcs in a cycle where one commodity has a negative flow must be a part of one SP-graph, but the second commodity has a negative flow where the first one has a positive and reverse, which leads to that all positive arcs in a cycle belongs to one SP-graph, while all negative arcs belong to the other SP-graph.

This kind of cycle will be called 1-feasible, and I will use the notation \(C = F \cup B\), where \(F\) is the arcs that are used in the normal direction or alternatively the arcs with a positive flow, and \(B\) the arcs that is used in the opposite direction, which is equal to say that they have a negative flow. The second cycle is found by making the arcs in \(F\) to \(B\) and the arcs in \(B\) becomes \(F\). Finding a 1-feasible cycle means that it will equal a valid, but not necessarily optimal, solution to MNFP. If it is also optimal, then it is called 1-valid, and is defined as:

A cycle, \(C = F \cup B\), is called 1-valid if there are two SP-graphs, \(A_i\) and \(A_j\), such that \(F \subseteq A_i\) and \(B \subseteq A_j\), while it is also true that \(B \not\subseteq A_i\) and/or \(F \not\subseteq A_j\).

The latter part of the definition is also known as an eligible arc and means that there is at least one arc in the cycle that is only part of one of the two SP-graphs. There is of course a reason to demand an eligible arc, and that is to make it possible for the objective function to be anything but zero. The objective function counts the flow in the arcs that is a part of the corresponding SP-graph, and the total flow for each arc is 0. This means that if both the positive and the negative flow for each arc will count, then the objective function must be zero. The negative flow must always be a part of the SP-graph, per the third condition, so at least one of the arcs with positive flow must be outside the corresponding SP-graph in order to not count, and thus making the objective smaller than zero. No part of the discussion above has specified how big the flow of the cycles are, so it will be possible for one commodity to have a \(+\infty\) flow and the other commodity to have a \(-\infty\) flow in the same arc. If there is an eligible arc, then its flow will be the objective, thus making the objective \(-\infty\). This dual function assumes that all SP-graphs are spanning, but it is showed in [5] that a 1-valid cycle as defined above means that there are no set of compatible weights, even if the SP-graphs are not fully spanned.

There are valid cycles that uses more than two commodities, and [4] defines and describes several other sorts of valid cycles that uses at least three commodities, called 2-valid, 3-valid, 4-valid and 5-valid cycles, but there isn't any practical implementations to find these kinds of cycles yet.
Here is an example with two SP-graphs in the left part above and the resulting 1-valid cycle in the right part. The arcs with a complete line belong to $A_1$ (the first SP-graph) and the ones with a dotted line belongs to $A_2$ in this and the rest of the examples. It is easy to see that the two SP-graphs have different paths between node 3 and node 6, which means that we use one subpath to get from node 3 to node 6 and the other one from node 6 back to node 3. This sort of simple cycle is called subpath inconsistent (see more about this later). The cycle has four arcs and all four of them are eligible, in fact the two SP-graphs does not have a single arc in common. The cycle in the right part has (3,4) and (4,6) as F and (3,5) and (5,6) as B, so that the cycle uses the opposites to those arcs, (6,5) and (5,3), which equals to a positive flow on arcs (3,4) and (4,6) and a negative flow on (3,5) and (5,6). A second cycle can be constructed easily by reversing the cycle and thus have a negative flow on (3,4) and (4,6) and a positive flow on (3,5) and (5,6).

### 2.4 Subpath consistency

This section deals with a term called subpath consistency and the opposite term subpath inconsistency. If two SP-graphs are compared at a time, and each combination of SP-graph-pairs are subpath consistent, then it is equal to say that all SP-graphs are subpath consistent. A subpath from a to b is a part of an SP-graph, and more specifically the part of each path in the SP-graph that starts in node a and ends in node b, that is, all the possible shortest routes to get from node a to node b. If there are no shortest arcs that can be used to create a subpath from a to b, then there does not exist a subpath from a to b. If there exists a subpath from a to b in an SP-graph, then there will not exist a subpath from b to a, and reverse (unless there is a cycle in just one SP-graph, something that I have dismissed earlier). There is for example a subpath from node 3 to node 4 in the first SP-graph in figure 3 above, but no subpath from 4 to 3. It is also possible that there is neither a subpath from a to b or from b to a, and an example of this would be to try and find a subpath from node 5 to node 6 (or from 6 to 5) in the very first figure. Two SP-graphs are subpath inconsistent if there exists at least one pair of nodes a and b, such that both the first and second SP-graph has an existing subpath from a to b, where the arcs in the subpath for the first SP-graph are different from the arcs in the second subpath. A clear example of this can be found in Figure 3 above, where the two SP-graphs has different subpaths from node 3 to node 6. The first SP-graph has a path that contains (3,4)(4,6), while the second SP-graph has the subpath (3,5)(5,6) and there is thus two different routes to get from node 3 to node 6, but each SP-graph only contains one of them. It is also easy to see that it would be impossible to get one set of weights such that the weights for the first subpath should be cheaper than the second and at the same time the second being cheaper than the first. The opposite of this is that the subpaths are equal for all node-pairs where both SP-graphs has a valid subpath and this is then called subpath consistency. A more condense definition would be:

Start with two SP-graphs, $A_i$ and $A_j$, and call all the subpaths from s to e, $S_i(s,e)$ and $S_j(s,e)$. Then $A_i$ and $A_j$ are called subpath consistent if $S_i(s,e) = S_j(s,e)$ for all $s \in N$ and $e \in N$ (where N are all the nodes in the graph), where both $S_i(s,e)$ and $S_j(s,e)$ exists.

**Theorem 1:** If two SP-graphs are subpath inconsistent then they will have a 1-valid cycle.

**Proof:** Let's call the two subpath inconsistent SP-graphs $A_i$ and $A_j$, then there is at least one pair of nodes with different subpaths, and to make it easier, let's assume that these different subpaths are from node a to node b. This means that one of the subpaths from a to b in $A_i$ contains contains at
least one arc that isn't part of any subpath from a to b in SP-graph $A_i$ (or if not, switch $A_i$ and $A_j$ and then it will be true), that is $A_i$ has at least one eligible arc (and one or more eligible arc(s) is necessary for a cycle to be valid). The eligible arc could either be a “short cut”, i.e. an arc that skips one or more nodes that any possible common subpaths connects or two or more arcs that connects to a new node, but there will always be at least one arc that only one SP-graph uses from a to b, in order for them to be subpath inconsistent. Now to create a cycle use the subpath from a to b in $A_i$ that has at least one eligible arc (or choose anyone of several paths with eligible arc) and then choose a random subpath that is a part of $A_j$ to finish the cycle from b to a. The two subpaths chosen can't be the same, since the one in $A_i$ had an eligible arc, so one subpath will contain the arcs in F and the other the arcs in B, and it must be a 1-valid cycle. QED.

Theorem 1 above means that subpath consistency is a weaker demand than having no 1-valid cycles and figure 4, 5 and 6 below will show three examples of when all SP-graphs are subpath consistent, but there are still valid cycles. This means that there are three different cases that are of interest, first when some of the SP-graphs are subpath inconsistent and they will thus always have 1-valid cycles, as shown above. The second case is when all the SP-graphs are subpath consistent, and they still has a 1-valid cycle (as in the examples below), and the third case is when there is no 1-valid cycle. The third case is when no 1-valid cycle can be found. This could be for two different reasons, the first and most common is that they have compatible weights, and the second reason is that they have some sort of valid cycles with three or more commodities. I haven't tried to separate those two reasons, but instead just look at them as one case. The valid cycles that are found in the first case, are simple to interpret, since they basically says that path a should be cheaper than path b and path b should be cheaper than path a. Most of the examples that I was given to test my program with was also of the first case, especially as the examples grew larger. The second case is more interesting, and has also been harder to find in the examples. Figure 4 right below shows an example of a valid cycle where the two SP-graphs are subpath consistent and only has one set of paths each, but the examples in the second case that I tested the program on only had valid cycles after several SP-graphs had been combined.

![Figure 4: Two SP-graphs with one set of paths each, that are subpath consistent and have a valid cycle](image)

This example consists of two SP-graphs that only have one set of paths each. Both SP-graphs are subpath consistent, since they only have four common nodes, 2, 3, 4 and 5. The first SP-graph has a subpath from 2 to 3 but the second doesn't (since it is not possible to get from node 2 to node 3 with the arcs in the second SP-graph) and only the second SP-graph has a subpath from 2 to 5. The second has a subpath from 4 to 3, but the first doesn't, and only the first SP-graph has a subpath from 4 to 5. There is however a 1-valid cycle, that consists of the arcs (2,3) (4,3) (4,5) and (2,5).

![Figure 5: A subpath consistent example with a valid cycle](image)
In this example there is a 1-valid cycle with four arcs, two from each SP-graph and with two eligible arcs. The interesting thing about this cycle is that each of the four arcs belongs to a different set of paths, (7,5) belongs to the path from node 3 to node 5 while (1,2) belongs to the set of paths from node 3 to node 1, both from the first SP-graph, while (1,7) in its reversed shape comes from the set of paths from node 4 to node 1 and finally (5,2) comes from the path from node 4 to node 5 (and has then been reversed to form a cycle) and both the latter sets of paths are from the second SP-graph. This means that this cycle can only be found when several different paths are combined in one SP-graph. It is also easy to check that these two SP-graphs are subpath consistent, since both SP-graphs only has subpaths for the node-pairs 2-6, 2-1 and 7-1, and the subpaths are equal in these three cases.

Why does this problem have a valid cycle? The arcs (3,7), (3,2), (4,7) and (4,2) are all used, so we can expect them to have a relatively low node weight, and the arcs (7,1) and (2,1) are used by both SP-graphs so they would also have a low weight. Then there are the arcs (7,5) and (2,5), that is used in one SP-graph each, which leads to that both these arcs must be relative cheap, at the same time that the arcs from each start node to both node 2 and node 5 is cheap, so it would seem more natural to use one of (7,5) and (2,5) for both SP-graphs. There seems to be some kind of conflict here. We can also see that if (7,1) wouldn’t have been used by the second SP-graph, then the weight for arc (4,7) could rise and thus explain why (7,5) wasn't used and analogous for the arc (3,2) in the first SP-graph.

Here are two SP-graphs, one that has node 3 as origin and another with node 1 as origin, and both are out-graphs. The right part of the figure shows the valid cycle that were found. Each one of the six arcs comes from a different set of paths. We can also see that all arcs in the cycle except (10,8) are eligible. The main reason for including this example is to show that cycles can be hard to find and grasp, by just looking at the figure, even in a relatively small problem.

2.5 Restrictions to combining SP-graphs

If there are two SP-graphs, each with one set of paths, then combining those SP-graphs means that we would instead have one big SP-graph with two sets of paths in it. This means that the same information will now be stored in a smaller number of SP-graphs than before and that each SP-graph will have a larger number of arcs than before. Both of these things are good when looking for valid cycles, since there will be fewer combinations of SP-graphs to look through and more arcs makes it easier to find valid cycles.

The general idea when combining SP-graphs is that the resulting SP-graph should give us the same information about the structure of the shortest paths, as each set of paths does. Not more and not less.

First, consider a path from a to b and a path from b to c. If we combine these two paths, then we get a longer path from a to c through b. But this will clearly give us more information than the separate
paths, because there could very well be another path from a to c that we want to be shorter, and since this would lead to two different paths from a to c, it would now be possible to find a valid cycle, even though there may not have been one amongst the original paths (or if both paths were stored in the same SP-graph, then the program would think that both should be equally cheap, which wasn't the original intent and the combination has by that altered the data).

A more general way to show how adding paths can create new origin-destination pairs are when we have one path, $P_1$, from $o_1$ to $d_1$ and another path, $P_2$, from $o_2$ to $d_2$ (where $o_2 \neq o_1$ and $d_1 \neq d_2$). If the two paths have one or more node in common, the combined SP-graph will (unless several specific criteria are fulfilled, see further down) create new origin-destination pairs, $o_1$ to $d_2$ and $o_2$ to $d_1$, when there may be another path from $o_1$ to $d_1$ that should be shorter, which means that this will also create more information than was available in the original paths.

Above is an example with two SP-graphs, one with two paths, from node 1 to node 4 and node 5 and another with a path from node 2 to node 4. It can be seen that if we try to add the arc $(3,4)$ to the first SP-graph, that it will then have two paths from node 1 to node 4 instead of one, and it is easy to see that there can be a different path from node 2 to node 5 in another SP-graph, and in both cases the result would be that the SP-graphs would give us different information than they did before, which is exactly what we would want to avoid.

But let's move on to when it is possible to combine paths. If we have two paths that have the same origin (destination) and two (or more) different destinations (origins), then they can be combined if they are identical in a subpath starting in the origin and then splits, without sharing a common node after they split. Or to put it in a more mathematical and precise language:

$A_1$ and $A_2$ are two SP-graphs. $N_0$ are the nodes that are common to $A_1$ and $A_2$, $N_1$ are the nodes that only $A_1$ connects and $N_2$ are the node that only $A_2$ connects. $o_1$ and $o_2$ belong to $N_0$, $d_1$ belong to $N_1$ and $d_2$ to $N_2$. Let $\gamma(B)$ represent all arcs with both start- and endnode in set $B$, then $\gamma(N_0) \cap A_1 = \gamma(N_0) \cap A_2$, i.e. all arcs in $A_1$ with both endpoints in $N_0$ are the same arcs as the ones in $A_2$ with both endpoints in $N_0$. This means that $A_1$ and $A_2$ are identical in a common part of the graph (with nodes $N_0$), which means that both SP-graphs must have the same startnode, i.e. $o_1 = o_2$, since they are both a part of $N_0$ and any arc leaving either $o_1$ or $o_2$ must be present in both SP-graphs. Let $N(C)$ be the nodes that are either startnode or endnode to any of the arcs in set $C$. We then demand that $N(A_1) \cap N_2 = \emptyset$ and $N(A_2) \cap N_1 = \emptyset$. This means that $A_1$ and $A_2$ are totally separated outside the identical part ($N_0$). Let $\delta'(B)$ be all arcs that enters node set $B$, i.e. all arcs with endnodes in node set $B$, but startnode outside $B$ and $\delta(B)$ be all arcs that leaves node set $B$. The final demand is that $A_1 \cap \delta'(N_0) = \emptyset$ and $A_2 \cap \delta'(N_0) = \emptyset$, which means that there are no arcs in either $A_1$ or $A_2$ that enters node set $N_0$. The only difference for when the paths has the same destination instead is that $d_1$ and $d_2$ belong to $N_0$, $o_1$ belong to $N_1$ and $o_2$ to $N_2$ and that we must check for arcs that leaves node set $N_0$ instead of entering, that is $A_1 \cap \delta(N_0) = \emptyset$ and $A_2 \cap \delta(N_0) = \emptyset$. All this can be condensed into Conditions A, with four parts (referred to as A.1 to A.4):

1. $N_0 \cup N_1 \cup N_2 = N(A_1 \cap A_2)$, where $o_1 = o_2$ and belongs to $N_0$, $d_1$ to $N_1$, $d_2$ to $N_2$
2. $N(A_1) \cap N_2 = \emptyset$ and $N(A_2) \cap N_1 = \emptyset$
3. $A_1 \cap \delta'(N_0) = \emptyset$ and $A_2 \cap \delta'(N_0) = \emptyset$ (out-graphs) $A_1 \cap \delta(N_0) = \emptyset$ and $A_2 \cap \delta(N_0) = \emptyset$ (in-graphs)

Erik Haraldsson, 2008

2.5 Restrictions to combining SP-graphs
4. \( \gamma(N_0) \cap A_1 = \gamma(N_0) \cap A_2 \)

It is shown in [5] that if these conditions are true, then the two SP-graphs can be combined without altering the solution to the WFP. Conditions A will also work when one or both of the paths has several destinations. We can also discard both original SP-graphs if conditions A are true and we save the combined SP-graph, and in this way diminish the number of SP-graphs. I will give several examples to show how these conditions can be used:

First both SP-graphs has the same startnode, node 1, and \( N_1 = \{3, 6, 7\} \) (the nodes that only SP-graph 1 connects to) and \( N_2 = \{5\} \). This means that \( N_0 = \{1, 2, 4\} \), since they are the nodes that are left, or alternatively, the nodes that both SP-graphs connects. It is also easy to see that both SP-graphs has the same arcs with both endpoints in \( N_0 \), that is the arcs \((1,2)\) and \((2,4)\). However condition A.3 is not fulfilled, since the first SP-graph has one arc that has its endnode in \( N_0 \), but its startnode outside and that is the arc \((3,4)\). This means that there two SP-graphs can't be combined. The reason for this is that there is a 1-valid cycle in the two SP-graphs, with the arcs \((1,3)\) and \((3,4)\) from SP-graph 1 and the arcs \((1,2)\) and \((2,4)\) from SP-graph 2 and this part of the information would be lost if they had been combined.

In this example, the SP-graphs has the same endnode, node 7, and the first SP-graph has two startnodes. Here \( N_1 = \{1, 2, 5\} \), \( N_2 = \{3, 4\} \) and \( N_0 = \{6, 7\} \). Here condition A.3 means that we should look for arcs leaving \( N_0 \), i.e. arc with a startnode in \( N_0 \), but an endnode outside \( N_0 \), which clearly isn't the case here, and both SP-graphs also has the same arcs that connects both node 6 and node 7 (that is the arc \((6,7)\)). This means that all conditions are fulfilled and we are allowed to combine the two SP-graphs into one big, that will look like:

Now a slightly more complicated example:
Here node 1 is the common startnode and the nodes are divided as $N_1 = \{5, 7\}$, $N_2 = \{6\}$ and $N_0 = \{1, 2, 3, 4\}$, according to conditions A.1 and A.2. There are no arcs that ends in node 1, both SP-graphs has the same arc that ends in node 2, and its startnode is also in $N_0$, and the same is true for node 3 and node 4, which means that conditions A.3 and A.4 are also true and the two SP-graphs can be combined to one, without any information being lost or added.

Another similar case is when one SP-graph is connected from $o_1$ to $d_1$ through $o_2$ and the other goes from $o_2$ to $d_2$. It is then possible to add the set of subpaths from $o_2$ to $d_1$ in the first SP-graph to the second SP-graph, if conditions A are true for the part of SP-graph 1 that starts in $o_2$ (this is again shown in [5]). We can however only delete the second SP-graph, after they have been combined, since the first SP-graph still contains information about the shortest paths from $o_1$, that isn't used in the combined SP-graph. This is useful to make each SP-graph contain more arcs and connect more nodes, but it doesn't shrink the number of SP-graphs used.

Here SP-graph 1 starts in node 1 and SP-graph 2 starts in node 2, but we can try and add SP-graph 1 to SP-graph 2, since SP-graph 1 connects to node 2. This means that we take the subpath from node 2 to node 5 from SP-graph 1 and sees if it is allowed to combine it with SP-graph 2, as shown in the right part above. This means that $N_1 = \{5\}$, $N_2 = \{4, 6\}$ and $N_0 = \{2, 3\}$ and that arc $(2,3)$, that is part of both paths, is the only arc with an endnode in $N_0$. This means that the subpath from node 2 to node 5 can be added to the second SP-graph without any problems, but that both SP-graph 1 and the combined SP-graph should be saved.

**Theorem 2:** Let's start with two SP-graphs, that has one common startnode (endnode) and that it is the only startnode in either SP-graph. Then the two SP-graphs being subpath consistent is equivalent with that condition A for combining SP-graphs being true. That is if the SP-graphs are subpath inconsistent then condition A is false, and if they are subpath consistent then condition A is true.

**Proof:** First, if both SP-graphs has $o$ as the only startnode, then it means that all sets of paths in these SP-graphs has $o$ as their startnode, and that there exists a subpath from $o$ to any other node connected by the SP-graph. We can always divide the nodes so that condition A.1 and A.2 are true. If condition A.4 is false, that is if $\gamma(N_0) \cap A_1 \neq \gamma(N_0) \cap A_2$, then this means that one SP-graph has at least one arc between two of the nodes in $N_0$, that is missing from the other SP-graph. Assume that it is an arc from $a$ to $b$. We know that it is possible to get from $o$ to any other node, so there is at least one subpath from $o$ to $a$, and this means that a subpath from $o$ to $a$ and the arc $(a,b)$ will form a subpath from $o$ to $b$. But this means that the two SP-graphs won't be subpath consistent from $o$ to $b$, since $(a,b)$ isn't in the other SP-graph, and it is possible to get from $o$ to $b$ here too. Now let's look at condition A.3, that $A_1 \cap \delta'(N_0) = \emptyset$ and $A_2 \cap \delta'(N_0) = \emptyset$. If this isn't true, then there is a arc that
connects to a node in \( N_0 \), so call this arc \((c,d)\), where \( d \) is in \( N_0 \), but \( c \) isn't. There will in this case be a subpath from \( o \) to \( c \), since all sets of paths starts in \( o \), and there will again be a subpath from \( o \) to \( d \), that contains a subpath from \( o \) to \( c \) and the arc \((c,d)\). That \( c \) isn't in \( N_0 \) means that only one SP-graph is connected to \( c \), but both SP-graphs are connected to \( d \), so both will have one or more subpaths from \( o \) to \( d \), and one has the subpath through \( c \) and the other isn't connected to \( c \), so the two SP-graphs will be subpath inconsistent in this case too. The result is that if the first two conditions are used to divide the nodes, and are thus always true, then we can test if the last two conditions are also true. If at least one of them are false, then the two SP-graphs can't be combined. If either the third or the fourth condition is false, then it is shown that the SP-graphs are subpath inconsistent.

Second, now we have two SP-graphs with the same startnode (\( o \)) that are subpath inconsistent. This means that they have different arc from \( a \) to \( b \) (where \( a \) and \( b \) represents the smallest part that is subpath inconsistent) and that both SP-graphs are connected to both \( a \) and \( b \). At least one SP-graph will thus have an arc that ends in \( b \), and that isn't a part of the other SP-graph. Call this arc \((c,b)\). If \( c \) is a part of \( N_0 \), then condition A.4 is false (unique arc in \( N_0 \)). However if \( c \) is not a part of \( N_0 \), then condition A.3 is instead false (arc that ends in \( N_0 \)).

Call subpath consistent B and condition A being fulfilled C. Then the first part of the evidence shows that \( \neg C \rightarrow \neg B \) and the second part that \( \neg B \rightarrow \neg C \), which is equivalent to \( C \rightarrow B \), and thus \( C \leftrightarrow B \). QED.

This means that two SP-graphs with the same startnode (endnode) can be combined according to Conditions A if they are subpath consistent, but not if they are subpath inconsistent.

**Theorem 3:** Let's start with a set of SP-graphs, \( \tilde{A} \). If \( \tilde{A} \) are subpath consistent before combining the SP-graphs according to condition A, then the resulting SP-graphs will also be subpath consistent. If \( \tilde{A} \) instead are subpath inconsistent, then the combined SP-graphs will also be subpath inconsistent, i.e. combining SP-graphs doesn't change the status of subpath consistency.

**Proof:** Part A: Let's first assume that \( A_i \) and \( A_j \) are subpath inconsistent in the set of paths from \( a \) to \( b \), \( S_i(a,b) \) and \( S_j(a,b) \). If \( \tilde{A} \) now is combined and \( A_i \) and \( A_j \) are part of two different resulting SP-graphs, then they will still have different sets of paths from \( a \) to \( b \). It is also not possible to combine \( A_i \) and \( A_j \), according to theorem 2 above, which means that the combined SP-graphs will still be subpath inconsistent.

Part B: Now assume that all SP-graphs are subpath consistent, and that several SP-graphs are successfully combined into \( A_\ell \) and that it has the subpaths \( S_\ell(a,b) \) from \( a \) to \( b \). There are several SP-graphs left to combine, let's call them \( A_1, A_2, A_3, A_4, ... \) and say that we want to combine them into \( A_\ell \) in such a way that \( A_\ell \) and \( A_\ell \) are subpath inconsistent, between say node \( a \) and \( b \). Is this possible? Since all SP-graphs are subpath consistent, if any of them has a set of paths between \( a \) and \( b \), it must be equal to \( S_\ell(a,b) \). Would it be possible to combine an SP-graph with a set of paths from \( a \) to \( c \), with one with a set of paths from \( c \) to \( b \) (but that is not connected to \( a \), in order to be subpath consistent)? No, because if they have a common node prior to \( c \), then the SP-graphs would be subpath inconsistent, which is contrary to what was initially assumed. If they instead had no common node before \( c \), then since both SP-graphs must be connected to at least one of the startnodes, it follows that \( c \) is the only possible option. However in that case only the part of the first SP-graph, that comes after \( c \) can be used, so that means that the \( a \) to \( c \) part can't be used. All this means that it is impossible to combine subpath consistent SP-graphs into subpath inconsistent ones.

So all in all, this means that subpath consistency isn't affected by combining SP-graphs. QED.
2.6 Summary

The main goal is to find a set of weights that will generate a specific set of shortest paths. An alternative first step is to check whether there can be a set of weight or not. I described two conditions that tell us that there are no valid weights, the 1-valid cycle and subpath inconsistency, where the latter is a subset of the former. I also described Condition A, which gives a sufficient but not necessary condition in order to combine SP-graphs and gave a simple way to interpret it, which will be used in my program.
Chapter 3: Mechanisms of the Program

3.1 Introduction

In this section I will at length discuss what the different parts of my program does and how it is built. When the program starts it reads the SP-graphs from a file, and saves the data for each SP-graph in a structure, called \textit{SPGraph}. It contains one vector that stores the arcs in the SP-graph, called \textit{arcList}, two vectors that stores the out-degree and in-degree respectively for each node, called \textit{outDegree} and \textit{inDegree}, two more to store the origins and destinations, called \textit{startNodes} and \textit{endNodes}, and finally a set to store the nodes that are connected to this specific SP-graph, called \textit{nodes}. Below I will first talk about what it means for a program to be efficient, and how I have built my program in general, in consideration to this and other important things. The focus will then shift to the different parts of the program.

3.2 Running Times

Here I will discuss running times and related subjects, which will give us a rough estimation of how effective a certain algorithm is and how much it is affected by a change in size. Let's look at the structure of the data. First, it contains of a finite number of SP-graphs, here called \textit{m}. Then each SP-graph has a list of arcs, called \textit{A_S}, which makes \(|A_S|\) the number of arcs for a random SP-graph and \(|A_L|\) the biggest of the \(|A_S|\), i.e. \(|A_L| = \max |A_S|\). In some cases the structure of an SP-graph will also be important, so \#end stands for the total number of endnodes (or startnodes depending on how a function is used, see more later) and \#paths stands for the number of paths that are combined (if an SP-graph has one startnode and two endnodes, and there are three paths to the first endnode and one path to the second, then \#paths=4). \textit{N} is all nodes that are a part of a problem, and \(|N|\) is the total number of nodes, which e.g. determines the size of the \textit{inDegree}-vector. The number of nodes that a specific SP-graph connects to is smaller than or equal to \(|N|\).

Now it is time to see if there are any relations between the variables. First, in the special case when each set of paths contains only one path, i.e. when ECM isn't used, then \(|A_L| \leq |N|-1\) (basically think of the SP-graph as a spanning tree), and in the general case when ECM is used, \(|A_L| \leq |N|*(|N|-1)/2\) (think of a case when \(a_{ij}\) is used if \(j>i\), i.e. every node has an arc for each node with a higher node number, so the first node has \(|N|-1\) arcs, the second \(|N|-2\) arcs and so on). However this is the absolute worst case and by thinking of what this is all about, determine routes to send data, and that since data will be sent from all nodes and to all nodes, it makes no sense whatsoever to let data from \(i\) to \(j\) go through as many different arcs as possible. It is smarter to let different origin-destination-combinations use different parts of a network and in that way get a reasonable balanced load. So the number of arcs for a specific path will likely be relatively low. One interesting thing is that a single path will always have less than \(|N|\) arcs (since it will by definition have one incoming and one outgoing arc for every connected node that isn't the origin or destination and one long path that connects all nodes have \(|N|-1\) arcs). Let then \(m\) be the number of SP-graphs in a problem. It is reasonable that there is only one SP-graph for every origin-destination-pair, which means that for each node, there are at most \(|N|-1\) SP-graphs, which yields that max \(m = |N|*(|N|-1)\). In the special case when all the SP-graphs are already combined, then \(m\) will be roughly the same size as \(|N|\). The reason for this is that the combination algorithm tries to create complete out-graphs (or in-graphs), and that most SP-graphs with the same origin will be in one SP-graphs and if there are more than one SP-graph with the same startnode, then they must be subpath inconsistent according to theorem 2.

When looking at how efficient the code is, the really significant thing is how the solving time changes as the size of the problem changes. So if a function depends linear on \(N\), it is written as \(O(\sqrt{N})\). This means that we are normally not interested in the constants or the slower growing parts of a
function, so that \( O(a|N| + b|N| + c|N|) = O(|N|) \) and \( O(a|N|^2 + b|N| + c) = O(|N|^2) \), and [1] has more information about calculating the complexity of a program. Throughout this paper I will discuss the complexity of the different parts of the program.

While having a low complexity is important, it doesn't tell everything about the efficiency of an algorithm. I did at one time accidently copy the SP-graphs from one function to another (instead of just sending a reference to the SP-graphs) and it made the total running time for the program go from about 0.5 to 20 seconds. One way to understand this is to look at how much space the program needs to save an SP-graph, the arcList needs \( A \) items, the inDegree and outDegree each needs \( N \) items, the nodes needs \( N \) items at the most and startNodes and endNodes needs \#start+\#end items. This brings the total to \( 2A + 3N + \#start + \#end \), where each arc takes twice as much memory as a node because an arc is a pair of nodes. Each node is saved as an integer variable and an integer is stored with 4 bytes (in Linux and most modern systems generally). Some of the largest SP-graphs that I have tested this program on have 90 nodes and about 3500 SP-graphs, but each SP-graph then normally only have one set of paths (so \#start=\#end=1) and most of them only have a few arcs. Here we would probably need around 3\( N \) to 4\( N \) integers to store each SP-graph (this is by no means a maximum size, but just what can be expected from the typical largest problems that I have used), which is equal to \( 12N-16N \) bytes to store an SP-graph. This equals to a total size of about 4 MB \((14*90*3500/1024^2)\) for the largest problems that I have studied. It is of course good to not copy 4 MB of data several times, and especially important to not copy data to functions that compares each pair of SP-graphs (because then the total amount of data will go from a few megabytes to a few gigabytes).

### 3.3 The structure of my program

There were a few general things that I wanted to do, when I wrote the program. Of course I wanted it to be effective, but there were also other things that was important. One important aspect was to make the program easy to read and understand. I wrote the program in C++, and generally used [6] as a guide on how to use C++. The main advantage of C++ over C, was, from my point, that it allowed me to use STL (see below).

Most functions have names that try to tell what they do, and all important variables either reflect what they stand for, or has a general name that describes the general idea of the variable. Examples on the second category are in\( SP \), that is used for a general SP-graph (or vector of SP-graphs) that is only used as indata to a function, and out\( SP \), SP-graph(s) that the function is supposed to write changes to. Generally if a variable has “in” in it, then it means that it is imported as a constant and therefore read-only and functions as (some of) the data to the function, and a variable with “out” means that the function will write data to the variable. There are a few exceptions to this rule in the first category, like in\( Graph \) is used to specifically note a graph with several startnodes and one endnode, and out\( Graph \) the opposite.

Functions are another way to make the programs easier, by dividing everything in steps and sub-steps, and most often make every step into its own function. I also tries to send all variables as an argument to a function. There are several benefits to do this, first if I change the name of a variable in one function, I don't have to change the name in every function, and thus making the program more robust to changes. Second, it is easier to reuse a function, since it is just to call the function with another variable as the argument instead. It can sometimes take a while to see a new use for an old function (which makes it harder to rewrite the function) and using arguments makes the program more flexible in general. The only negative thing is that it takes slightly longer to write a function. There are a few exceptions to this general case, where global variables are used, most important the integer variable number\( OfNodes \), that is \(|N|\), the highest node number in all the SP-graphs, and because the nodes are (assumed to be) numbered from 1 and up, the total number of nodes in the current data. The reason for this variable to be global, is that it is used in many functions and that it
stays the same for as long as the program is loaded (since the only way to read in new data from a
file is to restart the program). There are also a few global variables, that relates to the control of the
program. One important thing is to send a reference to a variable, instead of an actual copy of the
variable to a new function when it is possible and the variable is big (like if it is a vector or
structure). An alternative to references are pointers, but I find it more straight-forward with
references (it is possible to write all code except in the argument-list as if it was an actual variable
instead of a reference, but with pointers, you always has to use special commands).

Another thing that ties in with functions and arguments is to try and write as little similar code as
possible. First it makes the program smaller and easier to understand, and second it makes it easier
to do small changes in one function than two or more (the more functions the easier it is to not do
the change in one of them). One important thing here is that I use a variable, forward, to indicate
whether the function will treat an SP-graph as it looks, or as if all arcs were reversed. This makes it
possible to e.g. have one function to create both in-graphs and out-graphs.

I also use STL, which stands for Standard Template Library and is a collection of containers and
functions and iterators. The most important part is the STL-vector, that has fast access to all
elements and constant time for adding/removing elements in the end and linear time for
adding/removing random elements. C++ does already have a normal vector defined, so what makes
the STL version better? The most important aspect is that the size of a normal vector either has to be
defined in the source code or by using dynamic memory, and it is even then at least complicated to
resize a vector. Another thing with dynamic memory is that variables that are allocated dynamically
must be deleted manually or it will create a memory leak, which adds another layer of complication
to the code. A STL-vector, like any other STL-container, can be treated as a normal variable and at
the same time have the size be defined by the current problem and can even be resized at any time.
Basically, what STL does is to reserve some extra space when a new vector is created, and when the
extra space is all used up, the vector is moved to another part of the memory with more extra space.
This means that I have tried to either set the size as soon as possible, or at least reserve so much
extra space, that it will probably not need to move the vector, and by this saving time. Another
feature of the STL-containers is that the size of the vector is always easy to find, so there is no need
to use a special integer for the size. At some places I have used another container called set. A set is
automatically sorted and can only hold the same value once (if you have an int-set, and first adds 5
and then 7 and then 5, then it contains 2 items, 5 and 7). One drawback about the set compared to a
vector is that the set doesn't have a way to get instant access to the value in a certain element, i.e.
using vector[n-1] will return the value of the n:th element, but there isn't any similar thing for the
set. Another part of STL are iterators, that works like pointers to a certain item in a STL-container
and can be used in a for-loop to look at each item in a container. STL also contains several methods
(like getting the size of a container, or to resize a container) and functions to automatically do
certain tasks, like sort the items in a vector from lowest to highest, or find the total number of times
that an item is stored in a container. While these functions normally work with regular vectors, it is
often easier to use them with STL-vectors, because of the iterators and methods. I have used [7] to
get the complexity of all the STL-functions that I have used, and it is generally a good source for
information about STL and its parts.

Another choice that I made was to make all the variables as unsigned integers. This means that the
variable can store integers from 0 up to a predefined maximum value. First, nodes, arcs and
everything else interesting about the SP-graphs, are all given in a finite, countable number, so there
is no need for real variables (variables that can store numbers with decimal parts), and using
integers avoids any rounding errors. Second there isn't any use for negative numbers (except maybe
as to indicate that something went wrong, but it can also be done in a different way), so only using
unsigned integers doubles the range of numbers possible in the program (not that it was even likely
to hit the limit for normal integers, but it can't hurt).
The program assumes that the nodes in the incoming file are numbered from 1 and up to the final node, and also uses node number from 1 and up internally. However, in C++ the indexes for vectors must always start at zero. The solution that I used is to store the data for element 1 in position 0, the data for element 2 in position 1 and generally store data for element n in position n-1. One example here is the `outDegree`-vector, where the number of outgoing arcs from node 1 is stored in `outDegree[0]`, and generally `outDegree[n-1]` is the number of outgoing arcs for node n.

### 3.4 Storage

To store all the SP-graphs, I use two structures, where a structure is a way to easily use several different variables as if they were just one variable made up of several parts. The first one, `arc`, is (as the name suggest) used to store an arc, i.e. contains one integer for the startnode and one integer for the endnode. There are some important functions for `arc`:

- **A definition of the ==-operator**: Two arcs are identical (i.e. `arc1 == arc2`) if they have the same startnode and endnode. In that case the function returns the value `true` and otherwise value `false`.

- **The <-operator**: Defines what means if an arc is smaller than another arc. This function is mostly of interest when we want to be able to sort the arcs. The smallest arc is the one whose startnode is smallest, and if they have the same startnode, the one with the smallest endnode.

- **setArc and setArcB**: B stands here and in other functions for boolean and means that the function takes an extra boolean argument that tells if the function will give the correct value or the opposite. There are two versions of `setArc`, where the first version takes an arc as argument and returns a copy of the arc, while the second version takes two (unsigned) integers as the startnode and endnode and then returns the arc from the startnode to the endnode, that is (startnode, endnode), and which version is used depends on what arguments is used to call it. The function `setArcB` takes two integers and a boolean variable and returns the arc (startnode, endnode) if the boolean is true and the arc (endnode, startnode) if it is false.

- **returnArcStartB**: Takes an arc as an argument and returns its startnode, there is also an optional boolean argument, that decides if the function returns the real startnode or the opposite (i.e. the endnode).

- **returnArcEndB**: As `returnArcStartB` but obviously returns the endnode instead.

The second structure, `SPGraph`, is used to store an SP-graph. First, it contains a STL-vector of arc-structures, called `arcList`, like this:

\[
\begin{align*}
\langle 12 \rangle \\
\langle 23 \rangle \\
\langle 34 \rangle
\end{align*}
\]

From this vector of arcs, it is possible to get all possible knowledge of the SP-graph. Some information is easy to get, like the number of arcs, while other interesting information is not so obvious, like the number of startnodes for the SP-graph. There are thus several other elements in the structure `SPGraph`, in order to write a efficient program. First there is a vector for the in-degree (number of incoming arcs to each node), called `inDegree`, and one for the out-degree, called `outDegree`. Then there are vectors for the startnode(s) and endnode(s) of the whole path, called `startNodes` and `endNodes` respectively. And last there is a set-container, that holds all nodes that the path is connected to, naturally called `nodes`.

The whole problem is then saved in a vector of SP-graphs. Some basic functions that deals with SP-graphs are:
• **startNodeSizeB** and **endNodeSizeB** returns the number of origins or destinations for the SP-graph, and as above B means that there is also a boolean variable.

• **getStartNodeB** and **getEndNodeB** returns the value of a specific origin, and as above B means that if the boolean is false then it returns the specific destination instead of origin. While it would have been enough with one function in each case, having two functions makes it easier to read the code.

• **nodeInDegreeB**, returns the in-degree to a specific node (or if the boolean is false, the out-degree) in an SP-graph and **nodeOutDegreeB**, that returns the opposite as above.

• **calculatePathData** checks all arcs in an SP-graph and sets **inDegree**, **outDegree** and **nodes** for every arc. Then the origins and destinations are determined with the help of **inDegree** and **outDegree**. It also makes sure that all **arcList**-vectors are sorted, to more easily compare the arcs for two SP-graphs.

• **combinePaths**: There are two versions of this function and both adds more arcs to a SPGraph. One version gets the arcs from a sorted SPGraph-structure and the other version gets them from the “unsorted” **arcStack** that DFS uses.

• **updatePathData** is called from **combinePaths** to make sure that the rest of the SPGraph-structure is up-to-date. The reason for this to exist is that it is faster than **calculatePathData**.

There is also a structure that saves the processed information from the arguments given when the program starts.

### 3.5 Startup

The program starts by reading indata from a file. The data in the file is given as:

```plaintext
2    // This number tells the program how many SP-graphs that the data has.
3    // This number describes the number of arcs in the first SP-graph.
1 2  // This is a directed arc from node 1 to 2, that is a part of SP-graph 1.
2 3  // Arc from node 2 to node 3, that is also a part of SP-graph 1.
3 4  // Third and last arc in SP-graph 1.
2    // The number of arcs in SP-graph 2.
2 4  // First arc in SP-graph 2, from node 2 to node 4.
4 5  // Last arc, from node 4 to node 5.
```

The program assumes that all data is given correctly and doesn't check for where there is a new line or if there is more data after it has read the given number of SP-graphs.

First it saves the arcs in each SP-graph to the **arcList**-vector, and then go through the list to add up all the arcs that enter or leave each node and which nodes are used, then looks to see which nodes have a in-degree of 0 and an out-degree higher than 0 and mark those nodes as a startnode and similarly marks the endnodes. Finally the **arcList** is sorted, since some functions becomes easier if the arcs are sorted.

To make everything clear, here is a fairly short example with two SP-graphs, one from node 1 to node 6 (unbroken line) and one from node 1 to node 7 (dotted line):
For this example, the indata-file would look like:

```
2
6
1 2
2 3
3 5
2 4
4 5
5 6
6
1 2
2 3
3 5
2 4
4 5
5 7
```

SP-graph 1 would be stored like:

<table>
<thead>
<tr>
<th>Position</th>
<th>arcList</th>
<th>inDegree</th>
<th>outDegree</th>
<th>startNodes</th>
<th>endNodes</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2 3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2 4</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3 5</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4 5</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5 6</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 14:** Two SP-graphs to show how the program get the information in the graphs.

For this example, the indata-file would look like:

```
2
6
1 2
2 3
3 5
2 4
4 5
5 6
6
1 2
2 3
3 5
2 4
4 5
5 7
```

SP-graph 1 would be stored like:

<table>
<thead>
<tr>
<th>Position</th>
<th>arcList</th>
<th>inDegree</th>
<th>outDegree</th>
<th>startNodes</th>
<th>endNodes</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2 3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2 4</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3 5</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4 5</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5 6</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>0 0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.6 The use of DFS

The program needs to be able to move along the structure of an SP-graph, for instance if it should save only one of several paths. There are two general strategies for graphs that can be used, Depth-First Search (DFS) and Breadth-First Search (BFS), and a more in-depth discussion about them can be found in [1].

The general meaning DFS is that you start at the top and searches one branch at a time. Here it translates to starting at a selected origin and choose an arc from the origin to another node and then an arc from that node etc., until a stop criterion is fulfilled. Then the program backtracks until it finds a node with an arc that haven't already been visited.

BFS means that starting at the top and look through all branches in the same layer before moving on until the next layer. Here that means that you start at an origin and then look at all nodes that has an
arc from the origin to them and then at all nodes that has an arc from any of the now checked nodes etc. until a stop-criterion is reached in each branch.

This program uses DFS exclusively, because sometimes the path found needs to be saved and used in the program and it doesn't need to save as much data as BFS, while both algorithms are expected to be roughly equally fast.

When dealing with directed graphs, it is natural to see the arcs that start in a certain node as branches. An necessary stop criterion is to reach an endnode, but there can also be other criteria (the program also uses already searched nodes and in one case if it finds a node that it looked for as branch-stop-criteria).

Let's assume that the algorithm is used going forward in the path, from startnode the endnodes (it works just the same if it is used to go backwards):

• Start the DFS at node n and find an arc that starts in n.
• Save the arc.
• Mark that DFS has already searched one of the arcs starting in n.
• Check if the endnode of the arc to see if any stop criteria is fulfilled.
• Set n=the endnode that was just found and repeat.

Then the saved arcs are removed one at a time until it finds a node from which not all arcs (branches) are searched and performs a new DFS from there, and this continues until all branches has been searched.

To do all this the program needs some functions and variables:

arcStack: This arc-vector holds the current simple path from the startnode to the branch that the DFS is in now. As the name suggests, it basically works as a stack, i.e. after the last in, first out-method. The stack-container that is a part of STL isn't good enough, since I want the program to be able to copy the found path to a new path-structure.

nArcsToCheck: is an integer-vector, that initially is a copy of the out-degree for the nodes (in-degree if using DFS backwards) and represents how many of the outgoing arcs from a node that the algorithm hasn't checked yet, so each time we put a new arc in arcStack, the value for the corresponding startnode is lowered by one. If the program gets to a node with its value at 0, then it's either an endnode or a node that has already been visited (since the first time the node was visited, all of the arcs starting in the node was also searched) and there isn't really any point in continuing DFS in either case, which makes nArcsToCheck[node]==0 a good stop criterion.

getArc: This is a function that takes in an startnode, s, and returns an endnode, e, so that (s,e) is a valid arc in the incoming SP-graph. It is based on the condition/presumption that the arcList in the SP-graph that it looks through stays the same throughout the whole DFS-algorithm. This because it takes an argument that tells it if it should return the first or second or third (and so on) arc that matches the startnode-condition. The DFS-algorithm normally sends in the value-1 in nArcsToCheck for the current startnode.

combinePaths: If the algorithm finds a node that has already been searched, then it is smart to not try and search through branches further down a second time. Instead this function takes the new path and tries and combine it to one or several other paths, based on whether they are connected the searched node, by adding the new arcs at the end of the current arcList (this is for the outgoing path(s), not the incoming that is used in the DFS, so it doesn't affect getArc) and updating which of the nodes are connected to the path.
backTrackPath: This function removes one arc at the time from the arcStack until either it gets empty, in which case all arcs has been searched through and we can abort the DFS altogether, or until the endnode of the last arc has a nArcsToCheck-value that is greater than zero, i.e. all arcs from that node has not yet been searched through.

Here below follows the interesting part of my code for DFS (the complete function contains a few more lines that wasn’t relevant for this discussion):

```cpp
void getPartOfSPGraph(const SPGraph & inSP, SPGraph & outSP, const bool createOutGraph,
                       const unsigned int startNode, unsigned int nodeToFind)
{
    vector<arc> arcStack;
    vector<unsigned int> nArcsCheck (numberOfNodes);
    if(createOutGraph)
        copy(inSP.outDegree.begin(), inSP.outDegree.end(), nArcsCheck.begin());
    else
        copy(inSP.inDegree.begin(), inSP.inDegree.end(), nArcsCheck.begin());

    unsigned int tempNode, nextNode = startNode;
    while (nextNode != 0) //at most min(#paths,|A|) times
    {
        //1: Find new path
        if (nArcsCheck[nextNode-1] != 0)
            do
            {
                tempNode = nextNode;
                nextNode = getArc(inSP, tempNode, nArcsCheck[tempNode-1]-1,
                                  createOutGraph); //O(|A|), at most |A| times
                nArcsCheck[tempNode-1]--;
                arcStack.push_back(setArcB(tempNode, nextNode, createOutGraph));
            } while ( (nextNode != nodeToFind) and (nArcsCheck[nextNode-1] != 0) );
        //2A: Add the latest path to the outgoing SP-graph
        if (outSP.nodes.count(nextNode) > 0 or nodeToFind > numberOfNodes)
            //at most #paths-1 times
            combinePaths(outSP, arcStack); //O(|A|*log(|A|))
        //2B: Use the latest path as a base
        else if (nextNode == nodeToFind or nodeToFind == 0) //At most one time
            {
                outSP.arcList = arcStack;
                calculatePathData(outSP); //O(|A|*log(|A|))
                if (nodeToFind == 0)
                    nodeToFind = numberOfNodes + 1;
            }
        //3: backtrackPath removes arcs from arcStack until we find an arc we haven't been looking
        //at and returns 0 if we have searched all arcs
        nextNode = backtrackPath(nArcsCheck, createOutGraph, arcStack); //O(|N|)called at most #paths times
    }
}
```

This version of DFS will find all paths from startNode to nodeToFind, and return them as an SP-graph. It is also possible to call the function with nodeToFind=0, and in that case, the function starts to search in startNode and returns an SP-graph with the paths from startNode to all endnodes in the SP-graphs (that are reachable from startNode). Part 1, the inner loop, is used to add one arc at a time to arcStack until we either gets to nodeToFind or we get to a node with a out-degree of 0 or a node where all outgoing arcs has already been searched. The first time a valid path is found, its arcs are simply added to the outgoing arcList, step 2B, but for the rest of the valid paths the function combinePaths is called, step 2A. Finally the program runs backtrackPath, step 3, that will return a 0 and exit the outer loop when all paths starting in startNode has been searched. If the boolean createOutGraph is false instead of true, then the function will instead find paths that ends
in \textit{startNode}, and send the boolean flag on to those functions, that needs to know which direction to look at.

Here is an example that will make DFS even more clear. Let's assume that \textit{createOutGraph} is false and that the algorithm is supposed to be used to find the set of paths from node 9 (\textit{startNode}=9) to node 3 (\textit{nodeToFind}=3). First the \textit{nArcsToCheck}-vector is set to the in-degree for each node. The algorithm starts in node 9 and finds that the arc (7,9), which is then added as the first arc in \textit{arcStack}. This means that \textit{nArcsToCheck}[8]=0, since there was only one incoming arc to node 9, which means that there is no more arcs that starts (actually ends, but that starts from the perspective of the algorithm) in node 9 that hasn't already been searched. Then the arc (6,7) is found and added to \textit{arcStack}. Then arc (4,6) and the program notes that all arcs ending in node 6 are also searched. Then arc (3,4) is added, the \textit{nArcsToCheck}-element for node 4 is diminished by one, and therefore stands at one, to note that there is one arc left to search. The DFS part is aborted, since the program has found the node that it was looking for and the arcs (3,4), (4,6), (6,7) and (7,9) are added to the outgoing SP-graph, that is then updated (the new in-degree and out-degree and so on). Then the function \textbf{backTrackPath} is called, and it removes the arc (3,4) from \textit{arcStack} (but stops at node 4 because not all arcs entering has been searched). The program now starts the DFS over from node 4, and finds the arcs (4,2) and then (2,1) and since there are no incoming arcs to node 1 the DFS is aborted again. The program will discard this path since the last node, 1, isn't already a part of the first path added to the outgoing SP-graph. The program once more removes arcs from \textit{arcStack} and doesn't stop until it gets to node 7, that has one more arc to look through. It starts DFS again and finds the arcs (5,7) and (4,5) and notes that there are no more arcs to search in node 7 or 5. When it gets to node 4, the DFS will stop since all incoming arcs to node 4 has already been searched, and thus \textit{nArcsToCheck}[3]=0. Node 4 is however, unlike node 1, part of the nodes that the outgoing SP-graph has already connected, so the arcs (4,5) and (5,7) are then added to the SP-graph, and its data is updated. Now \textbf{backTrackPath} will remove all arcs in \textit{arcStack} and will return a 0 instead of the node that it stopped at, to indicate that all arcs from node 9 and further down in the graph has been searched and that the algorithm should stop.

Since DFS first tries to get as deep into an SP-graph as possible and then only backs enough to find the latest unsearched arc, it follows that if a new path gets to a node that the program has already been to, that all arcs leaving that node has already been searched. This means that the inner loop will at most be run \(|A_i|\) times. The \textbf{getArc} function in the inner loop takes \(O(|A_i|)\) time for each time it is called, which means that the total time for the inner loop is \(O(|A_i|^2)\). The things in the outer loop will at most be called \#paths times, since the program looks at a new path after each time that it has backtracked. We do however know that the inner loop will at most be called \(|A_i|\) times, so that must also be an upper limit for the number of times that the outer loop is run, so the functions in the outer loop is at most called \(|A_i|\) times (since \(|A_i|\) is more predictable than \#paths and also easier to find out). Here \textbf{backTrackPath} adds \(O(|A_i|*|N|)\) (\(|N|\) because the \textit{arcStack} contains just one path) while the first valid path adds \(O(|A_i|*\log(|A_i|))\). Finally the \textbf{combinePaths}-function adds \(O(|A_i|^2 * \log(|A_i|))\). This all means that \(O(|A_i|^2 * \log(|A_i|))\) will be the worst-case complexity for the normal DFS-function.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{sp-graph.png}
\caption{SP-graph used to illustrate DFS}
\end{figure}
3.7 How to divide SP-graphs

The program can divide all SP-graphs that are in the given indata-file, and save the result to a new vector where all SP-graphs has one origin and one destination, i.e. all SP-graphs contains just one set of paths. Since the origins and destinations are saved in their own vectors, it is easy to use the size of these vectors to check for the structure of the SP-graph. There are four cases:

1. One origin and one destination: This SP-graph already contains just one set of paths, and can't be divided more (without creating misinformation) and will just be added at the end of the vector.

2. One origin and several destinations: This is an out-graph and the program will use DFS in the direction of the graph, and then add as many divided SP-graphs, as there are destinations, at the end of the vector.

3. Several origins and one destination: The SP-graph is an in-graph and to get each set of paths, the program starts at the destination and uses DFS in the opposite direction of the graph, and then adds the found sets of paths at the end of the vector.

4. Several origins and several destinations: The program uses DFS once for every origin. The result will at most be origins*destinations new divided SP-graphs (it is possible that there are fewer since it may not be possible to get from all origins to all destinations).

This function uses DFS as described above, by start searching the first origin (or destination in case 3), and will find a path to an endnode. This path is then added to a new SP-graph. It will backtrack and then either find a new endnode, to add to a new SP-graph, or a node that has already been visited. The program will in the latter case check each of the new SP-graphs, and add the path to the SP-graphs that are connected to the endnode of the found path. We know that all arcs from a node has already been searched when the program gets to a node for the second or third time, since DFS searches through all branches from a node before returning to an earlier node. Here below is a small example that will illustrate the operations.

First DFS finds path 1-3-4-6 and adds it to the outgoing divided paths, backing one step and finds path 1-3-4-5 and then adds that to the outgoing SP-graphs. Now it backs to node 1. The program finds path 1-2-4 and then aborts DFS because node 4 has already been visited. Since both resulting SP-graphs contains node 1 and node 4, the path 1-2-4 is added to both of the two already found SP-graphs. The result is two SP-graphs with one set of paths each, instead of one old, as seen below (where the unbroken arcs represent the set of paths in the first SP-graph and the dotted arcs the second SP-graph).

Figure 16: A SP-graph with two sets of paths

Figure 17: The resulting two divided SP-graphs
There are a few differences between this case and the more general case of DFS. First, it will here search through all arcs (normally) since this version will necessarily start in an origin (or destination) and look for all destinations (origins). Second, the program needs to create several new SP-graphs instead of just one and third, a path that ends in an already searched node, could be added to several SP-graphs instead of just one. The resulting complexity for the modified DFS will at most be $O(#\text{end} \cdot |A_L|^2 \cdot \log(|A_L|))$, where #end represents that each new path may be added to several SP-graphs. This function is however not that interesting in general, but more of a tool, that it is good to have available and the general complexity to divide all SP-graphs will be $O(m \cdot #\text{start} \cdot #\text{end} \cdot |A_L|^2 \cdot \log(|A_L|))$ in the worst-case, but depends wholly on the structure of the SP-graphs, and specifically on how often we need to use DFS. #start and #end are however not easily available, but each is smaller than $|N|$, so an easier, but more pessimistic, expression would be $O(m \cdot |N|^2 \cdot |A_L|^2 \cdot \log(|A_L|))$.

### 3.8 How to check in order to combine SP-graphs

I will now discuss how the program works, when it looks to see if two SP-graphs can be combined. First, let's quickly refresh the sufficient conditions (Conditions A) to be able to join graphs:

1. $N_0 \cup N_1 \cup N_2 = (A_1 \cap A_2)$, where $o_1 = o_2$ and belongs to $N_0$, $d_1$ to $N_1$, $d_2$ to $N_2$
2. $N(A_1) \cap N_2 = \emptyset$ and $N(A_2) \cap N_1 = \emptyset$
3. $A_1 \cap \delta^+(N_0) = \emptyset$ and $A_2 \cap \delta^+(N_0) = \emptyset$ (out-graphs)
4. $A_1 \cap \delta^-(N_0) = \emptyset$ and $A_2 \cap \delta^-(N_0) = \emptyset$ (in-graphs)
5. $\gamma(N_0) \cap A_1 = \gamma(N_0) \cap A_2$

One way to look at the conditions above is to divide the nodes into $N_0$, $N_1$ and $N_2$ in such a way that condition A.1 and A.2 are always true (i.e. the two first conditions does not determine if the paths can be combined, but only how to divide the nodes for condition A.3 and A.4), and then check if condition A.3 and A.4 are also true. One positive thing with this interpretation is that the code to check the paths becomes very simple and fast.

Condition A.1 and A.2 says that $N_1$ are the nodes that $A_1$, but not $A_2$, connects, and similarly that $N_2$ are the nodes connected by $A_2$ but not $A_1$. This automatically leaves us with $N_0$ as the nodes that both $A_1$ and $A_2$ connects. It is easy for the program to calculate $N_0$, since a part of each SP-graph is a set-container that stores which nodes are connected by that SP-graph. There is also a function in STL called `set_intersection`, that returns the common elements from two sorted intervals, which here is the nodes that both SP-graphs connects to. This function has a linear complexity, see [7], that is the running time depends on the number of nodes, $O(|N|)$.

Condition A.3 says that all arcs with the endnode in $N_0$ should also have the startnode in $N_0$ and condition A.4 that all arcs in $A_1$ between nodes in $N_0$ should be the same as the arcs in $A_2$. Both of these conditions can be checked simultaneously, for each arc, by first checking if the endnode is a part of $N_0$ and then see if the startnode is also a part of $N_0$. If both nodes for an arc in $A_1$ are a part of $N_0$, then this arc is saved together with all other arcs for which this is also true. The saved arcs for both SP-graphs are then compared to see if they are equal, and thus if condition A.4 is fulfilled. If, on the other hand, only the endnode is a part of $N_0$, then the algorithm can be cancelled immediately, since this means that the arc doesn't satisfy the third condition, and the two SP-graphs can't be combined. This is done in a function called `getCommonArcs`.

There are two things here that affects the complexity. First the STL-function `set_intersection` has a complexity of $O(|N|)$, and second the function `getCommonArcs`, and its complexity depends on $|A_L|$, since it looks at each arc (or until it finds an invalid arc). This means that the complexity for this part as a whole will be $O(|A_L|)$.
3.9 How to combine SP-graphs

Let's now continue with how the program uses the algorithm described right above.

There are two different modes here, the program can either combine SP-graphs in order for each SP-graph to be an out-graph (that is that all paths starts in the same node) or an in-graph (where all paths ends in the same node). This means that if I want the program to create out-graphs, then there will be problems if some of the SP-graphs that are sent to the function has several startnodes, and there will similarly be a problem if the SP-graphs has several endnodes and the program should create in-graphs. To solve this, the program will first divide any SP-graphs with several startnodes (endnodes) and then use those divided SP-graphs instead of the original ones, when trying to combine SP-graphs. The complexity for this will at most be $O(m \cdot |N| \cdot |A_L|^2 \cdot \log(|A_L|))$, and we can ignore this step when all incoming SP-graphs only has one set of path each.

The code for combining SP-graphs has been divided into three different parts, and it is possible to tell the program if it should just use the first step, the two first steps or all three steps, when the program starts. The number of combined SP-graphs will be decided in the first step, while the second and third step tries to add more sets of paths to each combined SP-graph, so that they will each connect more nodes. The first step tries to combine two SP-graphs with the same startnode (endnode), the second step tries to add the subpaths in the first SP-graph, from the startnode in the second SP-graph to the endnodes in the first SP-graph, and the third step tries to add subpaths from the second SP-graphs startnode to any unconnected node in the second SP-graph.

![Figure 18: Illustrating the steps to combine SP-graphs.](image)

The example above will illustrate the three steps. We can't combine the two SP-graphs above in the first step, because they have different startnodes. We will then try to add paths to the second/dotted SP-graph in step 2, by first look at $S_1(2,6)$, that is the subpath from node 2 to node 6 in the first SP-graph, and it is easy to see that this can be added. The algorithm will ignore the subpath $S_1(2,5)$, since the second SP-graph has already been connected to node 5. The algorithm will now move on to the third step, and the second SP-graph is now connected to all nodes except 1 and 4. The algorithm first tries to add $S_1(2,1)$, but this is empty, and then it tries to add $S_1(2,4)$ to the second SP-graph, which is successful. The right part of the image shows the combined version of the second SP-graph.

The program will try to combine SP-graphs that has the same startnode (endnode) in the first step. The SP-graphs that are sent to the function are called incoming, and will not be changed. The combined SP-graphs will also be known as outgoing, to mark that they are new and to be returned from the function. The first incoming SP-graph will automatically be added to the outgoing SP-graphs, that is it is the start of the first combined SP-graph. The program moves on to the second incoming SP-graph, and if it has the same startnode as the first added outgoing SP-graph, then the program will call the algorithm described above to decide if they can be combined. If they can't be combined or if the second SP-graph has another startnode, then it is added as the base for the second combined SP-graph. Then the program looks to see if the third SP-graphs can be combined with any outgoing SP-graph, or it is added at the end of the outgoing SP-graphs. If an incoming SP-graph can be combined with a outgoing SP-graph, then the program will directly start looking at the next incoming SP-graph. The program will look through all the incoming SP-graphs, and each one of
them will either be combined with another SP-graph, or added at the end.

All this means that the complexity for the first step will look like \(O(m \times m(\text{out}) \times |A_L| \times \log(|A_L|))\), where \(m\) is the number of incoming SP-graphs and \(m(\text{out})\) is the number of combined SP-graphs (and \(m(\text{out})\) is normally smaller than \(m\)). However this is a absolute worst-case-scenario, and we know that the algorithm that checks to see if the SP-graphs are combinable (and that has a complexity of \(O(|A_L|)\)) will only be called if both SP-graphs that are compared has the same startnode (endnode), and since we can expect any interesting and real-life-like example to generally be combinable and that the number of resulting combined SP-graphs will only have one or a few SP-graphs for each startnode (one in the case when all SP-graphs with the same startnode are subpath consistent), which points towards the real complexity to be more like \(O(m \times |A_L| \times \log(|A_L|))\). We don't know \(m(\text{out})\) in advance so the worst case should really be written as \(O(m^2 \times |A_L| \times \log(|A_L|))\).

Next is the second step. The algorithm will first check to see if the first combined SP-graph is spanned, that is connected to all nodes. If it is spanned, then there is no need to look for paths to add to the SP-graph. If not, then it will look through the original SP-graphs, for an endnode that the first combined SP-graph isn't connected to, let's call it e(in), while the startnode for the first combined SP-graph, let's call it s(out), is a part of the other SP-graph. The program will then try and get the subpath from s(out) to e(in) in the original SP-graph with the use of DFS and if the subpath exists, use the algorithm above to try and add the subpath to the first SP-graph. The program will continue to look through the original SP-graphs to find subpaths to add to the first combined SP-graphs until either the first SP-graph becomes spanned, or all the incoming SP-graphs are searched. Then the program will look for subpaths to add to the second combined SP-graph and the same for the rest of the combined SP-graphs. The reason that I only look at the endnodes of a set of paths in the second step is that this will reduce the number of times a new path will be added to a combined SP-graph, since only looking at the endnode will make sure that the longest possible part of the path is added first.

Then the program will look at unconnected nodes that isn't endnodes in the third step. It starts by looking for combined SP-graphs that doesn't yet span all nodes. The idea is then to, for each unspanned SP-graph, to find a set of paths from the startnode to each of the unconnected nodes, by searching through the incoming SP-graphs. The program then adds all sets of paths that it is allowed to combine according the the rules above, until either the combined SP-graph becomes spanned or the algorithm has searched all incoming SP-graphs, and then gets on to the next SP-graph that isn't spanned. One important thing is that the program has already tried to add those sets of paths that ends in an endnode for the incoming SP-graph and failed in step 2. The reason that they couldn't be combined, according to theorem 2, is that they are subpath inconsistent. This means that step 3 doesn't help if all SP-graphs are subpath consistent.

This leads to the complexity of the second and third step. The running time for step two is \(O(m(\text{out}) \times m \times \#\text{end} \times |A_L| \times \log(|A_L|))\), since the program will try each incoming set of paths on each new combined SP-graph and must use DFS to get the part of the incoming SP-graph that starts in the same node as the combined SP-graph. We also know that a lot of sets of paths could be rejected immediately (for instance if they end in a already connected node), and that the program will stop looking/not start looking when a combined SP-graph is connected to all nodes. This all means that the worst-case prediction will probably be very pessimistic, but it will be hard to make an accurate prediction of what to expect in the normal case, unlike step 1. For step 3 it looks like \(O(m(\text{out}) \times m \times (#\text{unconnected nodes(\text{out}))} \times |A_L|^2 \times \log(|A_L|))\). Here I used #unconnected nodes(\text{out}) instead of the more general \(|N|\), since the function will only use DFS and then check if the SP-graphs can be combined, for the nodes that are still not connected to the outgoing SP-graphs. Since a lot of the combined SP-graphs are already likely to be connected to all nodes after step two, the actual average complexity will be a lot smaller, but it is again very hard to give any specific number. The
complexity will be $O(m^2 \times |N| \times |A_L|^2 \times \log(|A_L|))$ for both step 2 and 3 when expressed only in known parameters.

### 3.10 Remove extra SP-graphs

One thing that I noticed when I was working on my program, was that a lot of the SP-graphs in the given examples, had SP-graphs that was a part of another SP-graph, and that it would be possible to remove these SP-graphs without making the program lose any information. Below is an example that shows what I mean:

![Diagram](image1)

*Figure 19: An example when one of the two SP-graphs doesn't add to the total amount of information*

The first SP-graph contains a set of paths from node 1 to node 6, and the second SP-graph a set of paths from node 2 to node 6. You can see that all arcs that forms the paths in SP-graph 2 is also a part of SP-graph 1 and that removing SP-graph 2 wouldn't really change the overall problem.

It would be interesting to test the effect of removing these unnecessary or redundant SP-graphs, since the running time for almost all functions depends on the number of SP-graphs in the problem (sometimes linear and sometimes in square). There is one problem left, because as the next example shows so is it a necessary, but not sufficient condition, that all arcs are a part of another SP-graph:

![Diagram](image2)

*Figure 20: One SP-graph is a part of the other, but has extra information*

This example is very similar to the last one, but is needed to make a significant difference extra clear. You can once again see that all arcs in the second SP-graph are also present in the first one. One big difference, however, is that the first SP-graph contains an extra subpath from node 2 to node 6 (or if you wish to node 5). We know that the SP-graphs can't have any valid weights, since it is subpath inconsistent. But if the program would remove the second SP-graph, then the rest of the data might not tell this, and the information about subpath inconsistency would be destroyed. This means that a second condition must be added, that $S_1(a,b) = S_2(a,b)$, that is that the two SP-graphs has the same subpath from node a to node b. We do know that all arcs in SP-graph 2 is also a part of SP-graph 1, which means that SP-graph 1 connects both the startnode and the endnode for SP-graph 2, so those should be used as a and b (since we could miss some subpath inconsistency if a smaller interval was used and we know that this is a valid interval).

This has then been implemented into two functions in my program, but the algorithm written only works for SP-graphs that contains just one set of paths. This because the number of SP-graphs has likely already been reduced, if they has been combined.

All this is implemented into two functions, one called `checkForRedundantSPGraphs` that takes two SP-graphs and checks if the first one is a part of the second one. One necessary condition
that I didn't mention earlier, since it's pretty obvious, but that can make the function faster is that the SP-graph that is suspected to be a part of another SP-graph must either have fewer total arcs than the second, or be an exact copy (i.e. both SP-graphs are identical and thus each has the same arcs as the other). Then the function uses a STL-function called includes, that checks if one sorted interval is a part of another sorted interval, and whose running time is $O(|A_L|)$ according to [7]. If the response is positive and the two SP-graphs has the same number of arcs, then they are identical and the first one can be said to be redundant. If the response is negative, the function exits directly. The third alternative is that the response is positive and that the first SP-graph has fewer arcs than the second. The algorithm will in this case use DFS to see if the arcs in the subpath in the second SP-graph, that starts in the first SP-graphs startnode and end in its endnode is identical to the arcs in the first SP-graph. The first SP-graph will then be removed if the subpath is identical to it.

The second function is called testOrEditExtraSPGraph and it can, as the name suggests, be used either to actually remove the extra SP-graphs or just to check how many extra SP-graphs there are. It would be smart to not check an SP-graph against an SP-graph that has already been deemed as redundant, and the easiest way to do this is to just delete the redundant SP-graph. There are however two problems with this, first it doesn't work when the function just checks how many extra SP-graphs that there are and secondly is that one drawback with using a vector is that the elements must come after each other in order in the memory. This means that if a element in the middle is removed, then all elements after it must be moved one step to fill up the hole that was created by deleting an element, which basically means that random deletion is to be considered linear running time and thus slows down the function. The solution that I had adopted instead is to have a second vector where the value of the element tells if the corresponding SP-graph is considered redundant and not check later SP-graphs against them. The function will then, after the search is finished, move the first non-redundant SP-graph to position 0, the second to position 1 and so on.

The running time for each iteration of checkForRedundantSPGraphs will at most be $O(|A_L|^2 \log(|A_L|))$ (that is the running time of DFS) and it will at most be called $m^2$ times from testOrEditExtraSPGraph. This gives a maximum total of $O(m^3 |A_L|^2 \log(|A_L|))$. There are however several reasons to expect that it will be lower. First will almost half of the comparisons have a constant running time, because of the numbers of arcs in each SP-graph and some comparisons will not be made since some SP-graphs have already been found as redundant. There will also be a lot of comparisons that only takes an average of $O(|A_L|)$ running times, since not all the arcs are included in the other SP-graph. The running times for SP-graphs with the same number of arcs is also $O(|A_L|)$ for each comparison. So let's assume that no SP-graphs are identical, and that all SP-graphs are subpath consistent. This means that if one SP-graph contains all the arcs of another SP-graph, then the second SP-graph is always redundant. In theory this means that there is no need to perform the DFS, but in practice it means that DFS is at most called $m^2$ times, and in reality a lot less (since the extreme case means that all SP-graphs except one was redundant). The number of times that it is necessary to call DFS will increase, when there are more subpath inconsistent SP-graphs. This means that the worst case-running time is probably quite pessimistic.

### 3.11 Check for subpath consistency

I will here first give a general algorithm, that can be used to check if two SP-graphs are subpath consistent and then explain how the algorithm is used and the general complexity. The algorithm calls the two SP-graphs $A_1$ and $A_2$, and it looks like:

1. **Startup:** Save all nodes that are connected to both $A_1$ and $A_2$ and call them $\hat{n}$.

2. **Choice of node-pair:** Choose two nodes in $\hat{n}$, that hasn't yet been compared, and call them $a$ and $b$. Go to step 8, if there are less than two common nodes, or all node-pairs have already been searched.
3. **Get the subpaths for** $A_1$:

   a. Extract $S_1(a,b)$ from $A_1$. If $S_1(a,b) \neq \emptyset$, then set $S_1 = S_1(a,b)$ and go to step 4a.

   b. Get $S_1(b,a)$. Go to step 2 if $S_1(b,a) = \emptyset$ (Since this means that there is neither a subpath from $a$ to $b$ or from $b$ to $a$ in $A_1$) and else set $S_1 = S_1(b,a)$ and go to 4b.

4. **Get the subpaths for** $A_2$:

   a. Get $S_2(a,b)$. Go to step 2 if $S_2(a,b) = \emptyset$ (this means that one SP-graph has a subpath from $a$ to $b$ and the other from $b$ to $a$) and else set $S_2 = S_2(a,b)$ and go to 5.

   b. Get $S_2(b,a)$. Go to step 2 if $S_2(b,a) = \emptyset$. Else set $S_2 = S_2(b,a)$.

5. **Compare subpaths**: Check if $S_1 = S_2$. Go to step 7 if it is false.

6. **Reduce nodes**: Remove all nodes that is connected to $S_1$ (and thus also $S_2$) from $\tilde{n}$, except $a$ and $b$. Go to step 2.

7. **The SP-graphs are subpath inconsistent**: The algorithm has found that the SP-graphs are subpath inconsistent between the current $a$ and $b$, and is terminated. There isn't any reason to use the algorithm for any other SP-graph-pair in the same problem, since it is enough to find one pair of subpath inconsistent SP-graphs.

8. **The SP-graphs are subpath consistent**: The algorithm is terminated, since all node-pairs in $\tilde{n}$ has been searched, and the two SP-graphs are subpath consistent because all the node-pairs have been subpath consistent. The algorithm can then be run with another SP-graph-pair to see if they too are subpath consistent.

If we would want to know if all SP-graphs are subpath consistent, then we would have to run the algorithm once for every SP-graph-pair, or at least until we would find a pair that was subpath inconsistent. The complexity of the algorithm will clearly depend on the size of $\tilde{n}$ squared (since it needs to check each pair of common nodes). One problem is that the algorithm will sometimes change the size of $\tilde{n}$ when it runs, and I will for this reason define $\#CN$ as the number of common nodes for two SP-graphs, that is $\#CN = |\tilde{n}|$ after the first step in the algorithm is complete, so the complexity will at least be $O((\#CN)^2)$. Another important aspect is the complexity of getting a set of subpaths from a SP-graph. I has used a modified version of DFS here, with a complexity of $O(|N| \cdot |A_L|^2)$ (it doesn't save the new sets of paths, just which of the arcs are used). The DFS is performed at most three times for each pair of common nodes, so the total complexity will in this case be $O((\#CN)^2 \cdot |N| \cdot |A_L|^2)$.

We are interested to know if all SP-graphs are subpath consistent of not. This means that the program will use the algorithm for each SP-graph-pair, and will also stop checking SP-graph-pairs when it finds a subpath inconsistent pair, since it is not that interesting to see how many pairs are subpath inconsistent. The first step is done by the STL-function `set_intersection`, which gets the nodes that the first SP-graph is connected to and the nodes that the second SP-graph is connected to, and returns the nodes that both SP-graphs are connected to. The program looks through the elements in $\tilde{n}$, by first looking at the first and second element, then the first and third element. I also tested to have it start looking at the first and the last element, then the first and the second to last element and so on, but there wasn't any significant difference between the two methods.

I remove some nodes from $\tilde{n}$ in step 6 in the algorithm. Why is this possible? Let's say that the subpaths from $a$ to $b$ are the same for the two SP-graphs and that $c$ is a node between $a$ and $b$. Now
if node d is before a, then there exists a subpath from d to a, and a subpath from d to c and finally a subpath from d to b. If one of the SP-graphs contains the arc (d, c), then this arc will be a part of the subpath from d to b, but not in the subpath from d to a, and d to b will contain all arcs in the subpath from d to c and then some arcs. Similarly if e is after b, then a to e will cover all arcs in c to e and then some extra arcs. This means that if the subpaths from d to c are subpath inconsistent, then the subpaths from d to b will also be subpath inconsistent, and we can safely remove node c from ñ. Below is an short example of how the program works:

First, the common nodes are saved to ñ, so that ñ={3, 4, 5, 6, 7} and the program will compare them in order, so it starts by node 3 and 4. It calls DFS to get the subpath from node 3 to node 4 in the first SP-graph, and then DFS to get the same subpath in the second SP-graph. The two subpaths both contains the same four arcs, as can easily be understood by looking at the image above. This means that the program doesn't have to check the nodes in the middle, so those nodes can be removed from ñ, that now looks like ñ={3,4,7}. This means that the program will now go on to check from node 3 to node 7, and the two SP-graphs are subpath consistent here too, which will remove node 4, so that ñ={3,7} and there is no node pair left to search, so these two SP-graphs are subpath consistent. One thing that it can be interesting to notice, is that there isn't any subpath between node 5 and 6, even though they are both (initially) a part of ñ.

The running time for all this will be O(m²*(#CN)²*|N|²*|A|²), where m² comes from that each pair of SP-graphs will in the worst-case-scenario be searched through with the algorithm described above, and the rest is the complexity to look through one SP-graph-pair. One interesting thing is that if all SP-graphs contains just one set of paths, then m will be larger, but #CN and |A| will both be smaller. If the function instead gets SP-graphs that are combined to in-graphs or out-graphs, then m will be low, while both #CN and |A| will be high. The complexity is thus O(m² * |N|² * |A|²) in known terms.

3.12 Changing SP-graphs

There is also a function, called changePaths, in the program, that makes a change in a path in a random SP-graph. My purpose for creating this function was to try and create SP-graphs, that were subpath consistent, but that still had valid cycles. It is at least sometimes successful in this as you will see later. All this means that it is only useful to use this function when the example that we are starting with is one that doesn't have any valid cycles, because they will usually not remove any valid cycle that is there from the beginning (it could happen sometimes, but most likely not very often).

First it selects a random SP-graph, rindex, then a random unconnected node, rnew, and then a random connected node to replace, rold. There are a few requirements to the selection of these variables. First rindex must be an SP-graph with at least two arcs and one unconnected node. For a more random rindex, it is best if all or most of the SP-graphs can be chosen. Also the more unconnected nodes in a specific SP-graph, the more choices for rnew, so this function works best for SP-graphs that only has one set of paths each. A second demand is that rold has exactly one incoming arc and one outgoing, that is rindex will always be in the first case below. The idea behind this is that if it is in the first (and most simple) case, then the rest of the SP-graphs will hopefully have a greater chance of also being in case 1, and less likely to be in the rest of the cases, and specifically that case 4 will be rare.
When the variables are determined, the program changes so that the shortest path from the node before \textit{rold}, \textit{start}, to the node after \textit{rold}, \textit{end}, will now instead be (\textit{start}, \textit{rnew}) and (\textit{rnew}, \textit{end}). There are four possible cases that needs to be considered:

1. **\textit{rold}** has one incoming and one outgoing arc. Here the program simply replaces the arc (\textit{start}, \textit{rold}) with (\textit{start}, \textit{rnew}) and (\textit{rold}, \textit{end}) with (\textit{rnew}, \textit{end}) and finally updates the data for the SP-graph.

2. **\textit{rold}** has one incoming arc and several outgoing arcs. In this case I have chosen to let the program remove (\textit{rold}, \textit{end}) and keep (\textit{start}, \textit{rold}) (in order to keep the rest of the arcs starting in \textit{rold}) and of course add (\textit{start}, \textit{rnew}) and (\textit{rnew}, \textit{end}).

3. **\textit{rold}** has several incoming arcs and one outgoing. Naturally the opposite of case 2, keep (\textit{rold}, \textit{end}) and remove (\textit{start}, \textit{rold}).

4. **\textit{rold}** has several incoming and several outgoing arcs. Here the program replaces both (\textit{start}, \textit{rold}) and (\textit{rold}, \textit{end}) (although keeping one of them is also possible).

![Figure 22: Illustration of the four cases for changing an SP-graph](image)

Above is an example that illustrates the four cases, node 2 is \textit{start}, node 3 is \textit{rold}, node 5 is \textit{end} and node 4 is \textit{rnew}. The unbroken lines here represents the old arcs in the SP-graph and the dotted lines are the arcs that are in the modified SP-graph.

There is no theoretical reason for changing the SP-graphs one way or another. My thought here was that the changes should probably be such that they won't affect the problem too much, so most of the arcs in the original SP-graph are left in the modified one, and it will therefore at most remove two arcs (in this step). An alternative change would be to change all arcs that ends in \textit{rold}, to instead end in \textit{rnew}, and change all arcs that starts in \textit{rold}, so they would now start in \textit{rnew}. Yet another way would be to remove all other paths in the SP-graph that connects to \textit{rold}. I haven't tested any of these methods, since this one seems to work.

The way this works is that the function will look through all the SP-graphs to see if they contains both of the arcs (\textit{start}, \textit{rold}) and (\textit{rold}, \textit{end}) and will then determine what case that SP-graph
It is possible that some of the SP-graphs, that needs to be updated, are already connected to the node \( r_{\text{new}} \). In these cases the program does a DFS-search from \( r_{\text{new}} \) to \( r_{\text{new}} \) to see if there is a cycle, and removes all arcs in the cycle (if any). It is possible that more than two of the original arcs needs to be removed when we have a cycle. Here is an example of when we get a cycle:

![Figure 23: Where changing the SP-graph causes a cycle to appear](image)

Node 1 is \( \text{start} \), node 2 is \( \text{rold} \), node 3 is \( \text{end} \) and node 4 is \( r_{\text{new}} \). Here the unbroken lines notes the old arcs in the SP-graph and the dotted lines are the new arcs that has been inserted in the SP-graph. This can't be \( r_{\text{index}} \), since one demand there was that \( r_{\text{new}} \) was an unconnected node. The left part shows the original SP-graph, the middle part how the SP-graph looks after the old arcs has been removed and the new arcs added, in accordance with case 1 above. It then uses DFS to look for a set of paths from node 4 and back to node 4, here it finds the arcs (3,4) and (4,3) so these are removed from the SP-graph, and the part on the right shows how the final SP-graph looks, with one old arc and one new arc.

We know that \( r_{\text{index}} \) wasn't connected to \( r_{\text{new}} \) before, so this must mean that the subpaths from \( \text{start} \) to \( r_{\text{new}} \) contains just one arc, (\( \text{start}, r_{\text{new}} \)) and similar for the subpaths from \( r_{\text{new}} \) to \( \text{end} \). There could be other subpaths from \( \text{start} \) to \( r_{\text{new}} \) in other SP-graphs, but if there were no cycle among the SP-graphs that we started with, then there will only be conflicting subpaths between either \( \text{start} \) and \( r_{\text{new}} \) or \( r_{\text{new}} \) and \( \text{end} \) (otherwise the original SP-graph would have had a conflicting path from \( \text{start} \) to \( \text{end} \)). This means that the final step for the algorithm is to check for cases where there is conflict with \( r_{\text{index}} \), with the use of DFS, and then replace the conflicting set of paths with the arc that \( r_{\text{index}} \) uses (either the set of paths from \( \text{start} \) to \( r_{\text{new}} \) is replaced by \( \text{start}, r_{\text{new}} \)) or the set from \( r_{\text{new}} \) to \( \text{end} \) by \( r_{\text{new}}, \text{end} \)). It is possible that these changes can cause conflicts with other SP-graphs, but the program will not check to see if this is the case or not, because there can be a new conflict for each old that has been fixed. Now the program only checks each SP-graph to \( r_{\text{index}} \), and the function would be more complex if it also needed to check each SP-graph against each newly changed SP-graph.

The complexity is \( O(m^*|A_l|^2*\log(|A_l|)) \), since it will in the last step look through all SP-graphs and use DFS to get the set of paths from \( \text{start} \) to \( r_{\text{new}} \) and DFS has a complexity of \( O(|A_l|^2 * \log(|A_l|)) \).

### 3.13 Check valid cycles

I use an external program to check for valid cycles. It's written by Peter Broström and my supervisor Kaj Holmberg and called VC1 in [5]. I modified the main function a little bit, so that the program would take arguments from the command line instead of from a separate file, in order to be able to call this program directly from my own program. The negative thing with this solution is that all SP-graphs must be transferred to a file in order to be checked for valid cycles, since it is a separate program. The complexity for this program is \( O(m^*|A_l|+|N|) \) according to [5].

### 3.14 Summary

I have here described how I have constructed my program. It reads a problem from a file and can then modify it in several ways. The most important, as we shall see in the results is to combine SP-
graphs into either in- or out-graphs, but it can also divide SP-graphs (the opposite of combining SP-graphs) and remove SP-graphs that are a part of another SP-graph (and thus doesn't add anything to the problem). It can also check the problem to see if it is subpath consistent or use an external program to check if it has an 1-valid cycle. It can finally change a random path in the SP-graphs, which I has used to create examples that will only has 1-valid cycles after the SP-graphs has been combined (more about this in chapter 5).

3.15 List of complexities

Here follows a list of complexity for the different operations:

Divide SP-graphs: \( O(m^*|N|^2*|A_L|^2*\log(|A_L|)) \)

Combine SP-graphs, 1 Step: \( O(m^*|A_L|^2*\log(|A_L|)*(|N|^*|A_L| + m)) \)

Combine SP-graphs, 1 Step (Input Already divided): \( O(m^2*|A_L|^2*\log(|A_L|)) \)

Combine SP-graphs, 2 Steps: \( O(m^2*|N|^*|A_L|^2*\log(|A_L|)) \)

Combine SP-graphs, 3 Steps: \( O(m^2*|N|^*|A_L|^2*\log(|A_L|)) \)

Remove Extra SP-graphs: \( O(m^2*|A_L|^2*\log(|A_L|)) \)

Check For Subpath Consistency: \( O(m^2*|N|^3*|A_L|^2) \)

Change Arcs In SP-graphs: \( O(m^2*|A_L|^2*\log(|A_L|)) \)

Check For Valid Cycle: \( O(m^2*(|A_L|+|N|)) \)

With the following definitions used:

\( m = \) Total number of incoming SP-graphs
\( |A_L| = \) The highest number of arcs in a SP-graph
\( |N| = \) The total number of nodes that is a part of a problem

3.15 List of complexities - 36 - Erik Haraldsson, 2008
Chapter 4: Mechanisms of the Program

4.1 Menu

The program can be controlled either by choosing an alternative in a menu or by giving arguments when starting the program (see below). When the program starts, it will first ask for the datafile that you wishes to use, unless already specified in an argument. Then the program shows the main menu:

Choose an alternative!
1. Submenu for current SP-graphs
2. Divide SP-graphs
3. Combine SP-graphs to in-graph
4. Combine SP-graphs to out-graph
5. Change current data
6. Remove redundant paths
7. Change a path
8. Test subpath consistency
0. Exit

Some comments about the various options:

**Alternative 1** leads to a sub-menu, that looks like:

Choose an alternative!
1. Print startnodes/endnodes for all SP-graphs on screen
2. Print a specific SP-graph to screen
3. Print all SP-graphs to screen
4. Print all SP-graphs and their in-/out-degree on screen
5. Check if the SP-graphs can be divided
6. Save all SP-graphs to a file
7. Test current data for redundant SP-graphs
8. Check for valid cycles (External Program!)
0. Main menu

Here alternative 1 to 4 displays various amount of information about the SP-graphs to the screen, alternative 5 shows how many of the SP-graphs are in-graphs, out-graphs or just a set of paths, option 6 gives the opportunity to write the resulting SP-graphs to a new file, so that the modified data can be saved, option 7 tests how many SP-graphs can be removed without changing the problem, option 8 launches an external program that tests if there is a valid cycle in the current problem and option 0 makes the program return to the main menu.

**Alternative 2** returns a version of the SP-graphs, where every graph has one startnode and one endnode. If the data that the program starts with has one SP-graph from origin $o$ to two destinations $d_1$ and $d_2$, then this alternative will return two SP-graphs, one from $o$ to $d_1$ and the second from $o$ to $d_2$. Option 5 from the sub-menu in alternative 1 uses the same algorithm, but only tells how many SP-graphs that are divided, how many that are in-graphs, out-graphs or something more complex.

**Alternative 3** and **alternative 4** will try and create in-graphs and out-graphs respectively, from the selected indata. This can be seen as the inverse of alternative 2.

**Alternative 5**: The program is designed so that alternatives 2-4 doesn't destroy the version of the
SP-graphs that are sent to be processed by those functions, but they instead return a new version of the SP-graphs. Here you can change which version of the SP-graphs that the program will use as indata when it calls a function. So if you create an in-graph and want to save it, then you chooses alternative 5 and then option 3 for in-graph. Now you can choose alternative 1 and option 6 to save the created in-graph, instead of the original SP-graph. The sub-menu simply looks like:

Set current data to:
1. Original SP-graph
2. Divided SP-graph
3. In-Graph
4. Out-Graph
0. Go to main menu without changing current data

Alternative 6 checks to see if any SP-graph is a part of another SP-graph, and then removes all the redundant SP-graphs. Alternative 1 sub-option 7 does the same test, but doesn't remove any SP-graphs, and instead just gives the number of redundant SP-graphs. This alternative will only work correctly when the SP-graphs are divided. Unlike alternative 2-4 this function applies the result directly to the currently selected SP-graphs.

Alternative 7 will change a path in a random SP-graph, and then modify the rest of the SP-graphs in order to try and find a problem that is subpath consistent and still has a valid cycle.

Alternative 8 checks all SP-graphs to see if they are subpath consistent. This algorithm will abort immediately if it finds two SP-graphs that aren't subpath consistent.

Alternative 9 exits the program. The way the program is set up, you will need to exit and then restart it in order to choose a new datafile to work from (since the whole program is command line, the time it takes to restart it is negligible).

4.2 Arguments

In order to easier control the program and repeat the same actions on several files, it has several arguments/flags that can be used when starting the program from the command line:

- **-f <file>** This command tells the program to use the next argument, `<file>`, as the file to read indata from, instead of asking for a file inside the program.

- **-w <file>** Like -f, but gives the name of a file to write modified data to. However, you must still manually direct the program to alternative 1 sub-option 6 (or by -p), and the program will then check if you have named any specific file to save the data to.

- **-p <numbers>** It controls how to navigate through the menus/options of the program by using a sequence of numbers to operate the program, e.g. 40 will first combine SP-graphs to create out-graphs and then exit the program. Naturally the sequence should be exactly the same as one would have used to manually navigate through the menus, and the numbers can of course also be used in the sub-menus. The numbers are stored one and one in a STL-queue-container (here a queue works, just like in real life, as First-In-First-Out (FIFO)), so if the sequence 4541600 is entered, then the program will first do commando 4 then commando 5 then 4 and so on. Here it means to first combine SP-graphs into out-graphs and then select the newly created out-graphs and then print them to a file and finally exit the program. Since the use of -p implies that you know what you want to do, the menus are hidden as long as the queue isn't empty, so there is usually no need to use -m when you use -p.
-m  It will make the program hide the menus, and by that make more efficient use of the screen space.

-s <0/1/2>  This command tells the program how many extra steps to take to find complete out-graphs or in-graphs. More steps will take a longer time, but will make the in-graphs or out-graphs more complete. The default value is 2, i.e. by default it will execute all available steps. This is the only way to change the number of steps used.

-h  This command shows a condensed help text of the commands and then exits the program immediately.

It doesn't really matter in what order the commands comes, since they are used to control different aspects of the program (unless you for some strange reason use the same command twice, which is not at all recommended).

4.3 Information

The different functions of my program will print the interesting results and information to the screen. It is possible to get this information on a textfile instead by calling the program with:

```
./modifySP [arguments] > myresultfile.txt
```

This makes everything that the program normally prints to the screen instead go to the file myresultfile.txt (at least in Linux), and it is a good idea to disable the menus or they will also be in the file instead of on the screen, by either using `-p <numbers>` to control what the program will do or `-m`.

When the program divides the SP-graphs or when it removes extra SP-graphs from the problem, then it will print how the selected function has affected the number of SP-graphs in the problem. When it checks for subpath consistency so will it print the vector indexes for the first two SP-graphs that was subpath inconsistent or let us know that all SP-graphs were subpath consistent. The output for changing a random path looks like:

<table>
<thead>
<tr>
<th>Index to change in</th>
<th>Previous Node</th>
<th>Old Node</th>
<th>Next Node</th>
<th>New Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>11</td>
</tr>
</tbody>
</table>

This means that it is easy to go back to the original data and see what the main change was, and more importantly, save this so that we can see that the program doesn't randomly change the data the same way twice. It should be interpreted as that the arcs (1,9) and (9,4) has been changed to (1,11) and (11,4). The output for combining SP-graphs will look like (when `-s 0` is used):

We started with 1089 SP-graphs
and ended up with 48 SP-graphs
Number of SP-graphs that span's all nodes: 3
Number of nodes: 50
Average number of nodes/SP-graph: 32
Number of SP-graphs which is only connected to half or less of the nodes: 12

Part 1:
44337 checks.
1041 successful compares.
1 unsuccessful compares.

Or (when `-s 1` is used):

We started with 1274 SP-graphs

Erik Haraldsson, 2008
and ended up with 49 SP-graphs
Number of SP-graphs that span's all nodes: 48
Number of nodes: 50
Number of nodes/SP-graph (if less than max): 35

Part 1:
55053 checks.
1225 successful compares.
1 unsuccessful compares.

Part 2:
46994 checks.
628 successful compares.
15 unsuccessful compares.

The only difference between \texttt{-s 1} and \texttt{-s 2} is that in the latter case, so is there also some data for Part 3. It starts with how many SP-graphs the problem had from the beginning and how many it has after being combined. Then it says how many of the SP-graphs that is connected to all the nodes, and then the number of nodes, \(N\). We have in these two cases that \(N=50\) and we can see that \texttt{-s 1} tends to have a lot more spanned SP-graphs than \texttt{-s 0}. The program will then, for \texttt{-s 1} and \texttt{-s 2}, print the number of nodes connected to a SP-graph for those SP-graphs that doesn't span all nodes. This would normally not be useful for \texttt{-s 0}, since there will then normally only be a few SP-graphs that spans all nodes. Here I thought that it would instead be useful to see the average number of nodes connected and how many of the SP-graphs are only connected to half or less of the nodes, to give an idea of how effective it is on different examples. Then comes the specific information for each part. The number of checks tells how many times the program checks to see if two SP-graphs has the same startnode (or in the second and third part, how many times it checks to see if the startnode for one SP-graph, is a part of the other SP-graph). The second number is the total times that the SP-graphs has the same startnode and that they can be combined, while the third number is when they has the same startnode, but can't be combined. According to Theorem 2, this means that if the third number is not zero, then at least two of the SP-graphs will be subpath inconsistent.

The important line that we get from the external program that checks for a valid cycle is:

Valid cycle for SP-graphs 4 and 305: 11 13 19 15

The first two numbers are the indexes for the SP-graphs with a valid cycle, and the numbers in the end represents the nodes that are in the cycle. Here one cycle goes from node 11 to 13 to 19 to 15 and back to 11 and the other cycle goes around the nodes in the opposite direction.
Chapter 5: Computational tests and results

I will first talk a little about how I used the algorithm to modify SP-graphs described earlier, and then describe how I tested some of the functions above, and show the resulting average running time for different problems and different algorithms and finally draw some conclusions from my work.

5.1 Generating modified SP-graphs

I only used SP-graphs without any valid cycles, and tried to modify them in such a way that I would get a resulting SP-graph that was subpath consistent, and yet had a valid cycle when an in-graph or an out-graph was created. The original data consisted of SP-graphs that only had one set of paths each and the command that I used was:

```
./a.out -f <current_indata-file> -p 67845418600 -w <new_file>
```

Or, for smaller examples (N<=20):

```
./a.out -f <current_indata-file> -p 7845418600 -w <new_file>
```

What this means is that I, for bigger examples, first removed redundant SP-graphs, to speed things up, and then used the method described earlier to modify a random SP-graph. The modified problem was then tested to see if it was subpath consistent. Then the program combined the modified SP-graphs to out-graphs, and tested if those out-graphs had a valid cycle. The modified example was permanently saved if it was both subpath consistent and had a valid cycle for the combined SP-graphs.

Two of the saved modified SP-graphs can be seen in figure 5 (in a slightly modified form, by removing some uninteresting arcs and nodes and renaming a few nodes, to make them go from 1 and up) and 6 (unmodified).

I would estimate that about one in every 20-50 modifications generated this sort of examples, but I didn't bother to count all the unsuccessful modifications. It could be interesting to know that almost all unsuccessful modifications failed because the modified version was subpath inconsistent (except for when |N|=10). The result was 20 of these subpath consistent examples with valid cycles for each of |N|=10, 15, 20, 30, 40, 50, 90.

5.2 Information about the examples used

The first group of examples are called RAND in [5], and are based on examples that have been generated by first setting a weight to each arc and then modify one arc, according to either the RC or the TR method. These examples are created for |N|=10, 15, 20, 30, 40, 50 or 90 (where |N| is the total number of nodes available in the examples according to the definitions in chapter 3), and there are between 40 and 60 examples for each |N|. The version used below has already been divided so each SP-graph only contains one set of paths. They are created from in-graphs, which means that all SP-graphs will always span all nodes after being combined by only using the first step if we try to re-combine them to in-graphs. I have for this reason, chosen to instead combine them to out-graphs in the tests below.

The second group are the modified SP-graphs described in 5.1 above. They were created from the first group and have 20 examples for each value of |N| above. These examples are divided from the out-graphs created when finding them, so they will instead be combined into in-graphs below.

The third group are modifications of some real world problems, called Cost266, Di-Yuan, France,
Germany50 and Pioro40. They are all originally from [8], and [5] shortly describes how the RC, RM1, RM5, RND and TR modifications are made. I had to manually discard some of these examples from the beginning because they had a cycle in at least one SP-graph. This group is divided from in-graphs, like the first group, so these are also combined into out-graphs below.

Here is a table that shows some the average number of SP-graphs in a problem (m), and the average number of arcs in a SP-graph (|A|) for both the divided version and the version combined to out-graphs with only the first step used (called C(0) later), as well as the average number of endnodes for the combined SP-graphs (it is always 1 for the divided SP-graphs). It also shows the number of SP-graphs left after all redundant SP-graphs has been removed (Rem).

| Examples     | Group | |N| | # Ex. | Divided (average) m(div) | |A| | Rem (average) m(rem) | m(comb) | |A| | #end |
|--------------|-------|------------|--------|--------|-----------------|-------|---------|----------------|-------|--------|------|
|              | RAND  | 10         | 40     | 60     | 2,1             | 42    | 10      | 7,8           | 4,2   |
|              | RAND  | 15         | 40     | 115    | 2,9             | 69    | 15      | 10,2          | 4,6   |
|              | RAND  | 20         | 40     | 210    | 3,2             | 127   | 19      | 14,2          | 6,6   |
|              | RAND  | 30         | 40     | 423    | 4,1             | 228   | 29      | 20,1          | 8,0   |
|              | RAND  | 40         | 53     | 760    | 4,8             | 400   | 39      | 27,9          | 10,2  |
|              | RAND  | 50         | 54     | 1107   | 5,7             | 560   | 48      | 33,5          | 11,6  |
|              | RAND  | 90         | 59     | 3406   | 8,2             | 1560  | 87      | 58,4          | 17,9  |
|              | Cost266| 37         | 273    | 498    | 6,3             | 215   | 37      | 24,0          | 5,8   |
|              | Di-Yuan| 11         | 298    | 72     | 2,3             | 57    | 12      | 10,5          | 4,6   |
|              | France | 25         | 225    | 307    | 4,4             | 155   | 23      | 18,3          | 6,9   |
|              | Germany50| 50        | 171    | 878    | 7,4             | 356   | 45      | 31,0          | 8,0   |
|              | Pioro40| 40         | 294    | 585    | 6,1             | 303   | 50      | 24,9          | 6,1   |

We can see that m(rem) is on average about half of m(div), while m(comb) is approximately the same size as |N|. This makes sense, and means that most SP-graphs has an unique origin (when creating out-graphs). It can also be seen that |A| increases when the SP-graphs are combined. The table above only shows the combined SP-graphs after the program has run the first step, and we know that the second and third step won't affect m(comb), but that they will most likely increase |A| further.

### 5.3 Introduction to running the program

All tests were run on my personal computer with an AMD Athlon XP 2200+ processor and 640 MB memory and Ubuntu 7.04 (Linux) as the OS. The computer was rebooted before each test session, and was running in terminal-mode (i.e. I killed all the fancy graphics before running any test). All this to make sure that the conditions stayed roughly the same.

All times that I have listed below are processor clock times, which tell how much processor time the program needed, as opposed to real time. It was measured with the command `time`. While C++ has an internal way to measure clock time, it can't measure the total time over several runs and has a precision of 0.01 seconds compared to `time`'s approximately 0.004 seconds and the C++ version doesn't count the time for when you start another program from your program (i.e. to check for valid cycles). It is also possible to minimize rounding errors in `time`, by getting the total time of running the same problem several times, and get the total time and then divide by the number of times run to get the average time. This all made me believe that using `time` was the best way to go.
I use the program g++ to compile my program, with the flag -O 3, that tells the compiler to try to add a lot of optimization to the code in order for it to run faster.

Here is some guidance on how to interpret the headlines for the tables below, |N| is the number of nodes, and #Ex. is the number of examples. C(0)+VC means that the program has first tried to combine the SP-graphs with the flag -s 0, that is by just using the first step, as described above and then testing the combined SP-graphs to see if there is any valid cycles. C(1)+VC is basically the same thing, but with the flag set to -s 1 instead. Rem+C(0)+VC means that the program has first removed the redundant SP-graphs and then tried to combine them and check for valid cycles as described above. Of course, VC means that it just check for valid cycle and Rem+VC that it removes redundant SP-graphs and checks for a valid cycle. Finally Rem+SpC means that it has removed extra SP-graphs and then checked to see if all SP-graphs were subpath consistent. In all cases were each problem first run 5 times and time measured the total time for all the problems in a specific group, so the average time was given by total time/(5*#Ex).

Here is a table of the complexity for the methods used:

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(0)+VC</td>
<td>(O(m^2</td>
</tr>
<tr>
<td>C(1)+VC</td>
<td>(O(m^2</td>
</tr>
<tr>
<td>Rem+C(0)+VC</td>
<td>(O(m^2</td>
</tr>
<tr>
<td>VC</td>
<td>(O(m^2(</td>
</tr>
<tr>
<td>Rem+VC</td>
<td>(O(m^2</td>
</tr>
<tr>
<td>Rem+SpC</td>
<td>(O(m^2</td>
</tr>
</tbody>
</table>

The table can be a guideline, but it doesn't take into effect that some of the functions changes the size of m and |A_l|.

### 5.4 Results from running the program

The first table below shows the resulting average runtime (in seconds) for the examples in RAND. I have not divided the problem based on whether the RC or TR modification was used here, because there didn't seem to be any big difference between them. About half of the examples here have valid cycles for |N|=10, and almost all have valid cycles for when |N| is bigger:

| |N| # Ex. | C(0)+VC | C(1)+VC | Rem+C(0)+VC | VC | Rem+VC | Rem+SpC |
|---|---|---|---|---|---|---|---|
| 10 | 40 | 0,0045 | 0,0044 | 0,0046 | 0,0020 | 0,0048 | 0,0037 |
| 15 | 40 | 0,0053 | 0,0065 | 0,0059 | 0,0052 | 0,0068 | 0,0057 |
| 20 | 40 | 0,0078 | 0,0085 | 0,0084 | 0,0137 | 0,0127 | 0,0125 |
| 30 | 40 | 0,0127 | 0,0172 | 0,0165 | 0,0917 | 0,0453 | 0,0406 |
| 40 | 53 | 0,0227 | 0,0320 | 0,0370 | 0,4255 | 0,1649 | 0,1535 |
| 50 | 54 | 0,0354 | 0,0534 | 0,0679 | 1,0626 | 0,3835 | 0,3305 |
| 90 | 59 | 0,1384 | 0,2982 | 1,0310 | 19,5452 | 5,9114 | 4,1757 |

The second table consists of the result from the subpath consistent examples with a valid cycle, that were generated as described in 5.1:
The first and most obvious conclusion is that combining SP-graphs into in-graphs or out-graphs seems to have a huge impact of the total time that it takes to check for valid cycles. This is not surprising, because we saw in 5.3 that only the methods that starts with combining SP-graphs are linearly dependent on m(div), while the rest depends on m(div) square. Once the SP-graphs have been combined, the VC algorithm instead depends on m(comb), which is much smaller, and it will thus take a lot less time to find any valid cycles. One important aspect of combining SP-graphs is that it speeds up the valid cycle-algorithm more for larger |N| and examples that are without a valid cycle (see Appendix B for a division of the time between examples with and without a valid cycle), that is the speed-up is larger for the problems that takes the longest to solve with just VC. It also seems that the method Rem+VC is preferable to just VC when |N| is larger than 20. This can seem a bit strange, since the complexity for Rem+VC is $O(m(\text{div})^{2*|A_L|^2*\log(|A_L|)} + m(\text{rem})^{2*|A_L|+|N|})$, which is quite a bit larger than the complexity for just VC. One explanation may however be that once an SP-graph has been marked as redundant, then no other SP-graph will compare against that one and that the complexity for each compare is only $O(|A_L|^2*\log(|A_L|))$ when one SP-graph is a part of another SP-graph. There is however no gain to first remove redundant SP-graphs when we want to combine SP-graphs, which is as expected.

It is also interesting to compare the last two columns, since both checking for valid cycles and checking for subpath consistency will most of the time give the same result when all SP-graphs contains of one set of paths. Both of these algorithms will also abort when a valid cycle or subpath inconsistent SP-graph-pair has been found. The algorithm that checks for subpath consistency is slightly faster in both tables above (but that is not always the case as seen in the tables below). This result is somewhat surprising since the complexity is much lower for VC than SpC. It is however possible to increase the performance for VC (by combining SP-graphs as seen above), but Rem+SpC is the most efficient way to check for subpath consistency. This because combining SP-graphs will both increase the number of arcs and the number of connected nodes in each SP-graph, and thus the number of common nodes, which will bring down efficiency for SpC.

Below are five tables for the modified real world examples, that were in group three in 5.2. The first table is for Cost266, a network with 37 nodes:

<table>
<thead>
<tr>
<th>[N]</th>
<th># Ex.</th>
<th>C(0)+VC</th>
<th>C(1)+VC</th>
<th>Rem+C(0)+VC</th>
<th>VC</th>
<th>Rem+VC</th>
<th>Rem+SpC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>0,0039</td>
<td>0,0049</td>
<td>0,0045</td>
<td>0,0030</td>
<td>0,0052</td>
<td>0,0034</td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>0,0052</td>
<td>0,0058</td>
<td>0,0064</td>
<td>0,0066</td>
<td>0,0071</td>
<td>0,0064</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>0,0064</td>
<td>0,0078</td>
<td>0,0083</td>
<td>0,0272</td>
<td>0,0178</td>
<td>0,0208</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
<td>0,0120</td>
<td>0,0142</td>
<td>0,0154</td>
<td>0,1749</td>
<td>0,0678</td>
<td>0,0650</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>0,0214</td>
<td>0,0279</td>
<td>0,0352</td>
<td>0,7864</td>
<td>0,2955</td>
<td>0,2190</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>0,0330</td>
<td>0,0480</td>
<td>0,0658</td>
<td>2,3516</td>
<td>0,7693</td>
<td>0,5626</td>
</tr>
<tr>
<td>90</td>
<td>20</td>
<td>0,1278</td>
<td>0,2495</td>
<td>1,2353</td>
<td>59,3881</td>
<td>15,7279</td>
<td>9,1692</td>
</tr>
</tbody>
</table>
Here is the table for Di-Yuan, a network with 11 nodes:

<table>
<thead>
<tr>
<th>Mod.</th>
<th># Ex.</th>
<th>C(0)+VC</th>
<th>C(1)+VC</th>
<th>Rem+C(0)+VC</th>
<th>VC</th>
<th>Rem+VC</th>
<th>Rem+SpC</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>30</td>
<td>0,0171</td>
<td>0,0249</td>
<td>0,0211</td>
<td>0,0856</td>
<td>0,0355</td>
<td>0,0553</td>
</tr>
<tr>
<td>R02</td>
<td>30</td>
<td>0,0168</td>
<td>0,0266</td>
<td>0,0205</td>
<td>0,0478</td>
<td>0,0283</td>
<td>0,0410</td>
</tr>
<tr>
<td>R03</td>
<td>29</td>
<td>0,0178</td>
<td>0,0274</td>
<td>0,0203</td>
<td>0,0156</td>
<td>0,0224</td>
<td>0,0261</td>
</tr>
<tr>
<td>R04</td>
<td>27</td>
<td>0,0168</td>
<td>0,0286</td>
<td>0,0198</td>
<td>0,0097</td>
<td>0,0203</td>
<td>0,0204</td>
</tr>
<tr>
<td>R05</td>
<td>8</td>
<td>0,0171</td>
<td>0,0270</td>
<td>0,0216</td>
<td>0,0162</td>
<td>0,0253</td>
<td>0,0354</td>
</tr>
<tr>
<td>RC</td>
<td>30</td>
<td>0,0178</td>
<td>0,0242</td>
<td>0,0232</td>
<td>0,1397</td>
<td>0,0516</td>
<td>0,0666</td>
</tr>
<tr>
<td>RM1</td>
<td>30</td>
<td>0,0185</td>
<td>0,0241</td>
<td>0,0219</td>
<td>0,1159</td>
<td>0,0483</td>
<td>0,0606</td>
</tr>
<tr>
<td>RM5</td>
<td>30</td>
<td>0,0168</td>
<td>0,0240</td>
<td>0,0220</td>
<td>0,0166</td>
<td>0,0291</td>
<td>0,0284</td>
</tr>
<tr>
<td>RND</td>
<td>30</td>
<td>0,0169</td>
<td>0,0258</td>
<td>0,0227</td>
<td>0,0161</td>
<td>0,0250</td>
<td>0,0219</td>
</tr>
<tr>
<td>TR</td>
<td>29</td>
<td>0,0166</td>
<td>0,0243</td>
<td>0,0227</td>
<td>0,0858</td>
<td>0,0397</td>
<td>0,0467</td>
</tr>
<tr>
<td>All</td>
<td>273</td>
<td>0,0172</td>
<td>0,0256</td>
<td>0,0216</td>
<td>0,0586</td>
<td>0,0333</td>
<td>0,0409</td>
</tr>
</tbody>
</table>

France, a network with 25 nodes:

<table>
<thead>
<tr>
<th>Mod.</th>
<th># Ex.</th>
<th>C(0)+VC</th>
<th>C(1)+VC</th>
<th>Rem+C(0)+VC</th>
<th>VC</th>
<th>Rem+VC</th>
<th>Rem+SpC</th>
</tr>
</thead>
<tbody>
<tr>
<td>R01</td>
<td>21</td>
<td>0,0099</td>
<td>0,0117</td>
<td>0,0112</td>
<td>0,0288</td>
<td>0,0176</td>
<td>0,0201</td>
</tr>
<tr>
<td>R02</td>
<td>18</td>
<td>0,0102</td>
<td>0,0132</td>
<td>0,0119</td>
<td>0,0214</td>
<td>0,0159</td>
<td>0,0178</td>
</tr>
<tr>
<td>R03</td>
<td>16</td>
<td>0,0096</td>
<td>0,0147</td>
<td>0,0117</td>
<td>0,0103</td>
<td>0,0132</td>
<td>0,0120</td>
</tr>
<tr>
<td>R04</td>
<td>11</td>
<td>0,0092</td>
<td>0,0140</td>
<td>0,0113</td>
<td>0,0050</td>
<td>0,0119</td>
<td>0,0093</td>
</tr>
<tr>
<td>R05</td>
<td>10</td>
<td>0,0104</td>
<td>0,0145</td>
<td>0,0115</td>
<td>0,0046</td>
<td>0,0129</td>
<td>0,0095</td>
</tr>
<tr>
<td>RC</td>
<td>30</td>
<td>0,0097</td>
<td>0,0114</td>
<td>0,0109</td>
<td>0,0275</td>
<td>0,0186</td>
<td>0,0228</td>
</tr>
<tr>
<td>RM1</td>
<td>30</td>
<td>0,0098</td>
<td>0,0116</td>
<td>0,0111</td>
<td>0,0371</td>
<td>0,0216</td>
<td>0,0259</td>
</tr>
<tr>
<td>RM5</td>
<td>30</td>
<td>0,0100</td>
<td>0,0117</td>
<td>0,0114</td>
<td>0,0159</td>
<td>0,0152</td>
<td>0,0134</td>
</tr>
<tr>
<td>RND</td>
<td>29</td>
<td>0,0095</td>
<td>0,0126</td>
<td>0,0119</td>
<td>0,0095</td>
<td>0,0135</td>
<td>0,0122</td>
</tr>
<tr>
<td>TR</td>
<td>30</td>
<td>0,0098</td>
<td>0,0112</td>
<td>0,0114</td>
<td>0,0251</td>
<td>0,0189</td>
<td>0,0201</td>
</tr>
<tr>
<td>All</td>
<td>225</td>
<td>0,0098</td>
<td>0,0123</td>
<td>0,0114</td>
<td>0,0209</td>
<td>0,0167</td>
<td>0,0176</td>
</tr>
</tbody>
</table>

Germany50, a network with 50 nodes:

Erik Haraldsson, 2008

5.4 Results from running the program
Finally Pioro40, with 40 nodes:

We can see that first removing extra SP-graphs and then checking for a valid cycle (Rem+VC) is faster than removing redundant SP-graphs and then look for subpath consistency (Rem+SpC) for Cost266, France and Germany50 and slower for Di-Yuan and Pioro40. One interesting result is that the average time stays about the same for all modifications when we combine SP-graphs, and that it fluctuates the most when we only use VC. One reason for this is that some modification methods create more valid cycles than other [5], which in turn affects how many SP-graph-pairs that the VC-algorithm has to look through. There are also some fluctuations in Rem+VC and Rem+SpC, and the ups and downs for both of them roughly follows that of VC, but less extreme, which would indicate that Rem also dampens the fluctuations, but much less so than VC. This makes sense since Rem removes some SP-graphs, but less than C(0)/C(1).

5.5 About Valid Cycles

I used the default flag -s 2, when I generated the subpath consistent examples that has a valid cycles when combined, in order to get as many examples as possible as fast as possible. I have also found that Di-Yuan has three examples that are subpath consistent and also has a valid cycle, and that Pioro40 has one example (see Appendix C.1 for more information). This table will compare how many of these valid cycles are found by some different methods and settings:
There are several conclusions that can be drawn from this table. First there are no SP-graphs that are subpath consistent and still has a 1-valid cycle before being combined in the examples, but figure 4 shows that it is possible. The second conclusion is that using the flag -s 0 will catch about 75% of all the valid cycles. By instead using the first two steps (-s 1), then all the valid cycles will be detected in the examples above. This will make sense if you just think about it. First, if two SP-graphs can't be combined, then they must be subpath inconsistent (according to Theorem 2). Second, we will in step two try and add the part of the paths that ends in a destination. If we try to add a part of a path in step three, then it must also be a shorter part of one path that we couldn't add in step two (because otherwise it would have been added in step two). This means that step three doesn't help anything when the problem is subpath consistent, and the theory matches up with the result from the examples.

5.6 Conclusions

The main conclusions of this work is that combining SP-graphs before checking for a valid cycle will greatly improve the total performance as seen in the tables above, and that it is possible to combine two SP-graphs in accordance with condition A if and only if they are subpath consistent as proved by Theorem 2 and that combining SP-graphs won't affect subpath consistency as shown in Theorem 3.

Here are some general conclusions about the methods used in the computational test:

C(0)+VC: This is the fastest method to test if a problem has a valid cycle. Useful if a short solution time is really important or it is likely that most examples are subpath inconsistent (and then only use C(1)+VC for the rest).

C(1)+VC: This method needs about twice the time of C(0)+VC (and much faster than any other method for larger |N|), but using a second step when combining SP-graphs allows the VC algorithms to find more valid cycles, which makes this the method that is generally best to use.

Rem+C(0)+VC: This method is slower than the ones above. However, first removing all redundant might make m(comb) even smaller (if all divided SP-graphs with a specific origin are redundant).

VC: This is the original method, and is still fastest for |N|=15 or less, but is slowest for all the other examples. It will also find fewer valid cycles than the methods that combines SP-graphs, so there isn't really any reason to just use this method any more.

Rem+VC: Will find the same number of valid cycles as the VC method, but is 2-4 times faster for larger |N|. Probably only useful for testing purposes.
Rem+SpC: This is the method to use if you specifically want to know whether an example is subpath consistent or not (see Appendix B for some results for just SpC). Probably only for testing.

So in conclusion C(1)+VC, that is first combine the SP-graphs with the first two steps, and then check for a valid cycle, is the method that will find most valid cycles and is the second fastest for larger |N| and is thus the preferred method. The time difference for smaller |N| is probably not even noticeable between the different methods, and the total time is quite small even for larger |N|, especially considering that setting weights is something that is used when setting up a network, or when adding more routes or similar.
Sources


Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years 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/

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år 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/

© 2008, Erik Haraldsson
Appendix A: The Source Code

A.1 graftest.h

//Headerfil

#include <iostream> //To use cin/cout
#include <fstream> //To read to/from files
#include <vector> //STL(Standard Template Library)
#include <set> //STL
#include <queue> //STL
#include <string>
#include <cstring>
#include <algorithm>
#include <cstdlib>
#include <numeric>

using namespace std;

#ifndef GRAFTEST_H_
#define GRAFTEST_H_
//Only read the definitions once
#define GRAFTEST_H_

//Structure used to represent an arc, by saving the startnode and endnode
struct arc
{
    unsigned int arcStartNode;
    unsigned int arcEndNode;
};

//Structure that is used to represent a SP-graph
struct SPGraph
{
    vector<arc> arcList; //Saves all the arcs in a SP-graph
    vector<unsigned int> inDegree; //Number of incoming arcs for each node
    vector<unsigned int> outDegree; //Number of outgoing arcs for each node
    vector<unsigned int> startNodes; //The startnodes for a SP-graph (with only outgoing arcs)
    vector<unsigned int> endNodes; //Nodes with only incoming arcs = endnodes
    set<unsigned int> nodes; //Saves all nodes that are a part of the SP-graph
};

//Just a couple of variables that deals with settings that are possible to set by the help of commands
//Coupled into a structure for easier use.
struct arguments
{
    unsigned int steps; //if the program should use extra steps to combine SP-graphs
    bool showMenu;
    string dataFile;
    string newFile;
    queue<unsigned int> toDoCommands;
};

//control.cpp, main() is here and a few other functions that lets the user to control what the program //should do

//text.cpp, only things to print on screens, like menus
void showHelp();
void showMainMenu(bool showMenu = true);
void showChangeGraphMenu(bool showMenu = true);
void showDataMenu(bool showMenu = true);

//startup.cpp, functions to run to start up the program, like read settings and the data you want to use
bool readInArgument(unsigned int argc, char* argv[], arguments & instructions);
void readInData(vector<SPGraph> & outSP, ifstream & data);
//addOrSplit.cpp, overview of the steps in order to combine or divide paths
void combineMain(const vector<SPGraph> & inSP, vector<SPGraph> & outSP, const unsigned int addPartialPaths, const bool createOutGraph);
void operator+=(vector<unsigned int> & outVector, const vector<unsigned int> & inVector);
void divideMain(const vector<SPGraph> & inSP, vector<SPGraph> & outSP, bool edit);

//DFS.cpp, algorithms that use DFS, in order to "walk" in the graph for some reason
void divideSPGraph(vector<SPGraph> & outSP, const SPGraph & inSP, const unsigned int startNode, const bool isOutGraph);
void getPartOfSPGraph(const SPGraph & inSP, SPGraph & outSP, const bool createOutGraph, const unsigned int startNode, unsigned int nodeToFind);
void getPartOfSPGraphIndex(const SPGraph & inSP, vector<unsigned int> & arcIndex, set<unsigned int> & connectedNodes, const bool createOutGraph, const unsigned int startNode, unsigned int nodeToFind);
unsigned int getArc(const SPGraph & inSP, const unsigned int inNode, const unsigned int arcNumber, const bool forward, const bool warn = true);

//algorithms.cpp, other algorithms for adding paths etc
bool checkIfCombinable(const SPGraph & path1ToCompare, const SPGraph & path2ToCompare, bool createOutGraph);
unsigned int testOrEditExtraSPGraphs(vector<SPGraph> & outSP, bool edit);
void testSubpathConsistency(const vector<SPGraph> & inSP);
void changePaths(vector<SPGraph> & outSP);

//arcFunc.cpp, smaller functions that helps using arcs
bool arcOperator<inNode>(const arc & & arc1, const arc & & arc2);
bool arcOperator==(const arc & & arc1, const arc & & arc2);
arc setArc(arc & & arc, arc & & original);
arc setArc(const unsigned int start, const unsigned int end);
arc setArcB(const unsigned int start, const unsigned int end, const bool forward);
unsigned int returnArcStartB(const arc & arc, const bool forward = true);
unsigned int returnArcEndB(const arc & arc, const bool forward = true);

//pathFunc.cpp, common functions that gives more usability to paths
unsigned int endNodeSizeB(const SPGraph & inSP, bool forward);
unsigned int startNodeSizeB(const SPGraph & inSP, bool forward);
unsigned int getStartNodeB(const SPGraph & inSP, const bool forward, const unsigned int index);
unsigned int getEndNodeB(const SPGraph & inSP, const bool forward, const unsigned int index);
unsigned int nodeOutDegreeB(const SPGraph & inSP, const bool forward, const unsigned int index);
unsigned int nodeInDegreeB(const SPGraph & inSP, const unsigned int node, const bool forward);
void combinePaths(SPGraph & outSP, const vector<arc> & inArcList);
void combinePaths(SPGraph & outSP, const SPGraph & inSP);
void calculatePathData(SPGraph & outSP);
void calculatePathData(vector<SPGraph> & outSP);

//printorend.cpp, functions to print paths or use when the program ends
void printToFile(const vector<SPGraph> & inSP, string & fileToWriteTo);
void showMeta (const vector<SPGraph> & inSP);
void showPath (const SPGraph & inSP);
void showData (const SPGraph & inSP);
void showNodes (const SPGraph & inSP);

#endif

A.2 control.cpp

//Main file, about controlling what the program should do next (basically the interface)
//About counting the running time/operations calculation:
// |N| is the number of nodes in a tree
// |A| is the number of arcs in a tree
// m is the number of SP-graphs stored in vector<SPGraph>
// O(|A|) means linear to the number of arcs

#include "graftest.h"
#include <ctime>

void startLog(const vector<SPGraph> & inSP, vector<SPGraph> & outSP, const bool createOutGraph);
void useData(vector<SPGraph> & currentGraph);
bool changeCurrentGraph();
unsigned int returnChoice(unsigned int menuToShow);

unsigned int currentGraphValue = 1; //Always refers to the same ex
arguments instructions = {2, true}; //Only one ex, tailored for this program, sets start-up values
extern unsigned int numberOfNodes;

int main(unsigned int argc, char * argv[]) {
    clock_t start = clock();

    vector<SPGraph> startGraph, dividedGraph, inGraph, outGraph;
    vector<SPGraph>* currentGraph = &startGraph; //Defined as a pointer since references
           //can't be changed

    if ( readInArgument(argc, argv, instructions) ) //True if showing help-text, should
        return 0; // break program afterwards

    if ( instructions.dataFile.empty() ) //If no file is specified with the -f command, then it
        //can be chosen now
    {
        cout << "Name of file to read from:";
        getline(cin, instructions.dataFile);
    }

    ifstream indataFil;
    indataFil.open(instructions.dataFile.c_str());
    if(indataFil.is_open())
        readInData(startGraph, indataFil);
    else
    {
        cout << "Error open indata-file" << endl;
        return 1;
    }
    cout << "Current File: " << instructions.dataFile << endl;
    indataFil.close();

    cin.clear();

    while (true)
        switch (returnChoice(0)) {
        case 1 : useData(*currentGraph); //Submenu for current SP-graphs
            break;
        case 2:
            if ( currentGraphValue != 2 )
                divideMain(*currentGraph, dividedGraph, true);
            break;
        case 3:
            if ( currentGraphValue != 3 ) //Create in-graphs
                combineMain(*currentGraph, inGraph, instructions.steps, false);
            break;
        case 4:
            if ( currentGraphValue != 4 ) //Create out-graphs
                combineMain(*currentGraph, outGraph, instructions.steps, true);
            break;
        }
}

Erik Haraldsson, 2008
break;
case 5:
    if (changeCurrentGraph())
    {
        if (currentGraphValue==1)
            currentGraph = &startGraph;
        else if (currentGraphValue==2)
            currentGraph = &dividedGraph;
        else if (currentGraphValue==3)
            currentGraph = &inGraph;
        else if (currentGraphValue==4)
            currentGraph = &outGraph;
    }
    break;
case 6:
    cout << "Number of original SP-graphs: " << currentGraph->size() << endl;
    cout << "Number of extra SP-graphs: " << testOrEditExtraSPGraphs(*currentGraph, true) << endl;
    cout << "Number of SP-graphs left: " << currentGraph->size() << endl;
    break;
case 7:
    changePaths(*currentGraph);
    break;
case 8:
    testSubpathConsistency(*currentGraph);
    break;
case 9:
    showMainMenu();
    break;
case 0: //Commands to run when exiting the program
    clock_t stop = clock();
    cout << "Total time: " << (stop-start)/(float)CLOCKS_PER_SEC << endl;
    return 0;
}

void useData(vector<SPGraph> & currentGraph)
{
    string tempFile;
    while (true)
        switch (returnChoice(1))
        {
            case 1: showMeta(currentGraph);
                break;
            case 2:
                unsigned int index;
                cout << "What index to show (OBS! 0 for first path, 1 for second, ...)?";
                cin >> index;
                if (index<currentGraph.size())
                    showData(currentGraph[index]); //Calls print-function
                    //for the selected SP-graph
                break;
            case 3: //Calls print-function once for each SP-graph
                for (unsigned int i=0; i < currentGraph.size(); i++)
                    showPath(currentGraph[i]);
                break;
            case 4: //Calls print-function once for each SP-graph
                for (unsigned int i=0; i < currentGraph.size(); i++)
                    showData(currentGraph[i]);
                break;
            case 5:
                divideMain(currentGraph, currentGraph, false);
                break;
            case 6:
                printToFile(currentGraph, instructions.newFile);
                break;
        }
case 7:
    cout << "Number of extra SP-graphs: " << testOrEditExtraSPGraphs(currentGraph, false) << endl;
    break;

case 8:
    tempFile = "temp.spg";
    printToFile(currentGraph, tempFile);
    char nowt[50];
    sprintf(nowt, "./spgb %u temp.spg", numberOfNodes); // OBS!! This command
    // is sensitive for which OS is running. OBS!!
    system(nowt);
    break;

case 9:
    showDataMenu();
    break;

case 0:
    return;
}

bool changeCurrentGraph()
{
    unsigned int val;
    while (true)
    {
        switch (val = returnChoice(5))
        {
        case 1:
        case 2:
        case 3:
        case 4:
            currentGraphValue = val;
            return true;
        case 9:
            showChangeGraphMenu();
            break;
        case 0:
            return false;
        }
    }

    unsigned int returnChoice(unsigned int menuToShow)
    {
        unsigned int val;
        if (!instructions.toDoCommands.empty())
        {
            switch (menuToShow)
            {
            case 0:
                showMainMenu(instructions.showMenu);
                break;
            case 1:
                showDataMenu(instructions.showMenu);
                break;
            case 5:
                showChangeGraphMenu(instructions.showMenu);
                break;
            }
        }
        if (!cin >> val)
            return 0;
        }
    else
    {
        val = instructions.toDoCommands.front();
        instructions.toDoCommands.pop();
    }
    return val;
}
//Functions to display the text in the menus for guidance

#include "grafittest.h"

void showCurrentData(unsigned int number);

extern unsigned int currentGraphValue;

void showHelp()
{
    cout << "-f <file> \tuses <file> as the indata for the graph.\n";
    cout << "-h \tThis text.\n";
    cout << "-m \tHide menu.\n";
    cout << "-p <numbers> \tcontrols how to navigate through the options\n";
    cout << "-s <0/1/2> \tcontrols how many extra steps to take to find complete
\tout-graphs (in-graphs), default=2\n";
    cout << "-w <file> \tuses <file> as the filename that you want to write some
\tdata to.\n";
}

void showMainMenu(bool showMenu)
{
    if (showMenu)
    {
        cout << "Current data is: \n";
        showCurrentData(currentGraphValue);
        cout << "\nChoose an alternative!\n";
        cout << "1. Submenu for current SP-graphs\n";
        cout << "2. Divide SP-graphs\n";
        cout << "3. Combine SP-graphs to in-graph\n";
        cout << "4. Combine SP-graphs to out-graph\n";
        cout << "5. Change current data\n";
        cout << "6. Remove redundant paths\n";
        cout << "7. Change a path\n";
        cout << "8. Test subpath consistency\n";
        cout << "0. Exit\n";
    }
}

void showChangeGraphMenu(bool showMenu)
{
    if (showMenu)
    {
        cout << "Set current data to:\n";
        cout << "1. Original SP-graph\n";
        cout << "2. Divided SP-graph\n";
        cout << "3. In-Graph\n";
        cout << "4. Out-Graph\n";
        cout << "0. Go to main menu without changing current data\n";
    }
}

void showDataMenu(bool showMenu)
{
    if (showMenu)
    {
        cout << "Choose an alternative!\n";
        cout << "1. Print startnodes/endnodes for all SP-graphs on screen\n";
        cout << "2. Print a specific SP-graph to screen\n";
        cout << "3. Print all SP-graphs to screen\n";
        cout << "4. Print all SP-graphs and their in-/out-degree on screen\n";
    }
}
cout << "5. Check if the SP-graphs can be divided\n";
cout << "6. Save all SP-graphs to a file\n";
cout << "7. Test current data for redundant SP-graphs\n";
cout << "8. Check for valid cycles (External Program)\n";
cout << "0. Main menu\n";
}
}

void showCurrentData(unsigned int number)
{
    switch (number)
    {
    case 1: cout << "Original Graph";
            break;
    case 2: cout << "Divided Graph";
            break;
    case 3: cout << "In-Graph";
            break;
    case 4: cout << "Out-Graph";
            break;
    }
}

#include "graftest.h"

void autoCreateStart(arguments & instructions);

unsigned int numberOfNodes=0;

A.4 startup.cpp

//Functions basically used when starting the program

void autoCreateStart(arguments & instructions);

unsigned int numberOfNodes=0;

// Execution time depends on the number of arguments, but it could probably be neglected.
// This function processes the arguments given in the command-line

bool readInArgument(unsigned int argc, char *argv[], arguments & instructions)
{
    for (unsigned int numberOfArgumentsProcessed = 1; numberOfArgumentsProcessed < argc;
         numberOfArgumentsProcessed++)
    {
        if (argv[numberOfArgumentsProcessed][0] == '-' and
            argv[numberOfArgumentsProcessed][2] == '\0')
            switch (argv[numberOfArgumentsProcessed][1])
            {
            case 'f': //f flag specifies that the next argument of argv is the
                      //name of the datafile
                instructions.dataFile = argv[++numberOfArgumentsProcessed];
                break;
            case 'h': //Shows helptext for the commands
                showHelp();
                return true;
                break;
            case 'm': //Hides menu
                instructions.showMenu = false;
                cout << "Type '9' at any time to get a menu.\n";
                break;
            case 'p': //Lets the user specify which options to use, without having to do so
                      //during the run and also hides the menu display during that time.
                numberOfArgumentsProcessed++;
                char tempChar[2];
                tempChar[1] = '\0';
                for (unsigned int i=0; argv[numberOfArgumentsProcessed][i] != '\0';
                     i++)
                    {
                        tempChar[0] = argv[numberOfArgumentsProcessed][i];
                    }
            }
instructions.toDoCommands.push(atoi(tempChar));
}

instructions.steps = atoi(argv[++numberOfArgumentsProcessed]);

break;

instructions.newFile = argv[++numberOfArgumentsProcessed];
break;
}

return false;
}

// O( m*|A_L|*log(|A_L|) ) 
void readInData (vector<SPGraph>& outSP, ifstream & data)
{
    unsigned int numberOfSPGraphs, numberOfArcsInGraph, startNode, endNode, nNodes = 0;

data >> numberOfSPGraphs; //Reads the number of SP-graphs in a file
outSP.resize(numberOfSPGraphs); //Resize the vector to save 'numberOfSPGraphs' SP-graphs

for (unsigned int i=0; i<numberOfSPGraphs; i++) // m times
{
    data >> numberOfArcsInGraph; //the number of arcs in the current SP-graph
outSP[i].arcList.resize(numberOfArcsInGraph); //Makes enough place in arcList to be able to save all arcs

    for (unsigned int j=0; j<numberOfArcsInGraph; j++) // |A_L| times
    {
        data >> startNode >> endNode; //Reads the startnode and endnode for each arc
outSP[i].arcList[j] = setArc(startNode, endNode); //Saves the current arc

        if (endNode > nNodes) //Saves the highest nodenumber found
nNodes = endNode;
        if (startNode > nNodes)
nNodes = startNode;
    }
}

numberOfNodes = nNodes; //Saves the total number of nodes in the data to a global variable

calculatePathData(outSP); // O(m * |A_L|*log(|A_L|) )
}

A.5 addOrSplit.cpp

#include "graftest.h"

void findMultiPaths(const vector<SPGraph>& inSP, vector<SPGraph>& outSP, const bool createOutGraph); 
vector<unsigned int> addPathsPart1(const vector<SPGraph>& inSP, vector<SPGraph>& outSP, 
const bool createOutGraph); 
vector<unsigned int> addPathsPart2or3(const vector<SPGraph>& inSP, vector<SPGraph>& outSP, 
const bool createOutGraph, const bool isPart2); 
unsigned int addIfPossible(const SPGraph & inSP, SPGraph & outSP, const unsigned int endPoint, const bool createOutGraph);

extern unsigned int numberOfNodes;

void combineMain(const vector<SPGraph>& inSP, vector<SPGraph>& outSP, 
const unsigned int addPartialPaths, const bool createOutGraph)
vector<unsigned int> part1Result(3, 0), part2Result(3, 0), part3Result(3, 0);
vector<SPGraph> inSP_Modified;

if (!outSP.empty())
  outSP.clear();

findMultiPaths(inSP, inSP_Modified, createOutGraph); //O(m(in)\#start|A_i|^2log(|A_i|))

//Step 0:
part1Result += addPathsPart1(inSP, outSP, createOutGraph); //O(m\#out|A_i|^log(|A_i|))
if (!inSP_Modified.empty())
  part1Result += addPathsPart1(inSP_Modified, outSP, createOutGraph);

//Step 1: Check for SP-graphs to put together (from endnodes)
if (addPartialPaths >= 1)
{
  // O(m\#out)\#start|A_i|^2log(|A_i|)
  part2Result += addPathsPart2or3(inSP, outSP, createOutGraph, true); //true=part 2
  if (!inSP_Modified.empty())
    part2Result += addPathsPart2or3(inSP_Modified, outSP, createOutGraph, true);
}

//Step 2: Check for SP-graphs to put together (from all nodes)
if (addPartialPaths >= 2)
{
  // O(m\#out)\#unconnected nodes|A_i|^2log(|A_i|)
  part3Result += addPathsPart2or3(inSP, outSP, createOutGraph, false); //false=part 3
  if (!inSP_Modified.empty())
    part3Result += addPathsPart2or3(inSP_Modified, outSP, createOutGraph, false);
}

vector<unsigned int> nodesPerGraph(outSP.size(), 0);
//Uses node structure so show more interesting results
for (unsigned int i = 0; i < outSP.size(); i++)
  nodesPerGraph[i] = outSP[i].nodes.size();

//Print interesting results
cout << "We started with " << inSP.size() << " SP-graphs\n";
cout << " and ended up with " << outSP.size() << " SP-graphs\n";
cout << "Number of SP-graphs that span\'s all nodes: ";
cout << count(nodesPerGraph.begin(), nodesPerGraph.end(), numberOfNodes) << endl;
cout << "Number of nodes: " << numberOfNodes << endl;
if (addPartialPaths == 0)
{
  cout << "Average number of nodes/SP-graph: ";
cout << accumulate(nodesPerGraph.begin(), nodesPerGraph.end(), 0) /
    nodesPerGraph.size();
  cout << "\nNumber of SP-graphs which is only connected to half or less of the nodes: ";
cout << count_if(nodesPerGraph.begin(), nodesPerGraph.end(),
             bind2nd(less_equal<
               unsigned int>(), numberOfNodes/2)));
}
else
{
  cout << "Number of nodes/SP-graph (if less than max): ";
  for (unsigned int i = 0; i < outSP.size(); i++)
    if (nodesPerGraph[i] < numberOfNodes)
      cout << nodesPerGraph[i] << " ";
  cout << endl << endl;
}

cout << "Part 1:\n";
cout << part1Result[0] << " checks\n";
cout << part1Result[1] << " successful compares\n";
cout << part1Result[2] << " unsuccessful compares\n";
if (addPartialPaths == 1)
cout << "Part 2:\n";
cout << part2Result[0] << " checks:\n";
cout << part2Result[1] << " successful compares:\n";
cout << part2Result[2] << " unsuccessful compares:\n"
};

if (addPartialPaths > 1)
{
cout << "Part 3:\n";
cout << part3Result[0] << " checks:\n";
cout << part3Result[1] << " successful compares:\n";
cout << part3Result[2] << " unsuccessful compares:\n"
}

//Splits up SP-graphs with multiple startnodes (endnodes)
// At most O(m*n)\*log(|A_i|) times
void findMultiPaths(const vector<SPGraph> & inSP, vector<SPGraph> & outSP, const bool createOutGraph)
{
unsigned int oldSize = 0, nStartNodes;
for (unsigned int i = 0; i < inSP.size(); i++) // m times
if (startNodeSizeB(inSP[i], createOutGraph) > 1) // Several startnodes (endnodes)
{
  nStartNodes = startNodeSizeB(inSP[i], createOutGraph);
  outSP.resize(outSP.size() + nStartNodes);
  for (unsigned int j = 0; j < nStartNodes; j++) // #start times
    getPartOfSPGraph(inSP[i], outSP[oldSize + j], createOutGraph, getStartNodeB(inSP[i], createOutGraph[j]), 0); // O(|A_i|\*log(|A_i|))
  oldSize += nStartNodes;
}

}

void operator+=(vector<unsigned int> & outVector, const vector<unsigned int> & inVector)
{
unsigned int minSize = outVector.size() < inVector.size() ? outVector.size() : inVector.size();

for (unsigned int i = 0; i < inVector.size(); i++)
  outVector[i] += inVector[i];

// Only checks SP-graphs with one endnode (startnode)
// O(m\*m(out))\*log(|A_i|), but since the checkIfCombinable will only be called about 1-3 times for
// each added SP-graph the real time is most likely close to O(m^2\*log(|A_i|))
vector<unsigned int> addPathsPart1(const vector<SPGraph> & inSP, vector<SPGraph> & outSP, const bool createOutGraph)
{
  vector<unsigned int> sucessRate(3, 0);
  bool added;

  for (unsigned int i = 0; i < inSP.size(); i++) // m times
    if (startNodeSizeB(inSP[i], createOutGraph) == 1)
    {
      added = false;
      sucessRate[0] += outSP.size();
      for (unsigned int j = 0; j < outSP.size(); j++) // (m(out)) times
        if (getStartNodeB(outSP[j], createOutGraph, 0) ==
            getStartNodeB(outSP[j], createOutGraph, 0))
          if (checkIfCombinable(inSP[i], outSP[j], createOutGraph)) // O(|A_i|)
            { added = true;
              sucessRate[1] +=;
              combinePaths(outSP[j], inSP[i]); // O(|A_i|\*log(|A_i|))
              break; //Will only break the inner for-loop
            }
    }
}
Erik Haraldsson, 2008 - 63 -
void divideMain(const vector<SPGraph> &inSP, vector<SPGraph> &outSP, bool edit)
{
    unsigned int nSimplePaths=0, nInGraphs=0, nOutGraphs=0, nRest=0;

    if (edit)
    {
        if (!inSP.empty())
            outSP.clear();
        unsigned int numberOut=0;
        for (unsigned int i=0; i<inSP.size(); i++)
        {
            numberOut += inSP[i].startNodes.size()*inSP[i].endNodes.size();
            outSP.reserve(numberOut); //Reserve the resulting number of divided SP-graphs
        }

    } //loop that deals with one tree at a time, m times
    for (unsigned int i=0; i<inSP.size(); i++)
    { //The SP-graph was already a path
        if ( (inSP[i].startNodes.size() == 1) && (inSP[i].endNodes.size() == 1) )
        {
            nSimplePaths++;
            if (edit)
                outSP.push_back(inSP[i]); //Just add the SP-graph
        }

    } //One endnode and several startnodes => in-graph
    else if ( (inSP[i].startNodes.size() > 1) && (inSP[i].endNodes.size() == 1) )
    {
        nInGraphs++;
        if (edit)
            divideSPGraph(outSP, inSP[i], inSP[i].endNodes[0], false); // O( #start*|A_i|^2*log(|A_i| )
    }

    } //One startnode and several endnodes => out-graph
    else if ( (inSP[i].startNodes.size() == 1) && (inSP[i].endNodes.size() > 1) )
    {
        nOutGraphs++;
        if (edit)
            divideSPGraph(outSP, inSP[i], inSP[i].startNodes[0], true); // O( #end*|A_i|^2*log(|A_i| )
    }

    else if ( (inSP[i].startNodes.size() > 1) && (inSP[i].endNodes.size() > 1) )
    {
        nRest++;
        if (edit)
            //Divides the SP-graph once for every startnode
            for (unsigned int j=0; j<inSP[i].startNodes.size(); j++) //#start times
                divideSPGraph(outSP, inSP[i], inSP[i].startNodes[j], true); // O( #end*|A_i|^2*log(|A_i| )
    }

    if (edit)
    {
        cout << "Number of SP-graphs before dividing: " << inSP.size() << endl;
        cout << "Number of SP-graphs after dividing: " << outSP.size() << endl;
    }
    else
    {
        cout << "Total number of SP-graphs: " << inSP.size() << endl;
        cout << "Number of already divided: " << nSimplePaths << endl;
        cout << "Number of in-graphs: " << nInGraphs << endl;
        cout << "Number of out-graphs: " << nOutGraphs << endl;
        cout << "Number of more complex SP-graphs: " << nRest << endl;
    }
}
A.6 DFS.cpp

//Functions that relates to DFS
#include "graftest.h"

unsigned int backtrackPath(const vector<unsigned int> & useNode, const bool outGraph, vector<arc> & arcStack);

extern unsigned int numberOfNodes;

// Function that divides the SP-graph inSP into several SP-graphs with one set of paths each, and place
// the new SP-graphs at the end of outSP.
// Function that relates to DFS

unsigned int nArcsToCheck (numberOfNodes); //Vector that remembers how many arcs 
from a node that are left to check

if (isOutGraph)
  copy(inSP,outDegree.begin(), inSP.outDegree.end(), nArcsToCheck.begin());
else
  copy(inSP.inDegree.begin(), inSP.inDegree.end(), nArcsToCheck.begin());

bool newPath;
vector<arc> arcStack; //To save the current path

unsigned int minSteps, tempNode, nextNode = startNode, startingIndex=outSP.size();

// #end = The total number of endpoints in the currentSP-graph, inSP.endNodes.size()
// (startnodes if going backwards)
// #paths = total number of paths in a SP-graph
// getArc is called |A_l| times (since each arc is added to the stack once)
// calculatePathData is called #end times
// combinePaths is at most called (#paths-*end)*#end times (it is tested for each newly found path,
// but combinePaths will only be called when the path ends in a node that has already been visited,
// and then it will only be called for the SP-graphs that are connected to that node, which is at most
// #end)
// backtrackPath is called #paths. times (once for each lap in the outer while-loop).
// However, sicne the getArc-function in the inner loop is at most called |A_l| times, then any step in
// the outer loop must also be called at most |A_l| times, and probably a lot less.
// This all gives:
// O (|A_l|^2 + #end*|A_l|^log(|A_l|) + (#paths-*end)*#end*|A_l|^log(|A_l|) + |A_l|^2 |N| ) =
// Part: 1 + 2B + 2A + 3
// = O (|A_l|^2|A_l|^2log(|A_l|)
while (nextNode != 0) //At most min(#paths,|A_l|) times
{
  // 1: Find new path
  while (nArcsToCheck[nextNode-1] != 0)
  {
    tempNode = nextNode; //To know the startnode of the latest arc
    // When nArcsToCheck[tempNode-1] is 0 the function returns the first new node found
    // when it is 1 it returns the second node found...
    nextNode = getArc(inSP, tempNode, nArcsToCheck[tempNode-1]-1, isOutGraph);
    // O(|A_l|), at most called |A_l| times
    nArcsToCheck[tempNode-1]--; //Marks that one more arc from/to tempNode is checked
    arcStack.push_back(setArcB(tempNode, nextNode, isOutGraph)); // adds the new arc
    // to arcStack
  }

  // 2A: Add the latest path to other paths
  newPath = true;
  for (unsigned int i=startingIndex; i<outSP.size(); i++) // at most #end times
    if (outSP[i].nodes.count(nextNode) > 0) //True at max (#paths-*end)*#end times

combinePaths(outSP[i], arcStack); //O(|A| * log(|A|))
newPath = false;
}
// 2B: Use the latest path as a base of a new SP-graph
if (newPath) // True times
{
    outSP.resize(outSP.size() + 1);
    outSP.back().arcList = arcStack;
    calculatePathData(outSP.back()); //O(|A| * log(|A|))
}

// 3: backtrackPath removes arcs from arcStack untill we find an arc we haven't been looking
// at and returns 0 if we have searched all arcs
nextNode = backtrackPath(nArcsToCheck, isOutGraph, arcStack); // O(|N|),
//called #paths times
}
return;
}

// If nodeToFind == 0, then the function adds all paths going out from (coming in to) startNode.
// O(|A|² * log(|A|)) As above, except only one outgoing SP-graph to add new paths to.
void getPartOfSPGraph(const SPGraph &inSP, SPGraph &outSP, const bool createOutGraph,
const unsigned int startNode, unsigned int nodeToFind)
{
    outSP.arcList.clear(); //Just to make sure that the function writes to an empty SP-graph
    outSP.nodes.clear();
    outSP.startNodes.clear();
    outSP.endNodes.clear();

    vector<arc> arcStack;
    vector<unsigned int> nArcsToCheck(numberOfNodes);
    if (createOutGraph)
        copy(inSP.outDegree.begin(), inSP.outDegree.end(), nArcsToCheck.begin());
    else
        copy(inSP.inDegree.begin(), inSP.inDegree.end(), nArcsToCheck.begin());

    unsigned int tempNode, nextNode = startNode;

    while (nextNode != 0) // at most min(#paths, |A|) times
    {
        if (nArcsToCheck[nextNode-1] != 0)
            do
            {
                tempNode = nextNode;
                nextNode = getArc(inSP, tempNode, nArcsToCheck[tempNode-1]-1,
                createOutGraph); //O(|A|), at most |A| times
                nArcsToCheck[tempNode-1]--;
                arcStack.push_back(setArcB(tempNode, nextNode, createOutGraph));
            }
        while ((nextNode != nodeToFind) and (nArcsToCheck[nextNode-1] != 0));

        if (outSP.nodes.count(nextNode) > 0 or nodeToFind > numberOfNodes)
            combinePaths(outSP, arcStack); //O(|A| * log(|A|))
        else if (nextNode == nodeToFind or nodeToFind == 0) // At most one time
            {  
                outSP.arcList = arcStack;
                calculatePathData(outSP); //O(|A| * log(|A|))
                if (nodeToFind == 0)
                    nodeToFind = numberOfNodes + 1; // Mark that the rest of the found paths should be
                    // combined with the first one
            }
}
Erik Haraldsson, 2008

nextNode = backtrackPath(nArcsToCheck, createOutGraph, arcStack);
//O(|N|) called at most #paths times
}

// This version returns a vector where it marks which arcs will be used in a path instead of an actual path //O(|N|*|A|). In this version, there is no method implemented for when nodeToFind is 0!!
void getPartOfSPGraphIndex(const SPGraph & inSP, vector<unsigned int> & arcIndex, set<unsigned int> & connectedNodes, const bool createOutGraph, const unsigned int startNode, unsigned int nodeToFind)
{
arcIndex.assign(inSP.arcList.size(), 0);
connectedNodes.clear();
vector<arc> arcStack;
vector<unsigned int> nArcsToCheck(numberOfNodes);
if (createOutGraph)
copy(inSP.outDegree.begin(), inSP.outDegree.end(), nArcsToCheck.begin());
else
copy(inSP.inDegree.begin(), inSP.inDegree.end(), nArcsToCheck.begin());

connectedNodes.insert(nodeToFind);
unsigned int tempNode, nextNode = startNode, stackIndex = 0;

while (nextNode != 0)
{
if (nArcsToCheck[nextNode-1] != 0)
do
{
tempNode = nextNode;
nextNode = getArc(inSP, tempNode, nArcsToCheck[nextNode-1]-1,
createOutGraph); //O(|A|), at most |A| times
nArcsToCheck[nextNode-1]--;
arcStack.push_back(setArcB(tempNode, nextNode, createOutGraph));
} while ( (nextNode != nodeToFind) and (nArcsToCheck[nextNode-1] != 0) );

if (connectedNodes.count(nextNode) > 0) //true at most #paths-*end times < |A|
{
for (unsigned int i=stackIndex; i < arcStack.size(); i++) //at most |N| times
{
arcIndex[find(inSP.arcList.begin(), inSP.arcList.end(), arcStack[i])-
inSP.arcList.begin()]=1; //O(|A|), called at most |N| times
connectedNodes.insert(returnArcStartB(arcStack[i], createOutGraph));
}
nextNode = backtrackPath(nArcsToCheck, createOutGraph, arcStack);
//O(|N|) called at most #paths times
stackIndex = arcStack.size();
}
else
{
nextNode = backtrackPath(nArcsToCheck, createOutGraph, arcStack);
//O(|N|) called at most #paths times
if ( stackIndex > arcStack.size() )
stackIndex = arcStack.size();
}
}

// Removes arcs from arcStack, one at a time until we either find that there is an arc we haven't looked
// through or the vector arcStack is empty.
// At max O(|A|) generally since the loop is executed at most |A| times, but max O(|N|) as this
// function is used in my program.
unsigned int backtrackPath(const vector<unsigned int> & nArcsLeft, const bool outGraph,
vector<arc> & arcStack)
{
    unsigned int nextNode;

    //when nArcsLeft is higher than 0, there are still arcs to be looked through and if it is 0 for all
    //nodes, including the startnode, then the graph has been completely searched through,
    //and the DFS should be aborted.
    do
    {
        if ( arcStack.empty() ) //True if all arcs have been searched
            return 0;
        nextNode = returnArcStartB(arcStack.back(), outGraph);
        arcStack.pop_back(); //Removes the last added arc.
    } while ( nArcsLeft[nextNode-1] == 0 );

    return nextNode; //Use DFS to search forward from nextNode
}

//Function to find an arc from inNode to another node
//  Takes at most O(|AL|) iterations
unsigned int getArc(const SPGraph & inSP, const unsigned int inNode, const unsigned int arcNumber, const bool forward, const bool warn)
{
    unsigned int counter = 0;

    //If arcNumber is 0, the function will return the first arc that starts (ends) in inNode,
    //if 1 the second arc...
    for (unsigned int i=0; i<inSP.arcList.size(); i++) // |A| times
        if ( returnArcStartB(inSP.arcList[i], forward) == inNode )
            if ( counter == arcNumber ) //Test if it is the number that we want
                return returnArcEndB(inSP.arcList[i], forward);
            counter++;
    if (warn)
        cout << "Warning!!! Function getArc returns 0. Something is probably wrong
               somewhere!\n";
    return 0; //If the function can't find the node requested
}

A.7 algorithms.cpp

#include "graftest.h"

bool getCommonArcs(const SPGraph & inSP, const unsigned int & inNode, const unsigned int & arcNumber, const bool forward, const bool warn);
bool checkForRedundantSPGraphs (const SPGraph & testingPath, const SPGraph & referencePath);
bool checkSubpathConsistency (const SPGraph & inSP1, const SPGraph & inSP2);

extern unsigned int numberOfNodes;

//O(|A|)
bool checkIfCombinable(const SPGraph & inSP1, const SPGraph & inSP2, bool createOutGraph)
{
    set<arc> common arcs SP1, common arcs SP2;
    set<unsigned int> commonNodes; //Nodes that both SP-graphs connect to
    set_intersection(inSP1.nodes.begin(), inSP1.nodes.end(), inSP2.nodes.begin(),
                        inSP2.nodes.end(), insert_iterator<set<unsigned int> >(commonNodes,
                        commonNodes.begin())); //min(O(|N|),O(|A|))

    if ( getCommonArcs(inSP1, commonNodes, commonArcsSP1, createOutGraph) //O(|A|) at most
         and getCommonArcs(inSP2, commonNodes, commonArcsSP2, createOutGraph) ) //O(|A|)
if (commonArcsSP1 == commonArcsSP2)  //linear, O(|A|)
    return true;

return false;

} //O(|A|) at most

bool getCommonArcs(const SPGraph & inSP, const unsigned int & commonNodes, set<arc> & commonArcs, bool forward)
{
    for (unsigned int i=0; i<inSP.arcList.size(); i++) //|A| times at most
        if (commonNodes.count(returnArcEndB(inSP.arcList[i], forward)) == 1)
            if (commonNodes.count(returnArcStartB(inSP.arcList[i], forward)) == 1)
                commonArcs.insert(inSP.arcList[i]);
        else
            return false;

    return true;

} //Total: O(m²*|A|²*log(|A|))

unsigned int testOrEditExtraSPGraphs (vector<SPGraph> & outSP, bool edit)
{
    vector<unsigned int> markedAsExtra(outSP.size(),0);

    unsigned int counter = 0;
    for (unsigned int i=0; i<outSP.size(); i++) //To keep i>=0, m times
        for (unsigned int j=0; j<outSP.size(); j++) //m times
            if (markedAsExtra[i]==0)
                if (checkForRedundantSPGraphs(outSP[i], outSP[j])) //O( |A|²*log(|A|) )
                    {markedAsExtra[i]++; counter++; break;}

    if (edit)
    {
        unsigned int to, from=0;
        while (markedAsExtra[from]==0) //No need to copy paths to themselves
            from++;
        for (to=from; from<outSP.size(); from++)
            if (markedAsExtra[from]==0) //Paths that should be saved
                outSP[to++]=outSP[from];
        outSP.erase(outSP.begin()+to, outSP.end());
    }

    return counter;

} //at most O( |A|²*log(|A|) )

bool checkForRedundantSPGraphs (const SPGraph & testingPath, const SPGraph & referencePath)
{
    if (testingPath.arcList.size() < referencePath.arcList.size())
    {
        if (includes(referencePath.arcList.begin(), referencePath.arcList.end()),
            testingPath.arcList.begin(), testingPath.arcList.end()))) //O(|A|)
        {
            //Needs to checks so that referencePath doesn't contain any alternative paths.
            //First check in- and out-degree
            bool flag = true;
            for (set<unsigned int>::iterator it = testingPath.nodes.begin();
                it != testingPath.nodes.end(); it++) //N times
                if ( (*it != testingPath.startNodes[0] and nodeInDegreeB(referencePath, *it, true))
                    != nodeInDegreeB(testingPath, *it, true) or
                    (*it != testingPath.endNodes[0] and nodeOutDegreeB(referencePath,
A.7 algorithms.cpp

```cpp
// O( |CN|² |N|² |A|² )
bool checkSubpathConsistency (const SPGraph & inSP1, const SPGraph & inSP2) {
    set<unsigned int> connectedNodes; //CN times, short for #Common Nodes
    vector<unsigned int> tempSP1, tempSP2;
    for (node1 = commonNodes.begin(); node1 != commonNodes.end(); node1++) //CN times, short for #Common Nodes
        getPartOfSPGraphIndex(inSP1, tempSP1, connectedNodes, direction, *node1,
                               *node2); //O(|N|² |A|²)
    if (accumulate(tempSP1.begin(), tempSP1.end(), 0) == 0) //O(|A|)
        continue;
```

Erik Haraldsson, 2008


Erik Haraldsson, 2008 - 71 - A.7 algorithms.cpp


```cpp
void changePaths(vector<SPGraph> & outSP)
{
  static unsigned int counterRand = 0;
  srand(time(0) + counterRand*3); //This way `rand()` will start at different spots if called again
  //before `time(0)` is updated
  counterRand++;

  unsigned int rindex, nnew, rold, start, end, nPaths=outSP.size();

  bool flag = true;
  for (unsigned int i=0; i<nPaths; i++) //m times
    if (outSP[i].arcList.size() >= 2 and outSP[i].nodes.size() < numberOfNodes) //O(|A|)
      { flag = false; break; }
  
  if (flag) { cout << "The algorithm is not adapted to make a random change to these
                   SP-graphs!\n"; return; //Not possible to find a allowable `rindex`, so can't generate new arcs! }

  //Look until if finds a changeable SP-graph
  do { rindex = rand() % nPaths; } while (outSP[rindex].arcList.size() < 2 or outSP[rindex].nodes.size() == numberOfNodes);

  //Keeps looking for an unused node in the current SP-graph
  do { nnew = rand() % numberOfNodes + 1; } while (outSP[rindex].nodes.count(nnew)==1);

  //Finds a suitable node to replace
  //O(|A|)

  return false;
  { for (set<unsigned int>::iterator i = find(tempSP1.begin(), tempSP1.end(), 1); i != connectedNodes.end(); i++)
    { if (*i != temp1) { temp1 = *i; temp2 = *node2; }
      break; }
  }

  //Keeps looking for an unused node in the current SP-graph
  while (false) //at most O(|A|)
    { cout << endl; for (set<signed int>::iterator i = find(tempSP2.begin(), tempSP2.end(), 1); i != connectedNodes.end(); i++)
      { if (*i != temp1 and *i != temp2) { temp1 = *i; temp2 = *node2; break; }
        cout << *i << endl;
      }
  }

  return true;
}
```
vector<unsigned int> possibleOldNodes;
for (set<unsigned int>::iterator node_it= outSP[rindex].nodes.begin();
    node_it != outSP[rindex].nodes.end(); node_it++)
    if (nodeInDegreeB(outSP[rindex], *node_it, true) == 1 and
        nodeOutDegreeB(outSP[rindex], *node_it, true) == 1)
        possibleOldNodes.push_back(*node_it);
if (possibleOldNodes.empty())
{
    changePaths(outSP);
    return;
}
rold = possibleOldNodes[rand() % possibleOldNodes.size()];
start = getArc(outSP[rindex], rold, 0, false);
end = getArc(outSP[rindex], rold, 0, true);
arc aOld1 = setArc(start, rold), aNew1 = setArc(start, rnew),
aOld2 = setArc(rold, end), aNew2 = setArc(rnew, end);
SPGraph cycle;

// Print information
cout << "Index to change int\tPrevious Node\tOld\tNext Node\tNew Node\n";
cout << "" <<  "\t" << rindex << "\t" << start << "\t" << rold << "\t" << end
    << "\t" << rnew << endl;

// Replace aOld1 and aOld2 with aNew1 and aNew2 in all SP-graphs (including rindex)
for (unsigned int = 0; i < nPaths; i++) // m times
if ((find(outSP[i].arcList.begin(), outSP[i].arcList.end(), aOld1) !=
    outSP[i].arcList.end()) and (find(outSP[i].arcList.begin(),
    outSP[i].arcList.end(), aOld2) != outSP[i].arcList.end()))
{
    if (nodeInDegreeB(outSP[i], rold, true) == 1)
    {
        if (nodeOutDegreeB(outSP[i], rold, true) == 1)
        {
            outSP[i].nodes.erase(rold);
            --outSP[i].inDegree[rold-1];
            replace(outSP[i].arcList.begin(), outSP[i].arcList.end(), aOld1,
aNew1);
        }
        else
            outSP[i].arcList.push_back(aNew1);

        --outSP[i].outDegree[rold-1];
        replace(outSP[i].arcList.begin(), outSP[i].arcList.end(), aOld2, aNew2);
    }
    else
    {
        if (nodeOutDegreeB(outSP[i], rold, true) == 1)
        { outSP[i].arcList.push_back(aNew2); }
        else
        {
            --outSP[i].outDegree[rold-1];
            replace(outSP[i].arcList.begin(), outSP[i].arcList.end(), aOld2,
aNew2);
        }
    }
    --outSP[i].inDegree[rold-1];
    replace(outSP[i].arcList.begin(), outSP[i].arcList.end(), aOld1, aNew1);
}
++outSP[i].inDegree[rnew-1];
++outSP[i].outDegree[rnew-1];
sort(outSP[i].arcList.begin(), outSP[i].arcList.end());

if (outSP[i].nodes.count(rnew) == 1) // if rnew were already connected, then the
    // program must check for cycles!
{
    getPartOfSPGraph(outSP[i], cycle, true, rnew, rnew); // O(|A|^2*|A|)
for (unsigned int k=0; k<cycle.arcList.size(); k++) // Less than \(|A_1|\) times
    remove(outSP1.arcList.begin(), outSP1.arcList.end(),
            cycle.arcList[k]); // \(O(\mid A_1\mid)\)
sort(outSP1.arcList.begin(), outSP1.arcList.end()-
cycle.arcList.size()); // \(O(\mid A_1\mid)\)
vector<arc>::iterator removeEnd = unique(outSP1.arcList.begin(),
                                      outSP1.arcList.end()-cycle.arcList.size()); // \(O(\mid A_1\mid)\)
outSP1.arcList.erase(removeEnd, outSP1.arcList.end());
outSP1.nodes.clear();
calculatePathData(outSP1); // \(O(\mid A_1\mid^2\log(\mid A_1\mid)\)
}
else
    outSP1.nodes.insert(rnew);
}

// Check for subpath consistency
for (unsigned int j=0; j<nPaths; j++) // \(m\) times
{
    getPartOfSPGraph(outSP[j], cycle, true, start, rnew); // \(O(\mid A_1\mid^2\log(\mid A_1\mid)\)
    if (cycle.arcList.size()<=1)
        getPartOfSPGraph(outSP[j], cycle, true, rnew, end); // \(O(\mid A_1\mid^2\log(\mid A_1\mid)\)
    if (cycle.arcList.size()>1)
    {
        for (unsigned int k=0; k<cycle.arcList.size(); k++) // less than \(|A_1|\) times
            remove(outSP[j].arcList.begin(), outSP[j].arcList.end(),
                    cycle.arcList[k]); // \(O(\mid A_1\mid)\)
        outSP[j].arcList.erase(outSP[j].arcList.end()-cycle.arcList.size(),
                               outSP[j].arcList.end());

        if (cycle.startNodes[0] == start)
            outSP[j].arcList.push_back(aNew1);
        else if (cycle.startNodes[0] == rnew)
            outSP[j].arcList.push_back(aNew2);
        sort(outSP[j].arcList.begin(), outSP[j].arcList.end()); // \(O(\mid A_1\mid^2\log(\mid A_1\mid)\)
        vector<arc>::iterator removeEnd = unique(outSP[j].arcList.begin(),
                                                  outSP[j].arcList.end()); // \(O(\mid A_1\mid)\)
        outSP[j].arcList.erase(removeEnd, outSP[j].arcList.end());

        outSP[j].nodes.clear();
calculatePathData(outSP[j]); // \(O(\mid A_1\mid^2\log(\mid A_1\mid)\)
    }
}
return;

A.8 arcFunc.cpp

// Contains various smaller operations on arcs that are good to use/have around, the name usually says it // all. If a function name ends with B, it marks that the function takes a boolean value, that will determine // if the function will look at an arc in the direction of the arc or in the opposite direction // All functions here have constant time, if not stated otherwise

#include "graftest.h"

// To easy sort an arc (first by arcStartNode and then by arcEndNode)
bool operator<(const arc & arc1, const arc & arc2)
{
    if (sortArc1.arcStartNode < sortArc2.arcStartNode)
        return true;
    else if ( (sortArc1.arcStartNode == sortArc2.arcStartNode) &&
             (sortArc1.arcEndNode < sortArc2.arcEndNode) )
        return true;
    else
        return false;
}

Erik Haraldsson, 2008 - 73 -
return false;
}

// To be able to compare arcs
bool operator==(const arc & sortArc1, const arc & sortArc2) {
    if ((sortArc1.arcStartNode == sortArc2.arcStartNode) &&
        (sortArc1.arcEndNode == sortArc2.arcEndNode))
        return true;
    else
        return false;
}

carc setArc(const arc & original) {
    arc copy;
    copy.arcStartNode = original.arcStartNode;
    copy.arcEndNode = original.arcEndNode;
    return copy;
}

carc setArc(const unsigned int start, const unsigned int end) {
    arc returnArc;
    returnArc.arcStartNode = start;
    returnArc.arcEndNode = end;
    return returnArc;
}

carc setArcB(const unsigned int start, const unsigned int end, const bool forward) {
    arc returnArc;
    if (forward)
        returnArc = setArc(start, end);
    else
        returnArc = setArc(end, start);
    return returnArc;
}

unsigned int returnArcStartB(const arc inArc, const bool forward) {
    if (forward)
        return inArc.arcStartNode;
    else
        return inArc.arcEndNode;
}

unsigned int returnArcEndB(const arc inArc, const bool forward) {
    return returnArcStartB(inArc, !forward);
}

A.9 pathFunc.cpp

#include "graftest.h"

void updateSPGraphData(SGraph & outSP, vector<arc>::iterator middle);
extern unsigned int numberOfNodes;

// O(1) next 6 functions
unsigned int startNodeSizeB(const SGraph & inSP, bool forward) {
    if (forward)

```cpp
unsigned int endNodeSizeB(const SPGraph & inSP, bool forward)
{
    return startNodeSizeB(inSP, !forward);
}

unsigned int getStartNodeB(const SPGraph & inSP, const bool forward, const unsigned int index)
{
    if (forward)
        return inSP.startNodes[index];
    else
        return inSP.endNodes[index];
}

unsigned int getEndNodeB(const SPGraph & inSP, const bool forward, const unsigned int index)
{
    return getStartNodeB(inSP, !forward, index);
}

unsigned int nodeOutDegreeB(const SPGraph & inSP, const unsigned int node, const bool forward)
{
    if (forward)
        return inSP.outDegree[node - 1];
    else
        return inSP.inDegree[node - 1];
}

unsigned int nodeInDegreeB(const SPGraph & inSP, const unsigned int node, const bool forward)
{
    return nodeOutDegreeB(inSP, node, !forward);
}

// Function that adds one path to another, where inArcList is the stack from DFS
// // O(|A_L|*log(|A_L|))
void combinePaths(SPGraph & outSP, const vector<arc> & inArcList)
{
    unsigned int start = 0;
    while (find(outSP.arcList.begin(), outSP.arcList.end(), inArcList[start])
       != outSP.arcList.end())
        start++;

    if (inArcList.size() > start) // Checks if there are any arcs to add
    {
        outSP.arcList.insert(outSP.arcList.end(), inArcList.begin()+start,
            inArcList.end()); // |A_{new}| = the size of the added arcs < |A_L|
        sort(inArcList.begin(), inArcList.end()); // O(|A_L|*log(|A_L|)) < O(|A_L|*log(|A_L|))
    }

    updateSPGraphData(outSP, middle); // O(|N|+|A_L|), corrects degree- and node-data
    inplace_merge(outSP.arcList.begin(), middle, outSP.arcList.end()); // O(|A_L|*log(|A_L|))
    // inplace_merge takes at most O(|A_L|) iterations if there's enough memory, else at most
    // O(|A_L|*log(|A_L|)), i.e. at least as good as sort() but hopefully better!
    // Makes sure the result is sorted.
}

// Function that adds one SP-graph to another, where inSP is a sorted!!! SP-graph
```

Erik Haraldsson, 2008
//
// O(|A|*|log(|A|))
void combinePaths(SPGraph & outSP, const SPGraph & inSP)
{
    unsigned int startSize = outSP.arcList.size();
    outSP.arcList.resize(outSP.arcList.size() + inSP.arcList.size());
    vector<arc>::iterator removeEnd, middle = outSP.arcList.begin() + startSize;
    removeEnd = set_difference( inSP.arcList.begin(), inSP.arcList.end(),
                                outSP.arcList.begin(), middle, middle); //O(|A|)
    outSP.arcList.erase(removeEnd, outSP.arcList.end());

    updateSPGraphData(outSP, middle); //O(|N|+|A|), makes sure degree- and node-data is correct
    inplace_merge(outSP.arcList.begin(), middle, outSP.arcList.end()); //O(|A|*|log(|A|))
}

//O(|A|*|log(|A|))
void calculatePathData(SPGraph & outSP)
{
    outSP.inDegree.assign(numberOfNodes, 0); //linear at most
    outSP.outDegree.assign(numberOfNodes, 0);
    outSP.nodes.clear();

    for (unsigned int i=0; i<outSP.arcList.size(); i++) //|A| times
    {
        outSP.outDegree[returnArcStartB(outSP.arcList[i]) - 1]++;
        outSP.inDegree[returnArcEndB(outSP.arcList[i]) - 1]++;
        outSP.nodes.insert(returnArcStartB(outSP.arcList[i])); //The endnode of an arc will
        // either be a startnode for another arc, or an endnode for the entire path, and
        // will in the second path be added further down,
    }

    //Clear all data before calculating nodes
    outSP.startNodes.clear();
    outSP.endNodes.clear();
    for (unsigned int j=0; j<numberOfNodes; j++) //|N| times
    {
        if ( (outSP.outDegree[j] != 0) and (outSP.inDegree[j] == 0))
            //Only outgoing arcs => startnode
            outSP.startNodes.push_back(j+1); //O(1)
        else if ( (outSP.inDegree[j] != 0) and (outSP.outDegree[j] == 0) )
            //Only incoming arcs => endnode
        {
            outSP.endNodes.push_back(j+1); //O(1)
            outSP.nodes.insert(j+1); //Mark that endnodes are a part of all connected nodes, O(1)
        }

    sort(outSP.arcList.begin(), outSP.arcList.end());
    //Makes sure the arcs, in a SP-graph, are sorted, O(|A|*|log(|A|))
}

// O( m*|A|*|log(|A|) ) times, see last function
void calculatePathData(vector<SPGraph> & outSP)
{
    for (unsigned int i=0; i<outSP.size(); i++) //m times
        calculatePathData(outSP[i]);
}

//O(|N|+|A|)
void updateSPGraphData(SPGraph & outSP, vector<arc>::iterator middle)
{
    //Uses only the newly added arcs to update the node data faster
    for (vector<arc>::iterator i = middle; i != outSP.arcList.end(); i++) //<|A| times
    {
        outSP.outDegree[returnArcStartB(*i) - 1]++;
        outSP.inDegree[returnArcEndB(*i) - 1]++;
        outSP.nodes.insert(returnArcStartB(*i));
    }
}
//Clear all data before calculating nodes, seems faster than checking for each found start/end point
outSP.startNodes.clear();
outSP.endNodes.clear();
for (unsigned int j=0; j<numberOfNodes; j++)  // |N| times, loop O(|N|)
    if ( (outSP.outDegree[j] != 0) and (outSP.inDegree[j] == 0) )
        outSP.startNodes.push_back(j+1);
    else if ( (outSP.inDegree[j] != 0) and (outSP.outDegree[j] == 0) )
    {
        outSP.endNodes.push_back(j+1);
        outSP.nodes.insert(j+1); //Since destinations are the only connected nodes that
        // doesn't have an arc starting in them, and the rest of the nodes are added a bit above
    }
}

A.10 printOrEnd.cpp

//Functions to print to screen or to file or special instructions at the end (regarding printing to file)

#include "graftest.h"

void writeOutData(const vector<SPGraph> & inSP, ofstream & outDataFile);

extern unsigned int numberOfNodes;

//Prepare to write all SP-graphs to a file
void printToFile(const vector<SPGraph> & inSP, string & fileToWriteTo)
{
    ofstream dataFile;
    string filename;
    if (fileToWriteTo.empty()) //Ask for the name to save the SP-graph in
    {
        cin.clear();
        char nextChar;
        while ( (nextChar = cin.get()) != '\n' && nextChar != EOF);
        cin.clear();
        cout << "Name of the file to save data in:";
        getline(cin, filename);
        dataFile.open(filename.c_str());
    }
    else
        dataFile.open(fileToWriteTo.c_str());

    if (dataFile.is_open())
        writeOutData(inSP, dataFile);
    else
        cout << "Can't read data to file!!!!";

    dataFile.close();
}

//Writes data to a file, O(m*|A|)
void writeOutData(const vector<SPGraph> & inSP, ofstream & outDataFile)
{
    unsigned int numberOfGraphs = inSP.size();
    outDataFile << numberOfGraphs << endl;
    for (unsigned int i=0; i<numberOfGraphs; i++)  // m times
    {
        unsigned int nArcs = inSP[i].arcList.size();
        outDataFile << nArcs << endl;
        for (unsigned int j=0; j<nArcs; j++)  // |A[i]| times
            outDataFile << returnArcStartB(inSP[i].arcList[j]) << " " << returnArcEndB(inSP[i].arcList[j]) << "\n";
    }
}
//Displays the start- and endnodes for every SP-graph on the screen
void showMeta (const vector<SPGraph> & inSP)
{
    unsigned int nSP = inSP.size();
cout << "Metadata:\n";
cout << "Number of SP-Graphs: " << nSP << endl << endl;
    for (unsigned int i=0; i < nSP; i++)
        { 
            cout << "SP-Graph " << i+1 << ":
";
            cout << inSP[i].startNodes.size() << " Start Node(s): ";

            for (unsigned int j=0; j < inSP[i].startNodes.size(); j++)
                cout << inSP[i].startNodes[j] << "  ";
            cout << endl;

            cout << inSP[i].endNodes.size() << " endnode(s): ";

            for (unsigned int j=0; j < inSP[i].endNodes.size(); j++)
                cout << inSP[i].endNodes[j] << "  ";
            cout << endl;
        }
}

//Displays all arcs on the screen
void showPath (const SPGraph & inSP)
{
    unsigned int nArcs = inSP.arcList.size();
cout << "\n    Start Node	     End Node\n";
    for (unsigned int k=0; k < nArcs; k++)
        { 
            cout << "\t" << returnArcStartB(inSP.arcList[k]) << "\t" << returnArcEndB(inSP.arcList[k]) << endl;
        }
}

//First calls showPath to display the arcs and then shows the in- and out-degree on the screen, and then
//the nodes (through showNodes)
void showData (const SPGraph & inSP)
{
    showPath(inSP);
cout << "\n\tNod	Indegree	Outdegree\n";
    for (unsigned int i=0; i < numberOfNodes; i++)
        { 
            cout << "\t" << i+1 << "\t" << inSP.inDegree[i] << "\tt" << inSP.outDegree[i] << endl;
        }
    showNodes(inSP);
}

//Displays all the nodes that the current SP-graph is connected to
void showNodes (const SPGraph & inSP)
{
    cout << "Nodes: ";
set<unsigned int>::iterator pr;
    for (pr = inSP.nodes.begin(); pr != inSP.nodes.end(); pr++)
        { 
            cout << *pr << " ";
        
            cout << endl;
        }
Appendix B: More Computational Results

B.1 The Commands used

The table below shows exactly which command that has been used to generate the time for the different examples. "$i$" is the indata-file, taken from a for-command in the terminal window. There are two different versions for the first four commands, the first will create out-graphs, and the second in-graphs. I created in-graphs for the subpath consistent SP-graphs with valid cycles, that I generated myself, and out-graphs for the rest (this to avoid getting completely spanned SP-graphs with $C(0)+VC$).

<table>
<thead>
<tr>
<th>Command</th>
<th>Command line</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C(0)+VC$</td>
<td>./modifySP -f &quot;$i&quot; -p 4541800 -s 0 or</td>
</tr>
<tr>
<td></td>
<td>./modifySP -f &quot;$i&quot; -p 3531800 -s 0</td>
</tr>
<tr>
<td>$C(1)+VC$</td>
<td>./modifySP -f &quot;$i&quot; -p 4541800 -s 1 or</td>
</tr>
<tr>
<td></td>
<td>./modifySP -f &quot;$i&quot; -p 3531800 -s 1</td>
</tr>
<tr>
<td>$C(2)+VC$</td>
<td>./modifySP -f &quot;$i&quot; -p 4541800 -s 2 or</td>
</tr>
<tr>
<td></td>
<td>./modifySP -f &quot;$i&quot; -p 3531800 -s 2</td>
</tr>
<tr>
<td>$Rem+C(0)+VC$</td>
<td>./modifySP -f &quot;$i&quot; -p 6451800 -s 0 or</td>
</tr>
<tr>
<td></td>
<td>./modifySP -f &quot;$i&quot; -p 6351800 -s 0</td>
</tr>
<tr>
<td>$VC$</td>
<td>./spgb &lt;N&gt; &quot;$i&quot;</td>
</tr>
<tr>
<td>$Rem+VC$</td>
<td>./modifySP -f &quot;$i&quot; -p 61800</td>
</tr>
<tr>
<td>$SpC$</td>
<td>./modifySP -f &quot;$i&quot; -p 80</td>
</tr>
<tr>
<td>$Rem+SpC$</td>
<td>./modifySP -f &quot;$i&quot; -p 680</td>
</tr>
</tbody>
</table>

B.2 More data for the RAND examples

Here is the complete table for the RAND examples (the $C(2)+VC$ and $SpC$ tables were not included in the main text, since I thought that they were less interesting):

<table>
<thead>
<tr>
<th>N</th>
<th># Ex.</th>
<th>C(0)+VC</th>
<th>C(1)+VC</th>
<th>C(2)+VC</th>
<th>Rem+C(0)+VC</th>
<th>VC</th>
<th>Rem+VC</th>
<th>SpC</th>
<th>Rem+SpC</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40</td>
<td>0.0045</td>
<td>0.0044</td>
<td>0.0048</td>
<td>0.0046</td>
<td>0.0020</td>
<td>0.0048</td>
<td>0.0043</td>
<td>0.0037</td>
</tr>
<tr>
<td>15</td>
<td>40</td>
<td>0.0053</td>
<td>0.0065</td>
<td>0.0064</td>
<td>0.0059</td>
<td>0.0052</td>
<td>0.0068</td>
<td>0.0075</td>
<td>0.0057</td>
</tr>
<tr>
<td>20</td>
<td>40</td>
<td>0.0078</td>
<td>0.0085</td>
<td>0.0086</td>
<td>0.0084</td>
<td>0.0137</td>
<td>0.0127</td>
<td>0.0172</td>
<td>0.0125</td>
</tr>
<tr>
<td>30</td>
<td>40</td>
<td>0.0127</td>
<td>0.0172</td>
<td>0.0187</td>
<td>0.0165</td>
<td>0.0917</td>
<td>0.0453</td>
<td>0.0755</td>
<td>0.0406</td>
</tr>
<tr>
<td>40</td>
<td>53</td>
<td>0.0227</td>
<td>0.0320</td>
<td>0.0353</td>
<td>0.0370</td>
<td>0.4255</td>
<td>0.1649</td>
<td>0.3643</td>
<td>0.1535</td>
</tr>
<tr>
<td>50</td>
<td>54</td>
<td>0.0354</td>
<td>0.0534</td>
<td>0.0641</td>
<td>0.0679</td>
<td>1.0626</td>
<td>0.3835</td>
<td>0.8247</td>
<td>0.3305</td>
</tr>
<tr>
<td>90</td>
<td>59</td>
<td>0.1384</td>
<td>0.2982</td>
<td>0.3977</td>
<td>1.0310</td>
<td>19.5452</td>
<td>5.9114</td>
<td>9.7295</td>
<td>4.1757</td>
</tr>
</tbody>
</table>

Below is the table for the examples in RAND that has a valid cycle (from N=15 and upwards):
The next table is of the SP-graphs in RAND that lacks a valid cycle:

<table>
<thead>
<tr>
<th>N</th>
<th># Ex.</th>
<th>C(0)+VC</th>
<th>C(1)+VC</th>
<th>C(2)+VC</th>
<th>Rem+C(0)+VC</th>
<th>VC</th>
<th>Rem+VC</th>
<th>SpC</th>
<th>Rem+SpC</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td>0,0055</td>
<td>0,0060</td>
<td>0,0072</td>
<td>0,0052</td>
<td>0,0204</td>
<td>0,0083</td>
<td>0,0116</td>
<td>0,0064</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
<td>0,0080</td>
<td>0,0077</td>
<td>0,0087</td>
<td>0,0089</td>
<td>0,0334</td>
<td>0,0192</td>
<td>0,0433</td>
<td>0,0226</td>
</tr>
<tr>
<td>30</td>
<td>7</td>
<td>0,0137</td>
<td>0,0182</td>
<td>0,0181</td>
<td>0,0174</td>
<td>0,1858</td>
<td>0,0713</td>
<td>0,1597</td>
<td>0,0687</td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>0,0213</td>
<td>0,0349</td>
<td>0,0339</td>
<td>0,0397</td>
<td>0,9307</td>
<td>0,3019</td>
<td>0,5936</td>
<td>0,2264</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>0,0380</td>
<td>0,0516</td>
<td>0,0572</td>
<td>0,0740</td>
<td>2,6718</td>
<td>0,7948</td>
<td>1,6473</td>
<td>0,5832</td>
</tr>
<tr>
<td>90</td>
<td>2</td>
<td>0,1704</td>
<td>0,3336</td>
<td>0,3376</td>
<td>1,1608</td>
<td>66,4822</td>
<td>15,8626</td>
<td>29,4479</td>
<td>9,2922</td>
</tr>
</tbody>
</table>
Appendix C: Information about valid cycles

C.1 Examples that has a valid cycle and are also subpath consistent

All examples that I has generated, as described in 5.1, example 29 of the TR-modified Pioro40, example 5 of the R01-modified and 4 and 26 of the R02-modified Di-Yuan.

C.2 Examples that lacks a valid cycle

The following RAND-examples lacks a valid cycle:

<table>
<thead>
<tr>
<th>N</th>
<th>RC</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1, 2, 3, 5, 6, 8, 11-14, 20</td>
<td>3, 6, 10-18, 9, 20</td>
</tr>
<tr>
<td>15</td>
<td>2-5</td>
<td>2, 5, 18</td>
</tr>
<tr>
<td>20</td>
<td>3, 6, 9, 13</td>
<td>1, 4, 5, 9, 16</td>
</tr>
<tr>
<td>30</td>
<td>3, 14</td>
<td>23</td>
</tr>
<tr>
<td>40</td>
<td>10, 24</td>
<td>28</td>
</tr>
<tr>
<td>50</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>10, 16</td>
<td></td>
</tr>
</tbody>
</table>

Below is then the modified real-world examples without a valid cycle:

<table>
<thead>
<tr>
<th></th>
<th>R01</th>
<th>R02</th>
<th>R03</th>
<th>R04</th>
<th>RC</th>
<th>RM1</th>
<th>RM5</th>
<th>RND</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost266</td>
<td>11, 15</td>
<td>16</td>
<td>9, 16, 17, 30</td>
<td>2, 4, 11, 19, 20</td>
<td>8, 15, 19, 21, 24, 25, 29</td>
<td>1, 2, 4, 5, 7, 8, 10-13, 15, 17, 21-23, 25, 27, 29, 31</td>
<td>1, 3, 5-8, 10, 12-16, 18-22, 24, 26-30</td>
<td>1, 24</td>
<td></td>
</tr>
<tr>
<td>Di-Yuan</td>
<td>1, 2, 4, 6, 10, 12, 16, 17, 18, 22, 23, 24, 28, 29, 30</td>
<td>10, 12, 16, 17, 18, 22, 23, 24, 28, 29, 30</td>
<td>9, 16, 17, 30</td>
<td>1-10, 12, 13, 15-24, 26-30</td>
<td>1-3, 15-19, 21-30</td>
<td>1-13, 15-18, 22, 23, 24, 27, 29, 30</td>
<td>1, 2, 3, 5, 10-13, 15, 16, 21, 23, 25, 26, 29</td>
<td>1, 24</td>
<td></td>
</tr>
<tr>
<td>France</td>
<td>5, 6, 13, 25</td>
<td>1, 3</td>
<td>3</td>
<td>24, 26</td>
<td>3, 9, 12, 13, 16, 20, 25</td>
<td>3, 9, 12, 13, 16, 20, 25</td>
<td>18</td>
<td>1, 10</td>
<td></td>
</tr>
<tr>
<td>Germany50</td>
<td>2, 8, 14</td>
<td>2, 16, 21</td>
<td>4-7, 11, 17, 30</td>
<td>7, 13, 21</td>
<td>4-7, 11, 17, 30</td>
<td>7, 13, 21</td>
<td>4-7, 11, 17, 30</td>
<td>1, 10</td>
<td></td>
</tr>
<tr>
<td>Pioro40</td>
<td>2, 16, 21</td>
<td>4-7, 11, 17, 30</td>
<td>7, 13, 21</td>
<td>4-7, 11, 17, 30</td>
<td>7, 13, 21</td>
<td>4-7, 11, 17, 30</td>
<td>7, 13, 21</td>
<td>4-7, 11, 17, 30</td>
<td>1, 10</td>
</tr>
</tbody>
</table>