EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Efficient solution methods for the directed Steiner tree problem

van der Hoeven, Saskia P.

Award date:
2023

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 18. Mar. 2025

Department of Mathematics and Computer Science

ASML

Efficient solution methods for the directed Steiner tree problem

Master thesis

Author:
Saskia van der Hoeven
(0997356)

TU/e supervisors:
Dr. Christopher Hojny
Dr. Judith Keijsper
Company supervisor:
Laurens Versluis
Committee members:
Rob Eggermont

9th March 2023

Abstract

This thesis focuses on a problem in automatic data processing, posed by ASML. The problem can be modelled as a mathematical problem, where, given a directed graph, a set of start vertices and a set of terminals, one must find a minimum-weight subgraph in which there exist paths from start vertices to terminals. All terminals must be contained in the subgraphs, but not all terminals need to be. We prove that the decision variant of this problem is NP-complete, and that the problem can be modelled as a directed Steiner tree problem. We develop, analyse, and test several approximation algorithms, a combinatorial exact algorithm, and several linear programming approaches. We find that the fastest approximation algorithm is one that takes the union of the shortest paths from a root vertex to the terminals. The average approximation ratio of this algorithm is rather good for ASML's data. The combinatorial exact algorithm does not perform well. Two of the linear programming approaches complete quite fast for algorithms that solve the problem to optimality. These are a flow based and a path based linear programming formulation.

Contents

1 **Preliminaries** 1

2 **Introduction** 3

3 **Problem description and mathematical model** 4

3.1 Problem description 4

3.2 Modelling the problem as a directed Steiner tree problem 6

3.2.1 Pre-processing 6

3.2.2 Post-processing 7

3.3 Complexity of the problem 8

4 **Approximation algorithms** 10

4.1 Shortest paths algorithm 10

4.2 Shortest bunches algorithm 13

4.2.1 One bunch 13

4.2.2 Multiple bunches 16

4.3 Greedy algorithm 20

5 **Exact algorithm** 24

6 **ILP based solution methods** 28

6.1 Directed cut formulation 28

6.1.1 Constraints based on minimal sets of arcs 31

6.1.2 Adding constraints along the way 33

6.2 Flow based formulation 36

6.3 Path based formulation 38

7 **Testing the algorithms** 40

7.1 ASML graph 40

7.1.1 Input data 40

7.1.2 Computational results: random arc weights 41

7.1.3 Computational results: unweighted graph 47

7.2 Randomly generated graphs 50

7.2.1 Computing random graphs 50

7.2.2 Computational results 51

8 Discussion 55
8.1 Approximation algorithms 55
8.2 Exact (combinatorial) algorithm 56
8.3 LP based solution methods 56
8.4 Problem specific advice 57
9 Conclusion 58
References 60

1 Preliminaries

To ensure the reader has the necessary background information for this thesis, we discuss some relevant mathematical preliminaries. In this section, we will state various definitions that will be important for the understanding of this thesis. First of all, we define the *weight* of a graph.

Definition 1 (Weight of a graph). *Let $G = (V, E)$ be an undirected/directed graph with weight function $w : E \to \mathbb{R}$ on the edges/arcs. The weight of G is defined as $w(G) := \sum_{e \in E} w(e)$.*

Then, we define a *tree*.

Definition 2 (Tree). *A tree is an undirected graph that is connected and acyclic.*

Using this definition, we can define the *Steiner tree*.

Definition 3 (Steiner tree). *Let $G = (V, E)$ be a weighted undirected graph, with weight function $w : E \to \mathbb{R}_{\geq 0}$ defined on the edges. Furthermore, let $X \subseteq V$ be a subset of the vertices. A Steiner tree is a tree that is a subgraph of G that contains all vertices in X. To be more precise: a Steiner tree $T = (V', E')$ is a tree such that $V' \subseteq V$, $E' \subseteq E$ and $X \subseteq V'$. Here, we call each $x \in X$ a terminal.*

The Steiner tree problem in graphs, mostly referred to as the Steiner tree problem, is the problem of finding a minimum weight Steiner tree for the given set of terminals. Several variants of this problem exist, including the *directed Steiner tree problem*. For this, we first define r-arborescences.

Definition 4 (r-Arborescence). *An r-arborescence is a directed acyclic graph whose underlying undirected graph is a tree, such that exactly one vertex, r, has in-degree zero and all other vertices have in-degree one. Vertex r is called the root.*

Equivalently, we can define an r-arborescence as a directed graph in which there is exactly one directed path from the root r to any other vertex. Furthermore, we define the *shortest-path-r-arborescence*.

Definition 5 (Shortest-path-r-arborescence). *Let $D = (V, A)$ be a weighted directed graph with weight function $w : A \to \mathbb{R}_{\geq 0}$ defined on the arcs. Furthermore, let $r \in V$. Now, define $W \subseteq V$ to be the set of vertices $w \in V$ for which there exists a path from r to w in D, and let c_w be the length of the shortest path from r to w in D for some vertex $w \in W$. A shortest-path-r-arborescence of D is a subgraph of D that contains exactly one path from r to w for each vertex $w \in W$, each of which has length c_w.*

Now, we can define the *directed Steiner tree*.

Definition 6 (Directed Steiner tree). *Let $D = (V, A)$ be a weighted directed graph with weight function $w : A \to \mathbb{R}_{\geq 0}$ on the arcs. Furthermore, let $r \in V$ and $X \subseteq V$. A directed Steiner tree (DST) is an r-arborescence that is a subgraph of D that contains all vertices $x \in X$. To be more precise: a directed Steiner tree $T = (V', A')$ is an r-arborescence such that $V' \subseteq V$, $A' \subseteq A$ and $X \subseteq V'$. Here, we call each $x \in X$ a terminal.*

The *directed Steiner tree problem* in directed graphs is the directed counterpart of the Steiner tree problem in undirected graphs.

Definition 7 (Directed Steiner tree problem). *Let $D = (V, A)$ be a weighted directed graph with weight function $w : A \to \mathbb{R}_{\geq 0}$ on the arcs. Furthermore, let $r \in V$ and $X \subseteq V$. The directed Steiner tree problem is the problem of finding a minimum weight r-arborescence that is a subgraph of D and contains all terminals. To be more precise: the directed Steiner tree problem is the problem of finding an r-arborescence $T = (V', A')$ such that $V' \subseteq V$, $A' \subseteq A$, $X \subseteq V'$, and there exists no r-arborescence $\hat{T} = (V', A')$ that also satisfies these properties and has weight $w(\hat{T}) < w(T)$.*

We assume some preliminary knowledge on the concept of NP-hardness. On this topic, we do state the definition of the decision variant of a problem.

Definition 8 (Decision variant of an optimisation problem). *Let an optimisation problem Y with vector of variables y in solution space $\mathcal{S}$, objective function $\min_{y \in \mathcal{S}} f(y)$ and set of constraints C be given. Furthermore, let a number $t \in \mathbb{R}$ be given. The decision variant of Y is the problem of deciding whether there exists a solution y^* that satisfies all constraints in C, such that $f(y^*) \leq t$.*

2 Introduction

Since ASML was founded in 1984, the company has become a key player in the global semiconductor industry. ASML designs, develops and produces lithography systems, which are essential in chip manufacturing. A large part of ASML's resources are committed to improving their existing systems and developing new ones. The development of such intricate systems, especially at the scale at which ASML performs, asks for a large and complex software architecture. Therefore, different departments need to work together. Querying data in this network of data might not be straightforward, since the required data might not be directly connect to the input that is offered. Therefore, ASML uses a context model to describe how their data is structured. This is a visual model that can be expressed in a graph. The problem that ASML poses is how to efficiently query data in their model. In this thesis, we will see that the problem posed by ASML can be modelled as a directed Steiner tree problem. The main goal of this thesis is to investigate different techniques for solving the directed Steiner tree problem, and analysing how these techniques perform on the problem presented by ASML.

Named after Jakob Steiner, Steiner tree problems are a class of problems in combinatorial optimisation that focus on connecting a set of points while minimising the weight of the solution [Prömel and Steger(2012)]. There are many variants of the problem, the most well-known one being the problem that is often referred to as the Steiner tree problem. This is a problem defined on undirected graphs, where the goal is to compute a minimum weight tree that contains a given set of vertices. This thesis will focus on a version of the problem for directed graphs, called the directed Steiner tree problem. This is a problem where, given a root and a set of vertices, we want to find the cheapest way to connect the root to these vertices.

As will be shown in Section 3.3, the directed Steiner tree problem, and therefore ASML's problem, is NP-complete. For ASML it is important for its software to be both accurate and fast. This means that ASML needs an efficient and reliable algorithm for the querying of data. However, for a problem that is NP-complete, it can be challenging to develop such an algorithm. For NP-complete problems, run times of algorithms that solve the problem to optimality can quickly increase as the magnitude of the problem increases. Therefore, this thesis will explore various methods of solving and approximating ASML's problem, and analysing how these methods perform in terms of time and accuracy.

To this end, we start by describing the problem given by ASML more precisely in the next section. We first state the origin of the problem and its mathematical formulation in Section 3.1. In Section 3.2 we describe how the problem can be translated to a directed Steiner tree problem, and we will see that we can therefore solve the problem by solving the directed Steiner tree problem. In Section 3.3, we prove that an important theorem on the complexity of the problem. Then, we develop and investigate several approximation algorithms in Section 4, and an exact combinatorial algorithm in Section 5. Furthermore, we discuss several linear programming based methods of solving the directed Steiner tree problem in Section 6. Finally, we test the algorithms that were discussed in Section 7. We first analyse the data provided by ASML and discuss the computational results of the algorithms for this graph in Section 7.1. Then, to be able to test the algorithms more extensively, in Section 7.2 we generate graphs of different sizes randomly and test the algorithms on these graphs. In Section 8, we discuss the various algorithms and their results, and argue how they could be further improved. Finally, we conclude with a short summary, the most important findings and some recommendations in Section 9.

3 Problem description and mathematical model

In this section, we describe the problem posed by ASML, and formulate it mathematically. We note that this problem is very similar to a well-known mathematical problem: the directed Steiner tree problem. We model the ASML problem as a directed Steiner tree problem, such that we can take ideas from research that has already been conducted on the topic. To this end, we describe the pre-processing and post-processing steps that translate ASML's problem to a directed Steiner tree problem and back.

3.1 Problem description

The problem posed by ASML is a problem in automatic data processing. ASML has an analytics platform designed to let both internal an external parties, i.e. employees and customers, access data from ASML products. This platform can, for example, be used by customers for monitoring their products and for fault detection. One of the functions of the analytics platform is the querying of data: a user of the platform can give some data as input and request data that is in some way, possibly through some other data, related to the input data. It is not always easy to know where to get the requested data from. Therefore, ASML wants to automate this process in a way that one can quickly retrieve the data one needs.

To be able to automate the querying of data, ASML creates a graph based on how data is connected. This is visualised in Figure 1. The first diagram visualises a query. A user enters a set of data as input, shown in blue. In the example, the input data consists of the two categories c_1 and c_2. The user requests the data c_8 and c_{10}, shown in green. The input dataset is linked to other datasets, which are in turn linked to other datasets. This is visualised in the second diagram. The challenge of the analytics tool is to find a way to go from the input data to the requested output data. This happens through connections between datasets. These connections can be different kinds of relations, which do not always go both way. That means, if one can retrieve dataset a from dataset b, that does not always mean one can also retrieve dataset b from dataset a. By visualising the data in this way, we see that we can model the data and the connections as a graph, as shown in the third diagram. Here, the different data categories are vertices, and the connections between different data categories to arcs. The problem that needs to be solved is to find, given a set of start and end vertices, a way to connect the start vertices to the end vertices via a set of paths. Of course, there might be different paths that lead to a requested vertex. To ensure one gets the data quickly, the objective is to choose the paths that connect the input data to the requested data in the "cheapest" way. That is, we aim to use as few connections as possible.

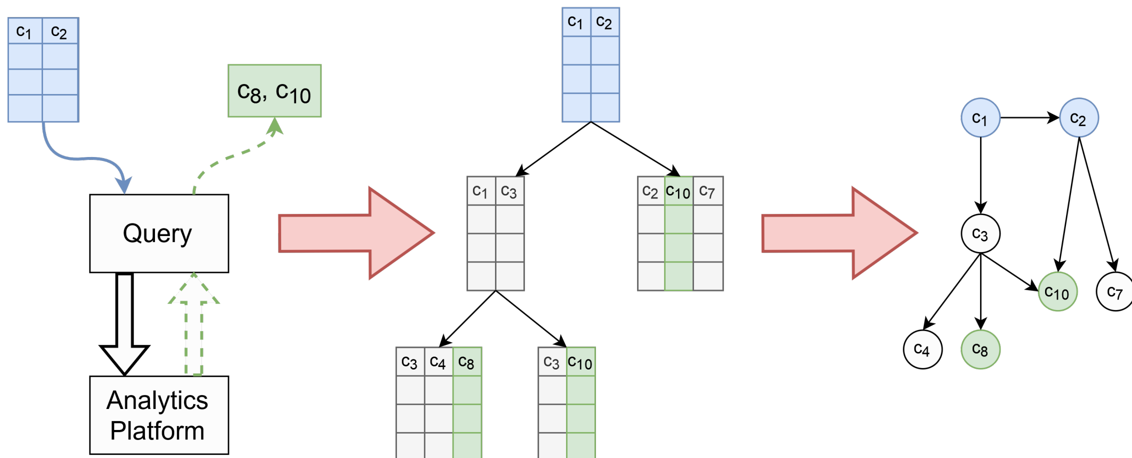


Figure 1: Visualisation of translation steps of a query in the analytics platform to a graph.

A big disadvantage to the above model is that, since the number of datasets can become very large, it can cost a lot of memory to save all the data in one place. Furthermore, allocating data in such a huge model, can cost a lot of time. Therefore, we will not create a graph based on the individual instances, but based on

the context model. The context model only considers categories of datasets, rather than actual datasets. This context model can be translated to a graph by looking at how the instances are connected between categories. When constructing this graph, we may choose to put weights on the arcs. For example, weights corresponding to the amount of data that needs to be processed if said relation would be used. If more data needs to be processed, it will cost more time to use this relation, which means we want this arc to be less attractive. It will always cost some amount of time to use some relation, and therefore the weights of the arcs will be strictly positive. The problem to solve in this graph is as follows.

Let a directed graph with positive arc weights be given. The task is to connect the start vertices to the end vertices, such that the sum of the weights of the arcs used is minimal. Here, not all start vertices need to be used, but all end vertices must be included. Furthermore, the aim is to have an efficient solution method.

To state the problem more precisely, we introduce a mathematical notation.

Definition 9. *Let a weighted directed multigraph $D = (V, A)$ be given. Furthermore, let a set of start vertices S, and a set of end vertices, called terminals, X be given. The weights $w(a)$ of the arcs are defined according to weight function $w : A \to \mathbb{R}_{>0}$. The* ***ASML problem*** *is to find a minimum-weight subgraph $T = (V', A')$ of D such that, for each terminal $x \in X$, there is a start vertex $s \in S$ such that T contains a path from s to x.*

The goal of the project is to develop an efficient algorithm to solve or approximate the ASML problem.

In addition to the problem described above, we have a few assumptions. First of all, we assume that users will always request data that is available given the input data. That means, we assume that for every terminal $x \in X$, there exists a start vertex $s \in S$ for which D contains a path from s to x. Furthermore, as stated before, we assume all arc weights to be strictly positive. The graph will never contain "free" arcs, since it will always cost some amount of time to go from one category of datasets to the next.

3.2 Modelling the problem as a directed Steiner tree problem

We immediately see that the ASML problem is very similar to the directed Steiner tree problem. In this section, we will show how the ASML problem can be modelled as a directed Steiner tree problem. First, we will explain which pre-processing steps need to be taken in order to transform an instance of the ASML problem to an instance of the directed Steiner tree problem. The, we will explain how we can transform a solution to the directed Steiner tree problem to a solution of the ASML problem.

3.2.1 Pre-processing

To model the ASML problem as a directed Steiner tree problem, we must be able to transform any instance of the ASML problem to an instance that satisfies the conditions stated in Definition 6. As the input for the ASML problem, let a directed multigraph $D_{ASML} = (\hat{V}, \hat{A})$, a set of start vertices S, and a set of terminals X be given. The weights $\hat{w}(a)$ of the arcs are defined according to weight function $\hat{w} : \hat{A} \to \mathbb{R}_{>0}$.

First of all, note that the input graph for the ASML problem may be a multigraph, which means the graph can have multiple arcs from a vertex to another vertex. For any ordered pair of vertices (u, v), for $u, v \in \hat{V}$, define $\hat{A}_{(u,v)}$ to be the set of arcs from u to v. Fix two vertices u and v for which $|\hat{A}_{(u,v)}| \geq 2$, and let two of those arcs be a_1 and a_2, with $w(a_1) < w(a_2)$. An optimal solution to the ASML problem will never contain arc a_2. Therefore, we can redefine the graph by only keeping the lowest weight arc in the case there are parallel arcs from one vertex to another. To be more precise: let $D_1 = (\hat{V}, A_1)$ be the graph that is obtained from D_{ASML} by, for each ordered pair of vertices (u, v) in D_{ASML}, adding only the minimum weight arc $\min A_{(u,v)}$ to A_1. In case there are multiple arcs of minimum weight, we may keep any one.

Let $D_1 = (\hat{V}, A_1)$ be the graph that is obtained as above. Note that D_1 may still contain self-loops: arcs that connect a vertex to itself. Since the arc weights are positive, an optimal solution will never contain a self-loop. Therefore, we may remove all self-loops from the graph. We call the graph that is obtained in this way $D_2 = (\hat{V}, A_2)$.

The DST problem concerns finding the cheapest way to connect a single vertex to a set of terminals. For the ASML problem, however, we need to connect a set of start vertices, or a subset thereof, to a set of terminals. To neutralise this difference, we add a vertex and some arcs to the graph. Let $D_2 = (\hat{V}, A_2)$ be the graph that is obtained as above. We add a vertex r to $\hat{V}$, and define $V := \hat{V} \cup \{r\}$. Then, we add arcs of weight 0 from r to each start vertex $s \in S$. We define $A := \hat{A} \cup \{(r, s) | s \in S\}$, and let $w((r, s)) = 0$ for each $s \in S$. We obtain a graph $D = (V, A)$. Let $T = (\widetilde{V}, \widetilde{A})$ be an optimal solution to the DST problem for D and the set of terminals X. Then, T is a minimum weight r-arborescence that is a subgraph of D, and $\widetilde{V}$ contains all terminals X. This means that D contains a path from r to each of the terminals $x \in X$. We know that the set of arcs leaving r is $\delta^+(r) = \{(r, s) | s \in S\}$. Therefore, $\widetilde{A}$ contains at least one of the arcs in $\{(r, s) | s \in S\}$, and $\widetilde{V}$ contains at least one the start vertices $s \in S$. We now define $T' = (V', A')$, with $V' := \widetilde{V} \setminus \{r\}$ and $A' := \widetilde{A} \setminus \{(r, s) | s \in S\}$. Note that T' contains at least one start vertex and all of the terminals. Now, since we have only remove arcs of weight 0, and T is the optimal solution to the DST problem in D for terminal set X, T' is an optimal solution to the ASML problem in D for terminal set X.

To summarise, the pre-processing starts by removing all arcs that are parallel to the minimum weight arc from a vertex to another vertex. Then, all self-loops are removed. Finally, we add a vertex r and an arc of weight 0 from r to each of the start vertices. We call the graph that we have now obtained $D = (V, A)$, and we define the integers $n := |V|$, $q := |A|$, $m := |X|$ and $p := |S|$. We have shown that an optimal solution to the DST problem for D and terminals X is, after a post-processing step, also an optimal solution to the ASML problem for D_{ASML} and terminals X. Therefore, we will solve the ASML problem by applying these pre-processing steps to the graph, solving the directed Steiner tree problem, and then applying a post-processing step to obtain a solution for the ASML problem. The large part of this thesis will therefore focus on efficiently solving the directed Steiner tree problem.

3.2.2 Post-processing

After solving an instance of the directed Steiner tree problem, we must be able to translate the solution back to a solution to the ASML problem. We do this by applying post-processing steps to the solution.

Let an instance of the ASML problem be given by a directed multi-graph $D_{ASML} = (\hat{V}, \hat{A})$ with weight function $\hat{w} : \hat{A} \to \mathbb{R}_{>0}$, a set of start vertices S and a set of terminals X. As pre-processing of the problem we removed parallel arcs and self-loops from D_{ASML}, and added a vertex r with arcs of weight 0 to each of the start vertices $s \in S$. Let $D = (V, A)$ be the graph that is obtained in this way. An optimal solution to the ASML problem will never contain parallel arcs or self-loops since all arc weights of D_{ASML} are positive. Therefore, we do not need to add these parallel arcs and self-loops to obtain an optimal solution for the ASML problem. Furthermore, since we kept the lightest weight arc for each ordered vertex pair (u, v), $u, v \in \hat{V}$, we do not need to consider the other arcs from u to v in D_{ASML} for an optimal solution.

Obviously, to obtain a solution to the ASML problem from a solution to the directed Steiner tree problem, we need to remove the root and the zero weight arcs that we added. This way we obtain a minimum weight directed graph that contains paths from (some) start vertices to each of the terminals.

3.3 Complexity of the problem

The decision variant of the Steiner tree problem is one of Karp's 21 NP-complete problems [Karp(1972)], and therefore the Steiner tree problem is well-known to be NP-hard. Also the directed Steiner tree problem is well known to be NP hard [Garey and Johnson(1979)]. Since the ASML problem is so closely related to the directed Steiner tree problem, we expect the ASML problem to also be NP-hard. In this section, we will prove that this is indeed the case with a reduction from the set cover problem, which is one of Karp's 21 NP-complete problems [Karp(1972)].

Definition 10. *Let a finite set of m elements $U = \{u_1, ..., u_m\}$, called the universe, be given. Furthermore, let a collection $S = \{S_1, ..., S_p\}$ of p subsets of U, such that the union of the subsets equals the set U, be given. The set cover problem is to find a smallest sub-collection of S such the the union of its subsets is U. To be more precise, the objective is to find a set $C \subseteq S$ such that $\cup_{S_i \in C} S_i = U$ and C attains the minimum in* $\min\{|C'|;\ C' \subseteq S,\ \cup_{S_i \in C'} S_i = U\}$.

Now, we prove that the decision variant of the ASML problem is NP-complete.

Theorem 1. *Let an instance of the decision variant of the ASML problem be given by a directed multigraph $D = (V, A)$, a weight function $w : A \to \mathbb{R}_{>0}$, a set of start vertices $S \subseteq V$, a set of terminals $X \subseteq V$ and a number $t \in \mathbb{R}$. The decision variant of the ASML problem is NP-complete.*

Proof. First, we need to prove that the decision variant of the ASML problem is in NP. Suppose we are given a directed multigraph $D = (V, A)$ with weight function $w : A \to \mathbb{R}_{>0}$, a set of start vertices S and a set of terminals X. Furthermore, we have a directed graph $T = (V', A')$ and a number $t \in \mathbb{R}$. To check if (T, t) is a yes-instance to the decision variant of the ASML problem, we must check (i) if T is a subgraph of D, (ii) if for every terminal $x \in X$ there is a start vertex $s \in S$ such that T contains a path from s to x, and (iii) if $w(T) \leq t$. To check (i), we must check if $V' \subseteq V$ and $A' \subseteq A$, which takes no more than $O(n^2 + q^2)$ time. To verify (ii), we can add a vertex r and add an arc from r to each start vertex $s \in S$. This can be done in $O(p)$ time. Then, with breadth first search, we can check if there exists a path from r to x in T for each $x \in X$, which can be done in polynomial time. Finally, to verify (iii), we need to compute the weight of T. This can be done by adding the weights of all arcs in A', which takes $O(q)$ time. Checking whether $w(T) < t$ can be done in constant time. We conclude that (i), (ii) and (iii) can be verified in polynomial time, and therefore the ASML problem is in NP.

We will prove that the ASML problem is NP-hard with a reduction from the set cover problem. Let a set $U = \{u_1, ..., u_m\}$ and a collection $S = \{S_1, ..., S_p\}$ of subsets of U be given. Suppose that C is a solution to the set cover problem with $|C| = t'$. That is, let $C \subseteq S$ be a collection of sets such that $\cup_{S_i \in C} S_i = U$ and C attains the minimum in $\min\{|C'|;\ C' \subseteq S,\ \cup_{S_i \in C'} S_i = U\}$, and $|C| = t' \leq t$. We now construct a directed graph $D = (V, A)$, where we have a vertex v_i for each subset $S_i \in S$ and a vertex y_j for each element $u_j \in U$. Furthermore, we add an arc of weight 0 from each vertex v_i to a vertex y_j if the corresponding subset S_i contains the element u_j that corresponds to y_j. Finally, we add a vertex s, and add arcs of weight 1 from s to each vertex v_i that corresponds to a subset. This is demonstrated on an example in Figure 2. Then, we define the set of start vertices to be exactly vertex s, and the terminals to be the vertices y_j. Now, using the solution C of the set cover problem, we will construct a solution to the ASML problem, and we will show that this solution also has value t'.

We construct a subgraph $T = (V', A')$ of D. We let $V' = \{v_i | v_i$ corresponds to a subset $S_i \in C\} \cup \cup_{j=1,..,m} y_j \cup \{s\}$. Furthermore, we let A' be the union of the incoming and outgoing arcs of the vertices v_i that correspond to subsets $S_i \in C$. That is, $A' = \{a \in A | a \in (\delta^+(v_i) \cup \delta^-(v_i))$ for some v_i corresponding to $S_i \in C\}$. Since C is a solution to the set cover problem, the union of the subsets of C equals the set U. Subgraph T contains arcs from s to each v_i corresponding to a subset $S_i \in C$, and from v_i to the vertices corresponding to the elements S_i consists of. We can therefore conclude that for every terminal, there is a path from start vertex s to the terminal. Since only the arcs from s to vertices v_i are weighted, and they each have weight 1, T is a graph of weight t'. Therefore, T is a yes-instance for the decision variant ASML problem for graph $D = (V, A)$, weight function $w(a) = 1$ if $a \in \delta^+(s)$ and $w(a) = 0$ otherwise, start vertices $S = \{s\}$, terminals $X = \{y_j | j = 1, ..., m\}$ and $t \in \mathbb{R}$.

Now, let $T = (V', A')$ be a yes-instance of value t' to the ASML problem in graph $D = (V, A)$. Let C be the set containing the subsets corresponding to the vertices v_i that are contained in T. The union of the subsets in C will contain all elements of U, since T must contain a path to each terminal. Furthermore, C is a set of size t', since there are t' arcs from s to the vertices v_i corresponding to subsets S_i. Therefore, given a solution to the ASML problem of value $t' \leq t$, we also have a solution to the set cover problem of value t'.

Finally, note that the construction of the instance of the ASML problem, when given an instance of the set cover problem, takes polynomial time. Constructing the graph $D = (V, A)$ and the start vertices and terminals can be done in $O(p + m + pm)$ time. Thus, the set cover problem can be reduces to the ASML problem in polynomial time. Since the set cover problem is known to be NP-complete, and we have shown that the ASML problem is in NP, the ASML problem is also NP-complete. □

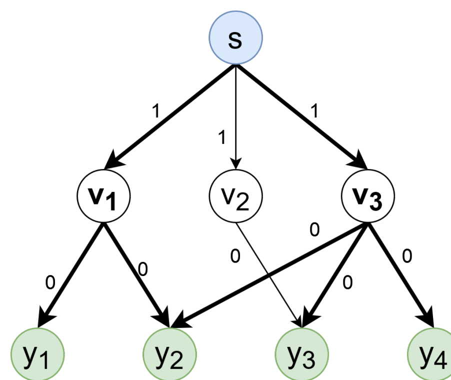


Figure 2: Example of an instance of set cover represented as an instance of the ASML problem. Here, the set cover instance is given by a universe $U = \{u_1, u_2, u_3, u_4\}$ and the collection of subsets $S = \{\{u_1, u_2\}, \{u_3\}, \{u_2, u_3, u_4\}\}$. The elements of the universe are represented by the vertices y_1, y_2, y_3 and y_4. The subsets are represented by the vertices v_1, v_2 and v_3. The optimal solution to the set cover problem is $C = \{\{u_1, u_2\}, \{u_2, u_3, u_4\}\}$. The corresponding solution for the ASML problem is shown in bold.

Since the decision variant of the ASML problem is NP-complete, we can conclude that the ASML problem is NP-hard.

4 Approximation algorithms

Since the directed Steiner tree problem is NP-complete, there are no polynomial time exact algorithms for the problem unless P=NP. This means that, for a large instance, exact algorithms can take an extremely long time to complete. It can therefore be interesting to look at algorithms that do not guarantee an optimal solution: *heuristics*. By using a heuristic to find an approximate solution, we are often able to find a solution much faster than by using an exact algorithm. However, this solution might be far from optimal. Nonetheless, these algorithms can give an upper bound for the optimal solution. Furthermore, the construction and analysis of the algorithms can also give us insight in the complexity of the problem.

Sometimes, we can give a guarantee of how good a solution is compared to the optimal solution. In this case, the algorithm is called an *approximation algorithm*. An approximation algorithm is an algorithm that finds, within polynomial time, an approximate solution for the problem with a guarantee on how good the solution is. For a minimisation problem this guarantee is that, for any instance I, the solution returned by the algorithm is at most α times as big as the optimal solution.

Definition 11. *Let a minimisation problem X be given. For an instance I of the problem, let the value of the optimal solution be $OPT(I)$. Then, an α-**approximation algorithm** is an approximation algorithm for which, for any instance I, we have*

$$APPR(I) \leq \alpha \cdot OPT(I).$$

*Here, $\alpha > 1$, and $APPR(I)$ is the value of the solution returned by the approximation algorithm. We call α the **approximation factor**.*

Since the inequality $APPR(I) \leq \alpha \cdot OPT(I)$ holds for any instance I, the approximation factor is a guarantee for how bad an algorithm may perform in the worst case. Note that it is only an upper bound; it does not imply that there exists an instance for which the bound is attained. Furthermore, the approximation factor does not offer insight into how an algorithm performs on average.

4.1 Shortest paths algorithm

The first idea that comes to mind might be computing the shortest path from the root r to each terminal, and combining these paths into an r-arborescence. This gives rise to the shortestPathsToTerminals Algorithm, as described below.

Algorithm shortestPathsToTerminals: Returns an r-arborescence that is the union of the shortest paths from the root to each terminal.

Data: Directed graph $D = (V, A)$, weight function $w : A \to \mathbb{R}_{\geq 0}$, root $r \in V$, set of terminals $X \subseteq V$

Result: r-arborescence $T = (V', A')$ that is a subgraph of D, with weight function $w' : A' \to \mathbb{R}^+$, such that $X \subseteq V'$

```
1 graph T ← (∅, ∅);
2 graph Y ← shortest-path-r-arborescence of D;
3 for terminal x in X do
4 |   graph SP ← path from r to x in Y;
5 |   Add SP to T;
```

The algorithm initiates an empty graph T for the solution. Then, it finds the shortest-path-r-arborescence, which is an r-arborescence containing, for each vertex that can be reached from r, the shortest path from r to this vertex. Then, for each terminal x, the algorithm extracts the shortest path from r to x from the shortest-path-r-arborescence. This path is then added to the solution graph T.

To compute the shortest-path-r-arborescence, the algorithm uses Dijkstra's algorithm. Dijkstra's algorithm for finding the shortest-path-r-arborescence can be performed in $O(q + n\log(n))$ time with

an implementation by Fredman and Tarjan(1987). The r-arborescence is saved in a *singleSourcePaths* structure, from which we can easily extract paths from the source to any vertex that can be reached from the source. The path is then added to the solution, Note that in the implementation, the vertex and arc sets are sets that cannot contain duplicate elements. Therefore, if a vertex or arc that is already in T is added again, it will not appear twice. Adding the shortest path to graph T will therefore take at most $O(q + n)$ time, and we can conclude that the entire for-loop takes $O(m(q + n))$ time. Therefore, the shortestPathsToTerminals Algorithm takes $O(m(q+n)+q+n\log(n)) = O(n\log(n)+nm+qm)$ time.

The algorithm is rather fast. However, since it is an approximation algorithm, it will not always return an optimal solution. To analyse the quality of the solutions returned by the algorithm, we look at its approximation factor.

Theorem 2. *The shortestPathsToTerminals Algorithm is an m-approximation algorithm for the directed Steiner tree problem.*

Proof. Let a directed graph $D = (V, A)$ with weight function $w : A \to \mathbb{R}_{\geq 0}$ defined on the arcs be given. Furthermore, let a root vertex $r \in V$ and a set of terminals $X \subseteq V$ be given. Define $\overline{y}$ to be the solution returned by the shortestPathsToTerminals Algorithm, and y^* the optimal solution to the DST problem. Here, y is a decision variable with $y_a = 1$ if arc a is contained in a solution, and $y_a = 0$ otherwise.

Let the shortest path from the root to a terminal x_i in graph D be defined as $\overline{P}_{x_i}$, for each $i = 1, ..., m$. Furthermore, define $P^*_{x_i}$ to be the path from the root to terminal x_i in the optimal solution, for each $i = 1, ..., m$. Note that the objective values of the shortestPathsToTerminals solution and the optimal solution are $\sum_{a\in A} w(a)\overline{y}_a$ and $\sum_{a\in A} w(a)y^*_a$, respectively. We now have the following inequalities.

$$\sum_{a\in A} w(a)\overline{y}_a \leq \sum_{i=1}^{m} \sum_{a\in \overline{P}_{x_i}} w(a)\overline{y}_a \tag{1a}$$

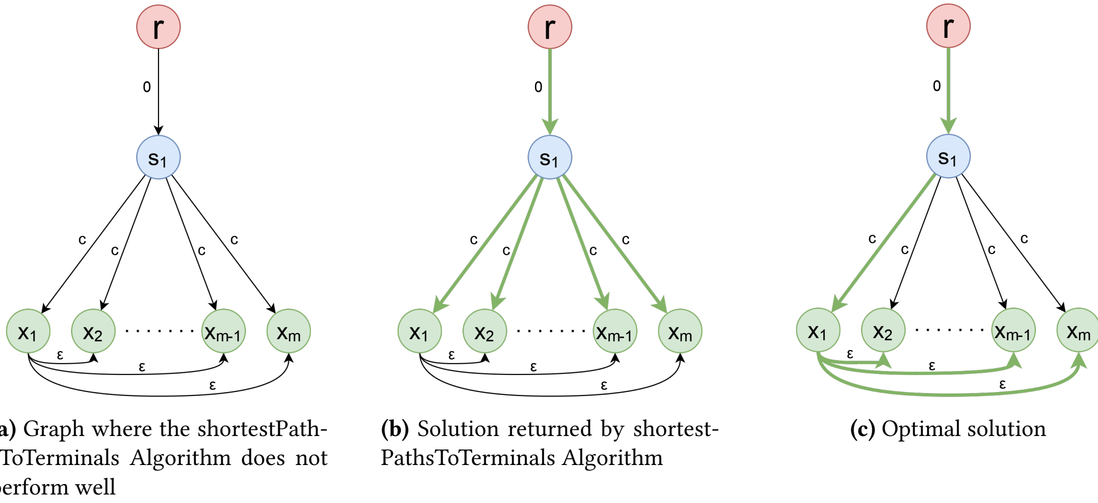
$$\leq \sum_{i=1}^{m} \sum_{a\in P^*_{x_i}} w(a)y^*_a \tag{1b}$$

$$\leq m \max_{i=1,\ldots,m} \left(\sum_{a\in P^*_{x_i}} w(a)y^*_a \right) \tag{1c}$$

$$\leq m \sum_{a\in A} w(a)y^*_a \tag{1d}$$

The first inequality states that the value of the shortestPathsToTerminals solution is at most the sum of the weights shortest paths to each terminal. Of course, it is possible for the paths to have overlapping arcs, which would make the value of the shortestPathsToTerminals solution smaller than the sum of the weights of the shortest paths. Note that each path from the root to a terminal in the optimal solution has a weight greater or equal to the weight of the shortest path from the root to this terminal. This implies inequality 1b. It is easy to see that the sum of the lengths of the paths to the terminals in the optimal solution has weight at most m times the longest path to a terminal in the optimal solution, implying inequality 1c. Finally, we can bound this by m times the value of the optimal solution, since the value of the optimal solution is larger or equal than the length of the longest path to a terminal in the optimal solution. We can conclude that the solution of the shortestPathsToTerminals Algorithm is at most m times worse than the optimal solution. That means that the algorithm is an m-approximation algorithm. □

We have seen that the shortestPathsToTerminals Algorithm will always return a solution that is at most m times worse than the optimal solution. This does not necessarily mean that the upper bound of m times the optimal solution is actually achieved. Therefore, we look at an example where the shortest-PathsToTerminals Algorithm performs poorly.



(a) Graph where the shortestPathsToTerminals Algorithm does not perform well

(b) Solution returned by shortestPathsToTerminals Algorithm

(c) Optimal solution

Figure 3: Graph with one start vertex s_1 and m terminals $x_1, ..., x_m$

Figure 3 shows a directed graph containing the root, a start vertex s_1 and m terminals $x_1, ..., x_m$. There are arcs of weight $c > 0$ from s_1 to each of the terminals, and arcs of weight $\epsilon > 0$ from x_1 to each of the other terminals. As shown in Figure 3b, the shortestPathsToTerminals Algorithm will output the r-arborescence consisting of the arcs $\{(r, x_1), (r, x_2), ..., (r, x_{m-1}), (r, x_m)\})$. This is a graph of weight cm. However, as shown in Figure 3c, an optimal solution for this graph is an r-arborescence consisting of the arcs $\{(r, x_1), (x_1, x_2), ..., (x_{m-1}, x_m)\}$. This is a graph of weight $c + (m-1)\epsilon$. If we let $\epsilon \to 0$, the algorithm returns a solution that is almost m times worse than the optimal solution. This shows that the upper bound in Theorem 2 is tight.

4.2 Shortest bunches algorithm

4.2.1 One bunch

As seen in the previous section, simply computing the shortest paths tree and restricting it to the paths from the root to the terminals, does not always give a good solution. It can often be more beneficial to choose paths that overlap, so that a section of the path to a terminal is also used in the path to another terminal. One way to implement this idea into an algorithm is by computing a bunch, an idea inspired by Charikar et al.(1999). A bunch is an r-arborescence that consists of a path from the root to an intermediate vertex v, and a set of shortest paths from v to each of the terminals. This gives rise to an algorithm that computes a shortest bunch, as described below.

Algorithm shortestBunch: Returns minimum weight bunch from the root to the terminals. A bunch is a combination of a shortest path from the root to an intermediate vertex, and the shortest paths from this intermediate vertex to each of the terminals.

```
Data: Directed graph D = (V, A), weight function w : A → ℝ≥0, root r ∈ V, set of terminals
      X ⊆ V
Result: r-arborescence T = (V', A') that is a subgraph of D, with weight function
        w' : A' → ℝ≥0, such that X ⊆ V'
1  graph T ← (∅, ∅);
2  real number c ← ∞;
3  graph Y ← shortest-path-r-arborescence in D;
4  for vertex v in V do
5      boolean validIntermediate ← true;
6      graph SPv ← shortest path from r to v in Y;
7      if there is a path from r to v then
8          graph P ← SPv;
9          graph Z ← shortest-path-v-arborescence in D;
10         for terminal x in X do
11             if validIntermediate == true then
12                 graph SPx ← shortest path from v to x in Z;
13                 if there is no path from v to x then
14                     validIntermediate ← false;
15                     break;
16                 Add SPx to P;
17         if validIntermediate == true then
18             if weight(P) < c then
19                 T ← P;
```

For each vertex $v \in V$, the algorithm tries to create a bunch: an r-arborescence that consists of a path from r to v, and the set of shortest paths from v to each of the terminals. This bunch is saved as a temporary r-arborescence if there exists a path from r to v and from v to each of the terminals, and if the total weight of the r-arborescence is smaller than the weight of the current temporary r-arborescence. After the algorithm has tried all choices of v, it returns the best r-arborescence. Note that in the case of $v = r$, the result is the same as the results of the shortestPathsToTerminals Algorithm. Since a terminal does not need to be a leaf, it is also possible for v to be a terminal.

Figure 4 shows the same example as was used in Section 4.1, to show a worst case example for the shortestPathsToTerminals Algorithm. In this situation, the shortestBunch Algorithm returns the optimal solution, and therefore performs much better than the shortestPathsToTerminals Algorithm.

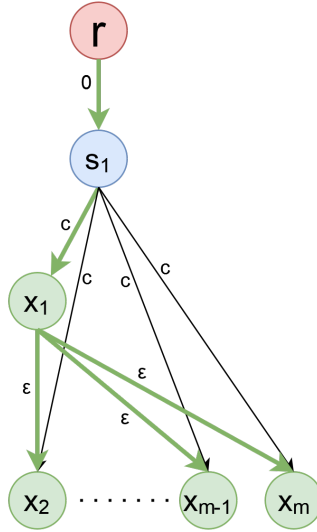


Figure 4: Graph where the shortestBunch Algorithm performs well

However, the shortestBunch Algorithm is slower. The algorithm performs Dijkstra's shortest-path-r-arborescence algorithm to find a shortest path from the root to each vertex $v \in V$. As discussed in Section 4.1, Dijkstra's algorithm for computing the shortest-path-r-arborescence has complexity $O(q+n\log(n))$. This means that lines 1-3 take $O(q+n\log(n))$ time. Then, for each $v \in V$, the algorithm gets the shortest path from the previously computed shortest-path-r-arborescence, which, as explained in the previous section, can be done in constant time. Line 8 will take at most $O(q+n)$ time. In line 9, Dijkstra's algorithm is performed for vertex v, again taking $O(q+n\log(n))$ time. The computed shortest paths are added to the temporary graph, again taking at most $O(q+n)$ time. This gives lines 5-16 a time complexity of $O(q+n+q+n\log(n)+m(q+n))$. Finally, setting the solution to the temporary graph in line 19 takes $O(q+n)$ time. This gives lines 4-19 a complexity of $O(n(q+n+q+n\log(n)+m(q+n)+q+n))$, and the entire algorithm a complexity of $O(n(n+1)\log(n)+q(3n+1)+2n^2+mn(q+n))$ $= O(n^2\log(n)+n^2m+nqm)$.

Since the shortestBunch Algorithm chooses the optimal intermediate vertex, it performs at least as good as the shortestPathsToTerminals Algorithm. Therefore, if the root is the best option for the intermediate vertex, and is thus chosen as the intermediate vertex, the result is exactly the same as the result of the shortestPathsToTerminals Algorithm. We can therefore conclude that the shortestBunch Algorithm is also an m-approximation algorithm for the directed Steiner tree problem. This is formally proven below.

Theorem 3. *The shortestBunch Algorithm is an m-approximation algorithm for the directed Steiner tree problem.*

Proof. Let a directed graph $D=(V,A)$ with weight function $w: A \to \mathbb{R}_{\geq 0}$ defined on the arcs be given. Furthermore, let a root vertex $r \in V$ and a set of terminals $X \subseteq V$ be given. Define $\hat{y}$ to be the solution returned by the shortestBunch Algorithm, y^* the optimal solution to the directed Steiner tree problem, and $\overline{y}$ the shortest-path-r-arborescence. Here, y is a decision variable with $y_a = 1$ if arc a is contained in a solution, and $y_a = 0$ otherwise.

Define $P^*_{v,w}$ to be the path from a vertex $v \in V$ to a vertex $w \in V$ in the optimal solution. Furthermore, let the shortest path from any $v \in V$ to any $w \in V$ in graph D be defined as $\overline{P}_{v,w}$. Note that the objective values of the shortestBunch solution and the optimal solution are $\sum_{a\in A} w(a)\hat{y_a}$ and $\sum_{a\in A} w(a)y^*_a$, respectively. We now have the following equations and inequalities.

$$\sum_{a\in A} w(a)\hat{y}_a = \min_{v\in V}\left(\sum_{a\in\overline{P}_{r,v}} w(a) + \sum_{i=1}^{m}\sum_{a\in\overline{P}_{v,x_i}} w(a)\right) \tag{2a}$$

$$\leq \sum_{a\in\overline{P}_{r,r}} w(a) + \sum_{i=1}^{m}\sum_{a\in\overline{P}_{r,x_i}} w(a) \tag{2b}$$

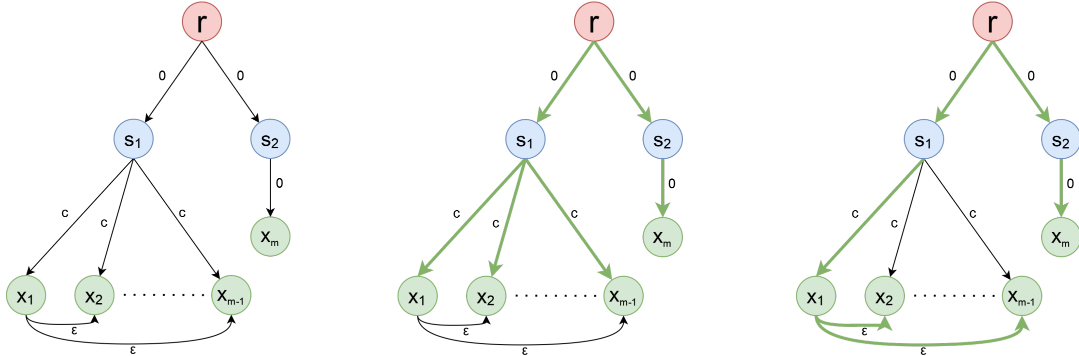
$$= \sum_{i=1}^{m}\sum_{a\in\overline{P}_{r,x_i}} w(a)\overline{y}_a \tag{2c}$$

$$\leq \sum_{i=1}^{m}\sum_{a\in P^*_{r,x_i}} w(a)y^*_a \tag{2d}$$

$$\leq m \max_{i=1,\ldots,m}\left(\sum_{a\in P^*_{x_i}} w(a)y^*_a\right) \tag{2e}$$

$$\leq m\sum_{a\in A} w(a)y^*_a \tag{2f}$$

The first equation holds because of how the shortestBunch Algorithm works: it computes the bunch rooted at r, containing all terminals $x \in X$, for an intermediate vertex $v \in V$ that gives the minimum weight bunch. This solution will always have a value smaller than or equal to the weight of a bunch where a specific vertex, such as the root, is chosen as intermediate vertex. This implies inequality 2b. Note that a bunch rooted at r is exactly the set of shortest paths from the root to each terminal, implying 2c. The rest of the inequalities follow by exactly the same reasoning as in the proof of Theorem 2. We conclude that the shortestBunch Algorithm is an m-approximation algorithm for the directed Steiner tree problem. □



(a) Graph where the shortestBunch Algorithm does not perform well

(b) Solution returned by shortestBunch Algorithm

(c) Optimal solution

Figure 5: Graph with two start vertices s_1 and s_2, and m terminals $x_1, \ldots, x_m$

Figure 5 shows a directed graph containing the root, two start vertices s_1 and s_2, and m terminals $x_1, \ldots, x_m$. There are arcs of weight $c > 0$ from s_1 to terminals $x_1, .., x_{m-1}$, and arcs of weight $\epsilon > 0$ from x_1 to terminals $x_2, \ldots, x_{m-1}$. Figure 5b shows that the shortestBunch Algorithm will have to choose the root as intermediate vertex, since that is the only way for all terminals to be contained in the bunch. Therefore, the result is the set of shortest paths from the root to the terminals. The shortestBunch Algorithm returns an r-arborescence of weight $(m-1)c$, whereas the optimal solution, shown in Figure 5c, has a value of $c + (m-2)\epsilon$. If we let $\epsilon \to 0$, the algorithm returns a solution that is almost $m-1$

times worse than the optimal solution. This means that the lower bound for the approximation factor is $m - 1$, and the upper bound cannot be much tighter than we have shown above.

Figure 5c shows the optimal solution for this example. This solution can be reached by computing not one, but multiple bunches. In the case of this example, we would have a bunch rooted at s_1 with intermediate vertex x_1, and a bunch rooted at s_2 with intermediate vertex s_2.

4.2.2 Multiple bunches

Algorithm shortestBunch can be changed such that it computes multiple bunches. However, there are many possibilities of implementing such an idea. For example, we can keep r the root vertex, compute multiple intermediate vertices, and then compute shortest paths from each intermediate vertex to several terminals. This leaves questions such as: how do we choose the intermediate vertices? And how many intermediate vertices do we want? And how do we choose which terminals to connect to which intermediate vertices?

Another method would be to use the input data of the ASML problem. Instead of using r as root vertex, we can use the start vertices as roots for our bunches. For each terminal we compute which start vertex is closest to it, and the terminal will be part of the bunch rooted at that start vertex. We then compute p bunches using the shortestBunch Algorithm. The algorithm is described below.

Algorithm multipleBunches: Returns p minimum weight bunches, each rooted at a start vertex.

Data: Directed graph $D = (V, A)$, weight function $w : A \to \mathbb{R}_{\geq 0}$, set of start vertices $S \subseteq S$, set of terminals $X \subseteq V$

Result: Directed graph $T = (V', A')$ that is a subgraph of D, consisting of p s-arborescences each rooted at a vertex $s \in S$, with weight function $w' : A' \to \mathbb{R}_{\geq 0}$, such that $X \subseteq V'$.

1 graph $T \leftarrow (\emptyset, \emptyset)$;
2 **for** *vertex* v *in* V **do**
3   graph $SP_v \leftarrow$ shortest-path-v-arborescence in D;
4 **for** *terminal* x *in* X **do**
5   real number $bestWeight \leftarrow \infty$;
6   **for** *start vertex* s *in* S **do**
7   graph $SP \leftarrow$ path from s to x in SP_s;
8   **if** *there is a path from* s *to* x *in* D *and weight(*SP*)* $< bestWeight$ **then**
9             $bestWeight \leftarrow$ weight(SP) ;
10   vertex $bestVertex \leftarrow s$;
11   Add x to $X_{bestVertex}$;
12 **for** *start vertex* s *in* S **do**
13     $P \leftarrow$ graph returned the shortestBunch Algorithm for graph D with weight function w, root s, set of terminals X_s;
14   Add P to T;

Algorithm multipleBunches first computes the shortest-path-r-arborescence rooted at v for each vertex $v \in V$. It then chooses, for each terminal, the start vertex that is closest to the terminal. Then, it performs the shortestBunch Algorithm for each start vertex. For each start vertex, the bunch that is returned by the algorithm is added to graph T. T is the graph that is returned by the algorithm. Note that, in contrary to the other algorithms, this algorithm does not use the root r. The graph that is returned is a set of p s-arborescences each rooted at a start vertex s, and its underlying undirected graph does not need to be connected.

As mentioned before, Dijkstra's algorithm for finding the shortest paths forest takes $O(q + n\log(n))$ time. This is performed for each vertex, which means lines 2-3 take $O(n(q + n\log(n)))$ time. Then, for

each terminal x, and each start vertex s, we check whether s is the closest start vertex to x. The steps in lines 7-11 take constant time, giving lines 4-11 complexity $O(mp)$. Finally, for each start vertex s, we perform the shortestBunch Algorithm. Note that that shortest-path-v-arborescences have already been computed for each v, giving line 13 a complexity of $O(2qn + 2n^2 + mn(q + n))$. Adding the bunch to graph T has a complexity of $O(q + n)$. Therefore, the complexity of the multipleBunches Algorithm is $O(n(q + n\log(n)) + mp + 2qn + 2n^2 + mn(q + n) + q + n) = O(n^2\log(n) + n^2m + nqm + mp)$.

Figure 6 shows the same example as in Section 4.2.1. The multipleBunches Algorithm returns two bunches; one rooted at s_1 with intermediate vertex x_1, and one rooted at s_2 with intermediate vertex s_2. This gives a better solution that the shortestBunch Algorithm, which chose the root as intermediate vertex. In fact, in this example, it returns the optimal solution.

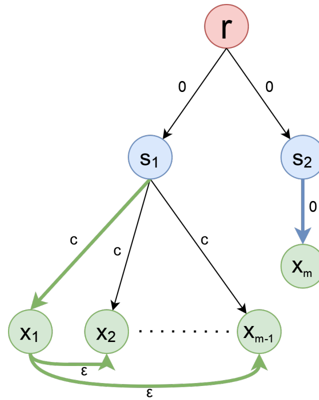
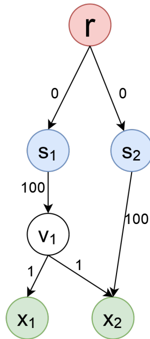
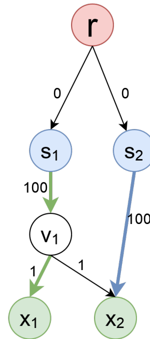


Figure 6: Graph where the multipleBunches Algorithm performs better than the shortestBunch Algorithm

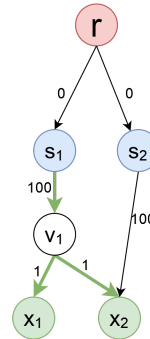
There are, however, some disadvantages to this method. First of all, choosing which terminal will be connected to which start vertex by only looking at the shortest distance does not always give the best solution. An example of where this goes wrong is shown in Figure 7. Here, the multipleBunches Algorithm first decides which start vertex x_1 and x_2 should be connected to. Terminal x_1 is closest to s_1 and terminal x_2 is closest to s_2. This means that there will be two bunches, as shown in Figure 7b. This gives a solution of value 201. The shortestBunch Algorithm chooses vertex v_1 as the intermediate vertex, and returns a bunch that consists of the path (r, s_1, v_1) and the arcs (v_1, x_1) and (v_1, x_2). This results in a solution of value 102, which is the optimal solution. This example shows that the multipleBunches Algorithm does not always perform better than the shortestBunch Algorithm.



(a) Graph where the multiple-Bunches Algorithm does not perform well



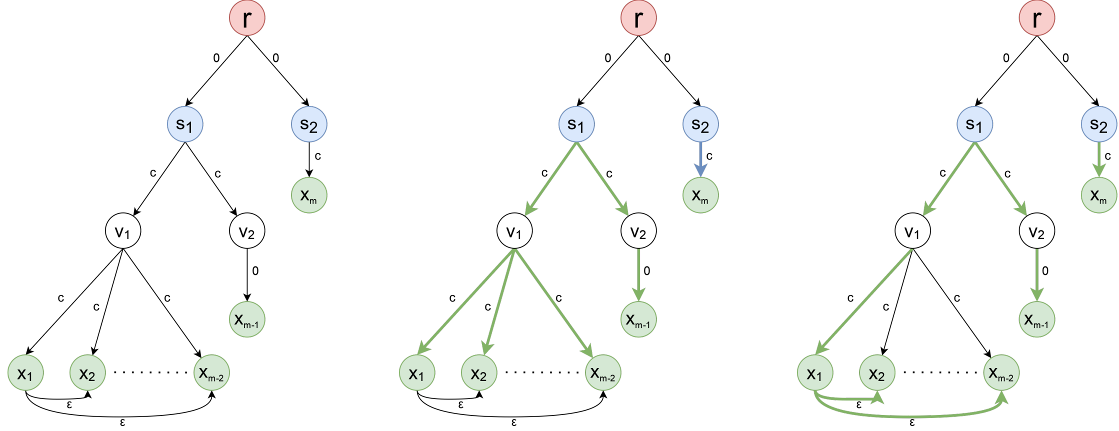
(b) Solution returned by multiple-Bunches Algorithm



(c) Optimal solution

Figure 7: Graph with two start vertices s_1 and s_2, 2 terminals x_1 and x_2, and one normal vertex v_1

Furthermore, choosing the bunches to be rooted at the start vertices might not always give the best result. In the situation shown in Figure 6, choosing bunches rooted at the start vertices solves the problem that occurred when using the shortestBunch Algorithm. However, if we slightly alter this situation to the graph shown in Figure 8, we see that the multipleBunches performs no better than the shortestBunch Algorithm on this example. As shown in Figure 8c, the optimal solution consists of three bunches: one rooted at s_1 with intermediate vertex x_1; one rooted at s_1 with intermediate vertex v_2; and one rooted at s_2 with intermediate vertex s_2.



(a) Graph where the multipleBunches Algorithm does not perform well

(b) Solution returned by multipleBunches Algorithm

(c) Optimal solution

Figure 8: Graph with two start vertices s_1 and s_2, m terminals $x_1, ..., x_m$, and two normal vertices v_1 and v_2

Despite these disadvantages, the multipleBunches Algorithm will never perform worse than the shortestPathsToTerminals Algorithm. This is because the algorithm connects each terminal to the start vertex it is closest to. Therefore, in the worst case, the algorithm returns the collection of shortest paths from the start vertices to the terminals. This is the same as the solution returned by the shortestPathsToTerminals Algorithm, minus the root. Therefore, the multipleBunches Algorithm will never return a solution worse than m times the optimal solution. This is formally proven below.

Theorem 4. *The multipleBunches Algorithm is an m-approximation algorithm for the ASML problem.*

Proof. Let a directed graph $D' = (V', A')$ with weight function $w' : A' \to \mathbb{R}_{>0}$ defined on the arcs be given. Furthermore, let a set of start vertices $S \subseteq V'$ and a set of terminals $X \subseteq V'$ be given. We let the graph $D = (V, A)$ be defined as D', with an extra vertex r, such that $V = V' \cup \{r\}$, and arcs of weight 0 from r to each start vertex $s \in S$. The weight function w is defined as $w : A\times \to \mathbb{R}_{\geq 0}$, and is equal to w' for $a \in A'$, and zero otherwise. Define $\hat{y}$ to be the solution returned by the multipleBunches Algorithm, y^* the optimal solution to the ASML problem, and $\overline{y}$ the shortest-path-r-arborescence. Here, y is a decision variable with $y_a = 1$ if arc a is contained in a solution, and $y_a = 0$ otherwise.

Let $\hat{P}_{v,x_i}$ be the path from v to w in the solution returned by the multipleBunches Algorithm, defined for each $v, w \in V$. Furthermore, define $P^*_{v,w}$ to be the path from a vertex $v \in V$ to a vertex $w \in V$ in the optimal solution. Let the shortest path from any $v \in V$ to any $w \in V$ in graph D be defined as $\overline{P}_{v,w}$. Moreover, let the bunch rooted at start vertex s be defined as B_s, for $s \in S$. Finally, define for each start vertex $s \in S$ the set of terminals s is closest to as X_s. Note that the objective values of the multipleBunches solution and the optimal solution are $\sum_{a \in A} w(a)\hat{y_a}$ and $\sum_{a \in A} w(a)y^*_a$, respectively. We now have the following equations and inequalities.

$$\begin{aligned}
\sum_{a\in A} w(a)\hat{y}_a &\le \sum_{s\in S}\sum_{a\in B_s} w(a)\hat{y}_a && \text{(3a)}\\
&= \sum_{s\in S}\min_{v\in V}\left(\sum_{a\in \overline{P}_{s,v}} w(a) + \sum_{x\in X_s}\sum_{a\in \overline{P}_{v,x}} w(a)\right) && \text{(3b)}\\
&\le \sum_{s\in S}\left(\sum_{a\in \overline{P}_{s,s}} w(a) + \sum_{x\in X_s}\sum_{a\in \overline{P}_{s,x}} w(a)\right) && \text{(3c)}\\
&= \sum_{s\in S}\sum_{x\in X_s}\sum_{a\in \overline{P}_{s,x}} w(a) && \text{(3d)}\\
&= \sum_{x\in X}\sum_{a\in \overline{P}_{r,x}} w(a)\overline{y}_a && \text{(3e)}\\
&\le \sum_{x\in X}\sum_{a\in P^*_{r,x}} w(a)y^*_a && \text{(3f)}\\
&\le m \max_{i=1,\dots,m}\left(\sum_{a\in P^*_{x_i}} w(a)y^*_a\right) && \text{(3g)}\\
&\le m\sum_{a\in A} w(a)y^*_a && \text{(3h)}
\end{aligned}$$

The different bunches may overlap. Therefore, the sum of the weights of the different bunches will be greater than or equal to the value of the solution returned by the multipleBunches Algorithm, implying inequality (3a). The multipleBunches Algorithm computes the shortest bunch rooted at a start vertex s containing terminals $x \in X_s$ for each start vertex $s \in S$. Equation (3b) holds because of how the shortestBunch Algorithm works. Inequality (3c) follows since the minimum is taken over all $v \in V$, and $s \in V$ is therefore one of the possible choices for an intermediate vertex. Equation (3d) follows immediately. The sets X_s, for $s \in S$, were constructed such that they contain the terminals $x \in X$ for which $\overline{P}_{s,x} = \min_{c\in S} \overline{P}_{c,x}$. Therefore, for each $s \in S$ and $x \in X$, the path $(r, x) \cup \overline{P}_{s,x}$ is a shortest path from r to x. Now, Equation (3e) follows.

The rest of the inequalities are the same as in the proof of Theorems 2 and 3. We conclude that the shortestBunch Algorithm is an m-approximation algorithm for the ASML problem. □

Note that for the ASML problem, a set of start vertices is given as input, and there are arcs of weight 0 from the root to, and only to, the start vertices. The multipleBunches Algorithm can also be applied for directed Steiner tree problems in general, if one supplies a set of start vertices. However, in this case, the above proof no longer holds op, since the distance from the root to the start vertices will generally not be 0, and there might be arcs from the root to other vertices too.

The performance of the algorithm could be improved by, for example, choosing which terminal should be connected to which start vertex in a different way. This could be done in the same way as the intermediate vertex for a bunch is chosen, by trying all combinations of terminals and start vertices. Furthermore, one could look at choosing the roots of the bunches in a different way. For example, by trying each vertex as root of a bunch. Another idea would be to compute one bunch rooted at r, and then trying each vertex that is contained in this bunch as root for a new bunch. However, these ideas would increase the time complexity. Instead of diving into the idea of bunches further, this thesis will continue to focus on different algorithms.

4.3 Greedy algorithm

Another approach to the problem is computing a minimum spanning r-arborescence and then taking a subgraph that contains all terminals. In contrary to the non-spanning variant, finding a minimum spanning r-arborescence can be done in polynomial time. This problem can be solved in $O(q + n \log n)$ with Chu-Liu/Edmonds algorithm with an implementation by Gabow et al.(1986). In undirected graphs it is easy to see when a spanning tree exists; the graph needs to be connected. However, in directed graphs, we cannot assume that all vertices are reachable from a given source vertex, even if the underlying undirected graph is connected. However, we are only interested in a subgraph that contains all terminals; and this graph does not need to span all vertices. Therefore, we will use a variation of Prim's algorithm for undirected minimum spanning trees. Prim's algorithm is an algorithm that greedily chooses arcs to add to the tree.

In an undirected graph, Prim's algorithm starts with a root vertex, which can be chosen at random. Then, in each iteration, a vertex that is in the r-arborescence is connected to a vertex that is not in the r-arborescence. This is done by choosing the minimum-weight arc that connects a vertex in the r-arborescence to a vertex not in the r-arborescence [Prim(1957)]. The algorithm cannot directly be translated to any directed graph, since it assumes that all vertices are connected. In a directed graph it is mostly not the case that each vertex is reachable from any other vertex. However, in our case, we are only interested in the r-arborescence containing all terminals. We have assumed that all terminals are reachable from at least one of the start vertices. Therefore, an alteration of the algorithm will return an r-arborescence that contains all terminals. This algorithm is called greedyrArborescence.

Algorithm greedyrArborescence: Computes an r-arborescence containing all terminals by greedily choosing arcs.

Data: Directed graph $D = (V, A)$, weight function $w_D : A \to \mathbb{R}_{\geq 0}$, root $r \in V$, set of terminals $X \subseteq V$

Result: r-arborescence $T = (V', A')$ that is a subgraph of D, with weight function $w' : A' \to \mathbb{R}_{\geq 0}$, such that $X \subseteq V'$

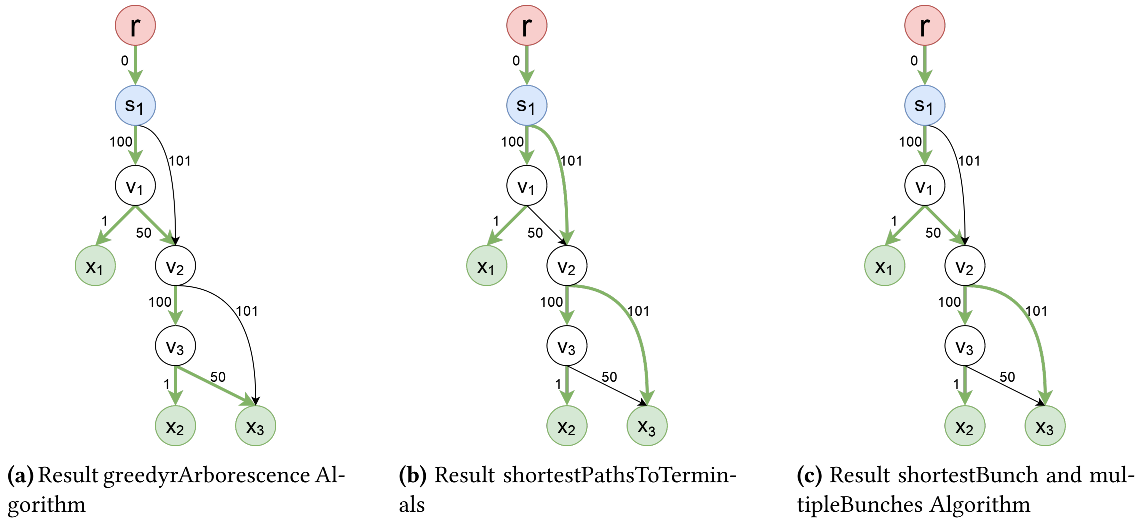
```
1  graph Y ← (∅, ∅);
2  Add vertex r to Y;
3  Mark r as visited;
4  Set minimum distance of r to 0;
5  for out-neighbours¹ v of r do
6  |   Set minimum distance of v to w_(r,v);
7  |   Set s as the parent of v;
8  while not all terminals x ∈ X are in Y yet do
9  |   v ← the vertex with the smallest minimum distance that has not been visited;
10 |   Add vertex v and arc (parent of v, v) to Y;
11 |   Set minimum distance of v to 0;
12 |   Mark v as visited;
13 |   for out-neighbours u of v do
14 |   |   if u has not been visited yet and w_(v,u) < minimum distance of u then
15 |   |   |   Set minimum distance of u to w_(v,u);
16 |   |   |   Set v as the parent of u;
17 graph T ← (∅, ∅);
18 graph P ← shortest-path-r-arborescence in Y;
19 for terminal x in X do
20 |   graph SP ← path from r to x in Y;
21 |   Add SP to T;
```

[1]Given a directed graph $D = (V, A)$, an out-neighbour of a vertex u is a vertex v such that $(u, v) \in A$.

The first 16 lines of Algorithm greedyrArborescence is similar to Prim's algorithm. In each iteration, the algorithm computes the lightest arc that goes from a vertex in the r-arborescence to a vertex outside of the r-arborescence. However, in this algorithm, this is only repeated until all terminals have been added. The r-arborescence that we get in this way, Y, is likely bigger than necessary; it contains vertices that are not on any of the paths from the root to the terminals. Therefore, the algorithm creates a new r-arborescence T that consists of all the shortest paths in Y from r to each terminal. This r-arborescence is then returned.

The greedyrArborescence Algorithm keeps track of which vertices have been visited, the current minimum distance from the r-arborescence to each vertex, and the parent of each vertex, in 3 arrays of size n. Initialising an array of size n takes $O(n)$ time, and therefore the first part of the algorithm takes $O(n)$ time. Since the root has p neighbours, lines 5-7 cost $O(p)$ time. Finding the vertex of smallest minimum distance that has not been visited can be done by iterating over all elements of the minimum distance array and checking if the corresponding vertex has been visited. This takes $O(n)$ time. Lines 13-16 can be performed in $O(n)$ time. Therefore, lines 8-16 will cost $O(n^2)$ time. Then, in line 18, the shortest-path-r-arborescence of Y is computed using Dijkstra's algorithm, which, as discussed before, takes $O(q + n\log(n))$ time. As mentioned before, extracting a path from this r-arborescence, as done in line 20, can be done in constant time. Finally, this path is added to T, which has a complexity of $O(q+n)$. Therefore, lines 19-21 take $O(m(q+n))$ time. We conclude that the greedyrArborescence Algorithm has a complexity of $O(n + p + n^2 + q + n\log(n) + m(q + n)) = O(n^2 + n\log(n) + qm)$.

Because of how the arcs are selected, the solution will never contain a directed or undirected cycle. Note that if a terminal is the last vertex that can be added to the r-arborescence, all vertices that can be reached from the root will have been added. Since all terminals are assumed to be reachable from the root, all terminals will be part of the solution.



(a) Result greedyrArborescence Algorithm

(b) Result shortestPathsToTerminals

(c) Result shortestBunch and multipleBunches Algorithm

Figure 9: Example where the greedyrArborescence Algorithm performs better than the previously discussed algorithms

Figure 9 shows a graph with a root r, one start vertex s_1, three terminals x_1, x_2 and x_3, and three normal vertices v_1, v_2 and v_3. This is an example of a graph where the greedyrArborescence Algorithm outperforms the previously discussed algorithms. The shortestPathsToTerminals Algorithm performs worst on this example, with a value of 404, as shown in Figure 9b. To reach x_1, arc (s_1, v_1) must be used. Therefore, it is in any case more efficient to re-use this arc to get to x_2 and x_3 than to use arc (s_1, v_2) for this. A similar problem was discussed in Section 4.2, and Figure 9c shows that the shortestBunch Algorithm performs better at this part of the graph, with a value of 353. However, the same problem arises with arc (v_2, x_3). The multipleBunches Algorithm performs exactly the same since it splits the bunches at the start vertices, and there is only one start vertex here. The greedyrArborescence Algorithm, as shown in

Figure 9a returns the optimal solution in this case, which has a value of 302. In this example, all vertices are necessary in order to reach all terminals. Therefore, in this case, a minimum spanning arborescence is also an optimal solution for the directed Steiner tree problem.

To analyse how the algorithm performs in the worst case, we look at the approximation factor.

Theorem 5. *The greedyrArborescence Algorithm is an $(n-1)$-approximation algorithm for the directed Steiner tree problem.*

Proof. We will prove this by showing that, for any instance I, the greedyrArborescence returns a solution that (I) contains at most $n-1$ arcs and (II) contains no arcs of weight greater than $OPT(I)$.

Let an instance I be given by a directed graph $D=(V,A)$ with weight function $w: A \to \mathbb{R}_{\geq 0}$ defined on the arcs, a root vertex $r \in V$ and a set of terminals $X \subseteq V$ be given. Let $n=|V|$. Furthermore, let Y be the graph that is initialised as the empty graph, that the algorithm adds a vertex and an arc to in each iteration.

(I) The algorithm starts out by marking r as visited. In each iteration, the algorithm adds one arc (u,v) to graph Y, for a vertex v that has not yet been visited, and marks v as visited. Since there are n vertices in the graph, there can be at most $n-1$ iterations of the algorithm. Since each vertex can be visited at most once, the algorithm can add at most $n-1$ arcs to Y. Therefore, the solution returned by the greedyrArborescence Algorithm can contain at most $n-1$ arcs.

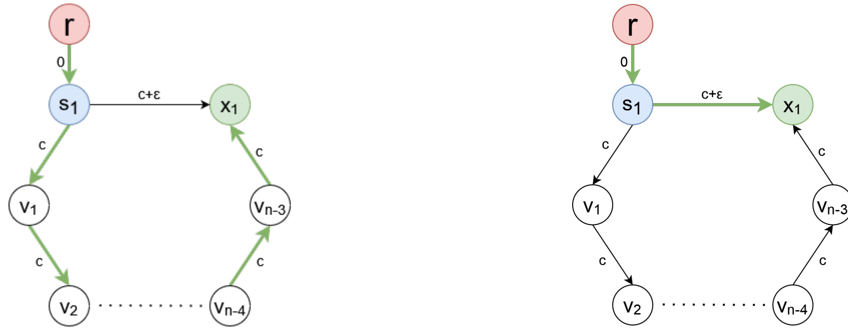
(II) We will show that the solution returned by the greedyrArborescence Algorithm will contain no arcs of weight strictly greater than the value of an optimal solution. To show this, we consider, for any iteration of the algorithm, an arc (u,v) of weight $c > OPT(I)$, for a vertex u that has already been visited and a vertex v that has not yet been visited. Here, $OPT(I)$ is the value of a minimum weight directed Steiner tree $T^* = (V^*, A^*)$ for instance I. We will show that arc (u,v) will never be added to Y. Since $c > OPT(I)$, we know that T^* cannot contain arc (u,v). We consider two cases: (a) $v \in V^*$; (b) $v \notin V^*$.

(a) Suppose $v \in V^*$, i.e. v is present in the optimal solution. Since T^* cannot contain arc (u,v), there must exist another path $P=(V_P, A_P)$ in graph D from r to v with weight at most $OPT(I)$. This implies that path P consists of arcs that each have weight at most $OPT(I)$, so for each $a \in A_P$ we have $w(a) \leq OPT(I)$. Since v has not been visited yet, and r has been visited, there must exist an arcs $(w,z) \in A_P$ such that w has been visited and z has not been visited, and this arc has weight at most $OPT(I)$. Then, (u,v) is not a minimum weight arc of which the source vertex has been visited and the target vertex has not been visited, and therefore (u,v) will not be added to Y.

(b) Suppose $v \notin V^*$, i.e. v is not in the optimal solution. This means that for each terminal $x \in X$, there exists a path from r to x in graph D that has weight at most $OPT(I)$ and that does not contain vertex v. If all terminals have been visited, the algorithm terminates, and (u,v) will not be added to Y. Otherwise, consider a terminal $x \in X$ that has not been visited and let $P=(V_P, A_P)$ be the path from r to x in T^*. Since r has been visited and x has not been visited, there exists an arc $(w,z) \in A_P$ such that w has been visited and z has not been visited. Since P has weight at most $OPT(I)$, we know that (w,z) has weight at most $OPT(I)$. This means that (u,v) is not a minimum weight arc of which the source vertex has been visited and the target vertex has not been visited, and therefore (u,v) will not be added to Y.

Since the solution returned by the greedyrArborescence Algorithm contains at most $(n-1)$ arcs and each arc has weight at most $OPT(I)$, we can conclude that the greedyrArborescence Algorithm returns a solution of value at most $(n-1)$ times the value of an optimal solution. Therefore, the greedyrArborescence Algorithm is an $(n-1)$-approximation algorithm for the directed Steiner tree problem. □

The approximation factor gives a guarantee that, for any instance, the algorithm will never return a solution worse than $n-1$ times the optimal solution. However, this is only an upper bound. Will the algorithm actually perform this bad? Figure 10 shows an example of a graph where the greedyrArborescence Algorithm does not perform well.



(a) Result greedyrArborescence Algorithm **(b)** Optimal solution

Figure 10: Example where the greedyrArborescence Algorithm performs poorly

Figure 10 shows a graph with a root vertex, one start vertex s_1, one terminal x_1 and $n-3$ normal vertices $v_1, ..., v_{n-3}$. The arcs (s_1, v_1), (v_1, v_2), ..., (v_{n-4}, v_{n-3}), (v_{n-3}, x_1) each have weight $c > 0$, and the arc (s_1, x_1) has weight $c + \epsilon$, for $\epsilon > 0$. The greedyrArborescence Algorithm will first add r to the solution. It will choose arc (r, s_1), then (s_1, v_1), (v_1, v_2), ..., (v_{n-4}, v_{n-3}) and finally (v_{n-3}, x_1). The solution that is returned is the minimum spanning r-arborescence of this graph, which has a total weight of $(n-2)c$. However, since we did not need vertices $v_1, ..., v_{n-3}$ to be part of the solution, the r-arborescence contains many arcs that we do not need. The optimal solution for our problem would simply be the arcs (r, s_1) and (s_1, x_1), which gives an r-arborescence of weight $c + \epsilon$. For this example, the shortestPathsToTerminals, shortestBunch and multipleBunches Algorithm all give the optimal solution, and therefore outperform the greedyrArborescence Algorithm. If we let $\epsilon \to 0$, this example shows that the greedyrArborescence Algorithm can return solutions almost as bad as $n-2$ times as bad as the optimal solution. This shows that the upper bound shown in Theorem 5 is not too far from how the algorithm can actually perform.

5 Exact algorithm

In the previous chapter, we discussed several approximation algorithms. Some have better time complexities than others, however, they all run in polynomial time. Since the directed Steiner tree problem is NP-hard, we know that, unless P=NP, there is no polynomial time exact algorithm. Nonetheless, an exact algorithm can give insight in the problem and can offer a good comparison to the previously discussed approximation algorithms.

This chapter will discuss an alteration of the Dreyfus-Wagner algorithm. The Dreyfus-Wagner algorithm, introduced in 1971, is a well-known exact algorithm for the Steiner tree problem. It is a dynamic programming algorithm that solves the Steiner tree problem to optimality for finite undirected graphs [Dreyfus and Wagner(1971)]. As we will see, the algorithm can be adapted to solve the directed Steiner tree problem to optimality.

The Dreyfus-Wagner algorithm originally only computes the value of the minimum weight Steiner tree, rather than the entire tree. However, the algorithm can be easily adjusted to return the entire tree [Korte et al.(2011)]. The idea of the Dreyfus-Wagner algorithm is to iteratively add new vertices to the tree until all terminal vertices are contained in the tree. In each iteration, the algorithm considers all vertices currently in the tree as possible vertices to connect the new terminal to. It chooses the vertex from which the distance to the new terminal is shortest. In the original paper, the algorithm stores the minimum cost of connecting a vertex to the current tree in a table. The algorithm can be adjusted to store the actual trees in a table.

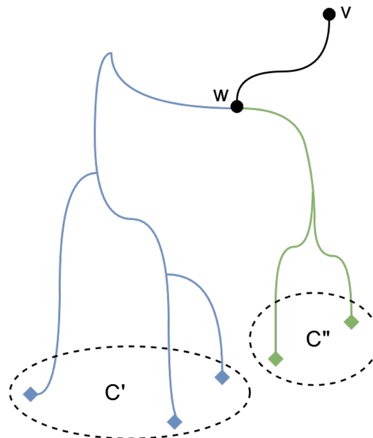


Figure 11: Steiner tree $T(C, v)$ for a vertex w and bipartition $C = C' \cup C''$

For an undirected graph $G = (V, A)$, we let $T(X)$ be the Steiner tree for terminal set X. The Dreyfus-Wagner algorithm is based on the observation that any leaf v of a minimum weight Steiner tree $T(C)$ is connected to the rest of the tree at a vertex w via a shortest path $P_{v,w}$ [Fuchs et al.(2007)]. Vertex w splits the minimum weight Steiner tree rooted at w for terminals C into two trees $T(C', w)$ and $T(C'', w)$, for some bipartition $C = C' \cup C''$. This is visualised in Figure 11. As shown by Korte et al.(2011), we can write $T(C, v) = P_{v,w} \cup T(C', w) \cup T(C'', w)$ for some shortest $v - w$ path $P_{v,w}$ and trees $T(C', w)$ and $T(C'', w)$, such that weight$(T(C, v)) = \min \text{weight}(P_{v,w} \cup T(C', w) \cup T(C'', w))$. Here, the minimum is taken over all vertices $w \in V$ and all nontrivial bipartitions $C = C' \cup C''$. The Dreyfus-Wagner algorithm uses this by recursively computing all minimum weight Steiner trees $T(C, v)$ for subsets C of the terminals X, and vertices $v \in V$. This way, the minimum weight Steiner tree for terminals X can be computed.

The above idea holds for connected undirected graphs. However, in a directed graph, not all vertices might be reachable from some other vertices, even if the underlying undirected graph is connected. The DirectedDreyfusWagner Algorithm is an alteration of the Dreyfus-Wagner algorithm for undirected graphs,

which takes this into account. We consider a directed weighted graph $D = (V, A)$ with weight function $w_D : A \to \mathbb{R}_{\geq 0}$, a set of terminals $X \subseteq V$ and a root $r \in V$. In the context of directed graphs, we define $T(C, v)$ to be the minimum weight v-arborescence that contains all vertices C, for any $v \in V$ and $C \subseteq X$. The DirectedDreyfusWagner Algorithm uses the OptimalvArborescence Algorithm to compute the r-arborescences $T(C, v)$ for any set of terminals C and vertex v.

Algorithm DirectedDreyfusWagner: Computes a minimum weight directed Steiner tree

---

Data: Directed graph $D = (V, A)$, weight function $w_D : A \to \mathbb{R}_{\geq 0}$, root $r \in V$, set of terminals $X \subseteq V$

Result: r-arborescence $T = (V', A')$ that is a subgraph of D, such that $X \subseteq V'$

```
1  graph T ← (∅, ∅);
2  if m = 1 then
3  │  T(X, r) ← shortest path from r to terminal x;
4  else
5  │  for terminal x ∈ X do
6  │  │  for vertex v ∈ V do
7  │  │  │  T({x}, v) ← shortest path from v to x;
8  │  for i = 2, ..., m do
9  │  │  for all subsets of the terminals C ⊆ X of size i do
10 │  │  │  for all vertices v ∈ V do
11 │  │  │  │  run the OptimalvArborescence Algorithm for input D = (V, A), v, C ;
12 T ← T(X, r)
```

Algorithm OptimalvArborescence: Computes the optimal v-arborescence that contains a given set of terminals C

Data: Directed graph $D = (V, A)$, weight function $w_D : A \to \mathbb{R}_{\geq 0}$, vertex $v \in V$, set of terminals $C \subseteq V$

```
1  Y ← (∅, ∅);
2  real number minimum ← ∞;
3  for all nontrivial bipartitions C = C′ ∪ C″ do
4  │  for all vertices w ∈ V do
5  │  │  if T(C′, w), T(C″, w) and P_{v,w} all exist then
6  │  │  │  if weight(T(C′, w)) + weight (T(C″, w)) + weight(P_{v,w}) < minimum then
7  │  │  │  │  minimum ← weight(T(C′, w)) ∪ weight (T(C″, w)) ∪ weight(P_{v,w});
8  │  │  │  │  Y ← T(C′, w) + T(C″, w) + P_{v,w};
9  if minimum < ∞ then
10 │  T(C, v) ← Y;
```

If there is only one terminal, the DirectedDreyfusWagner Algorithm simply computes the shortest path from the root to this terminal. Otherwise, the algorithm starts by initialising the $T(C, v)$ v-arborescences for terminal sets C of size 1. It does this by computing the shortest paths from each vertex $v \in V$ to each terminal $x \in X$, and saving these paths. Then, for each subset of the terminals and each vertex $v \in V$, the algorithm computes and saves $T(C, v)$. It does this by, in each iteration, increasing the size of the subsets it considers by 1. This way, the algorithm can always use the previously computed values of $T(\cdot\,, \cdot)$. Finally, the algorithm returns the $T(X, r)$ r-arborescence, which is the minimum weight r-arborescence that contains all terminals.

Computing $T(C, v)$ for a set C of size at least 2, and any vertex $v \in V$, is done with the OptimalvArborescence Algorithm. This algorithm considers all nontrivial bipartitions $C = C' \cup C''$ and all vertices $w \in V$ to find the minimum weight v-arborescence that is the union of the shortest path from v to w, $T(C', w)$ and $T(C'', w)$. Note that, the algorithm considers all options for $C' \cup C''$ and w, and D only contains arcs of strictly positive weights, with the exception of the arcs from r to the start vertices. Therefore, the OptimalvArborescence Algorithm will always return a v-arborescence. We will now prove that the DirectedDreyfusWagner Algorithm indeed returns an optimal solution to the directed Steiner tree problem.

Theorem 6. *Let a directed graph $D = (V, A)$ with weight function $w_D : A \to \mathbb{R}_{\geq 0}$, a set of terminals $X \subseteq V$ and a root $r \in V$ be given. The directed graph returned by the DirectedDreyfusWagner Algorithm is a minimum weight directed Steiner tree.*

Proof. Let a directed graph $D = (V, A)$ with weight function $w_D : A \to \mathbb{R}_{\geq 0}$, a set of terminals $X \subseteq V$ and a root $r \in V$ be given. Define $MDST_D(v, C)$ to be a minimum weight directed Steiner tree on graph D for some set of terminals $C \subseteq X$ with $|C| \geq 2$, rooted at some vertex $v \in V \setminus X$. First, note that $MDST_D(v, C)$ can be split into a shortest $v - w$ path $P_{v,w}$ and two minimum weight directed Steiner trees $MDST_D(w, C')$ and $MDST_D(w, C'')$, for some vertex $w \in V$ and some nontrivial partition $C = C' \cup C''$. This is visualised in Figure 12. Note that we may have $v = w$, in which case the shortest path from v to w has length 0, and all bipartitions of C give the same result. Because the minimum weight directed Steiner tree can be split in this way, we know that we can construct $MDST_D(v, C)$ by combining a shortest path from v to w and two minimum weight directed Steiner trees $MDST_D(w, C')$ and $MDST_D(w, C'')$, for some $w \in V$ and some nontrivial partition $C = C' \cup C''$. In fact, since $MDST_D(v, C)$ is a minimum weight directed Steiner tree, we have $MDST_D(v, C) = P_{v,w} \cup MDST_D(w, C') \cup MDST_D(w, C'')$, for some shortest path $P_{v,w}$ and two minimum weight directed Steiner trees $MDST_D(w, C')$ and $MDST_D(w, C'')$ such that weight$(MDST_D(v, C))$ = weight$(P_{v,w} \cup MDST_D(w, C') \cup MDST_D(w, C''))$. Here the minimum is taken over all $w \in V$ and all nontrivial bipartition $C = C' \cup C''$. We use this fact to prove that the algorithm returns a minimum weight directed Steiner tree.

First of all, suppose $m = 1$, i.e. there is exactly one terminal. In this case, the minimum weight directed Steiner tree is simply a shortest path from r to the terminal, which is exactly what the algorithm returns.

Now suppose $m > 1$. The algorithm saves the minimum weight directed Steiner trees for subsets of the terminals that have size 1, by computing the shortest path from any vertex $v \in V$ to any terminal $x \in X$, and saving this as $T(\{x\}, v)$. These are the minimum weight directed Steiner trees for terminal sets $\{x\}$, rooted at v. We will now prove by induction that, in any iteration $i \in \{2, ..., m\}$, for any subset of the vertices $C \subseteq X$ such that $|C| = i$, and for any vertex $v \in V$, the directed graph returned by the OptimalvArborescence Algorithm $T(C, v)$ is a minimum weight directed Steiner tree rooted at v, containing terminals C.

First, let $i = 2$. Consider a set $C \subseteq X$ such that $|C| = 2$, and a vertex $v \in V$. Define $C := \{c_1, c_2\}$. The OptimalvArborescence Algorithm now computes a directed graph $T(C, v)$ which is the combination of $P_{v,w}$, $T(\{c_1\}, w)$ and $T(\{c_2\}, w)$, for a vertex $w \in V$ that minimises the sum of the weights of these elements. $T(\{c_1\}, w)$ and $T(\{c_2\}, w)$ have already been computed in a previous step, and are the shortest paths from w to c_1 and c_2, respectively. It is easy to see that this is a minimum directed Steiner tree in D for terminals C and root v.

Now let k be some integer greater than 2. Assume the OptimalvArborescence Algorithm returns a minimum weight directed Steiner tree for any $C \subseteq X$ of size $i < k$ and any vertex $v \in V$. Then, let $\overline{C}$ be any subset of X of size k, and let $z \in V$ be any vertex. The algorithm will return a graph $T(\overline{C}, z)$ that is the combination of $P_{z,w}$, $T(\overline{C}', w)$ and $T(\overline{C}'', w)$, for a vertex $w \in V$ and a nontrivial bipartition $\overline{C} = \overline{C}' \cup \overline{C}''$ that minimises the sum of the weights of these elements. The sets $\overline{C}'$ and $\overline{C}''$ both have size strictly smaller than k, since we only consider nontrivial bipartitions. Therefore, by the induction hypothesis, $T(\overline{C}', w)$ and $T(\overline{C}'', w)$ are minimum weight directed Steiner trees rooted

at w for terminals $\overline{C}'$ and $\overline{C}''$, respectively. Since the algorithm chooses the vertex w and bipartition that minimise the sum of the weights of the elements, the result is again a minimum weight directed Steiner tree.

By induction, we conclude that the OptimalvArborescence Algorithm returns a minimum weight directed Steiner tree rooted at v, containing terminals C, for any $v \in V$ and any subset of the terminals.

The DirectedDreyfusWagner Algorithm returns the graph $T(X, r)$. Because of what we have just proven, this is a minimum weight directed Steiner tree rooted at r containing the terminals X. □

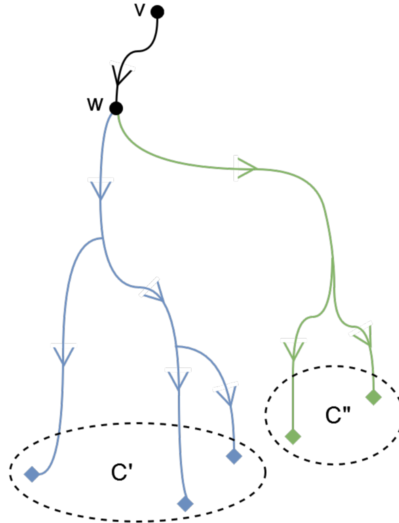


Figure 12: Directed Steiner tree rooted at v, with a shortest path from v to w and a bipartition of the vertices $C = C' \cup C''$

The fact that the DirectedDreyfusWagner Algorithm is guaranteed to give an optimal solution, comes at a cost: the complexity of the algorithm is exponential. The shortest paths between each pair of vertices can be computed at the beginning of the DirectedDreyfusWagner Algorithm. Computing the shortest-paths-r-arborescence can be done in $O(q + n\log(n))$ time using Dijkstra's algorithm, and therefore computing the shortest paths between each pair of vertices can be done in $O(nq+n^2\log(n))$ time. In the OptimalvArborescence Algorithm, we first compute all non-trivial bipartitions of C. This takes $O(2^{|C|-1})$ time, since there are $O(2^{|C|-1})$ non-trivial bipartitions. Then, we iterate over all those bipartitions and over all vertices $w \in V$. We then compute the weight of the shortest $v - w$ path and the two minimum weight directed Steiner trees, which can each take $O(q)$ time, since we need to add the weights of all arcs. Adding these three graphs to graph Y has complexity $O(n+q)$. Finally, the algorithm sets $T(C, v)$ to Y, which again has complexity $O(n+q)$. We can conclude that the OptimalvArborescence Algorithm runs in $O(2^{|C|-1}(1+n(n+2q))+n+q) = O(2^{|C|-1}n(n+2q))$.

The DirectedDreyfusWagner Algorithm starts, as mentioned, by computing the shortest paths for all pairs of vertices, which takes $O(nq+n^2\log(n))$ time. If $m > 1$, the algorithm first sets $T(\{x\}, v)$ to the shortest path from v to x, for each terminal $x \in X$ and each vertex $v \in V$. This takes $O(mn)$ time. Then, for each $i = 2, .., m$, we first compute all subsets of X of size i. This can be done in $O(m^i)$ time using dynamic programming. We run the OptimalvArborescence Algorithm for all these subsets, and all vertices $v \in V$. In total, this takes

$$O(\sum_{i=2}^{m} m^i(1+2^{i-1}n^2(n+2q))) = O\left(\frac{m(m^m-1)}{m-1} - m + \frac{1}{2}n^2(n+2q)\left(\frac{2m((2m)^m-1)}{2m-1} - 2m\right)\right).$$

This means the complexity of the DirectedDreyfusWagner Algorithm is exponential in the number of terminals.

6 ILP based solution methods

In the previous sections we have discussed several approximation algorithms and an exact algorithm. The approximation algorithms have good time complexities, but can, depending on the input, return solutions that are far from optimal. The exact algorithm that was discussed does not run in polynomial time, as we already expected since the directed Steiner tree problem is an NP-hard problem. Aside from combinatorial algorithms, we can look at *linear programming*. A Linear Program (LP) is a problem where a linear function needs to be minimised or maximised, subject to a set of linear constraints. A linear program can always be expressed in the following form.

$$\min \quad \mathbf{c}^\top \mathbf{x} \tag{4a}$$

$$\text{s.t. } A\mathbf{x} \leq \mathbf{b} \tag{4b}$$

$$\mathbf{x} \geq 0 \tag{4c}$$

Here, $\mathbf{x}$ is the vector of *decision variables*, and the components of $\mathbf{c}$, $\mathbf{b}$ and A are given constants. We call (4a) the *objective function*. A vector $\mathbf{x}$ is called a *feasible solution* if it satisfies all constraints. We define a vector $\mathbf{x}$ to be an *optimal solution* if it is feasible and there is no other feasible solution $\widetilde{\mathbf{x}}$ such that $\mathbf{c}^\top \widetilde{\mathbf{x}} < \mathbf{c}^\top \mathbf{x}$.

An LP that has a constraint that restricts all decision variables to be integer is called an *Integer Linear Program* (ILP). Whereas LPs can be solved efficiently, ILPs cannot. The ILP problem is known to be NP-complete [Schrijver(1998)]. This means that, unless P=NP, there exists no polynomial time algorithm to solve the problem. Linear programs that contain both integer and continuous decision variables are called Mixed Integer Programs (MIP).

A big advantage of formulating the problem as an ILP is that there exist LP solvers that use various strategies and algorithms to efficiently solve an LP. Of course, if the ILP is too hard, the solver will still take very long to solve the problem, since ILP's are NP-complete. How well a solver performs very much depends on the problem formulation. Furthermore, one can implement algorithms that "collaborate" with the solver, to help solve the ILP faster. Since the solver only knows the ILP formulation, and does not have any additional information about the problem and its context, it can be beneficial to come up with problem-specific methods to help the solver.

There are various ways of formulating the directed Steiner tree problem as an ILP. In this section we will discuss three such formulations.

6.1 Directed cut formulation

The first ILP formulation that will be discussed is the directed cut formulation, as described by Friggstad et al.(2014). Let graph $D = (V, A)$ and weight function w be defined as in Section 3.1, and let $X \subset V$ and $r \in V$. For any set of vertices $U \subseteq V$ we define the set of arcs entering U as $\delta^-(U) = \{(v, u) | v \in V \setminus U, u \in U\}$. Then, we define the set $\mathcal{C} = \{\delta^-(U) | U \subseteq V \setminus \{r\}, U \cap X \neq \emptyset\}$. The directed Steiner tree problem can be formulated as an integer linear program as described in (5).

$$\min \quad \sum_{a \in A} w(a) y_a \tag{5a}$$

$$\text{s.t. } \sum_{a \in C} y_a \geq 1 \qquad \forall C \in \mathcal{C} \tag{5b}$$

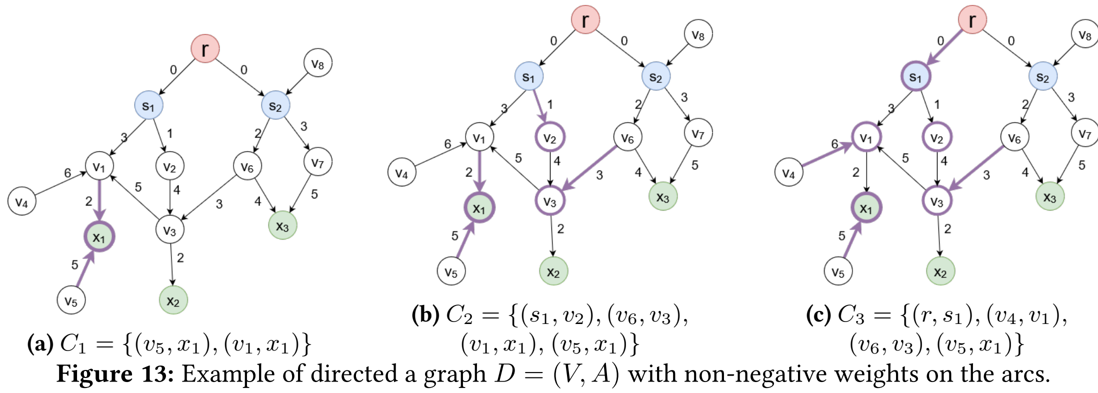
$$y_a \in \{0, 1\} \qquad \forall a \in A. \tag{5c}$$

Here, the binary decision variables y_a decide whether an arc $a \in A$ is used in the solution. The objective function minimises the sum of the weights of the arcs that are used in the solution. Constraint (5b) ensures

that, for every set of vertices that contains a terminal but not the root, there is at least one arc entering this set. This guarantees a path from the root to each of the terminals.

Figure 13 shows an example of a directed graph with non-negative arcs weights. The start vertices s_1 and s_2 are marked in blue, and the terminals x_1, x_2 and x_3 are marked in green. Figure 13a visualises the set $C_1 := \{\delta^-(U_1) | U_1 = \{x_1\}\} = \{(v_5, x_1), (v_1, x_1)\}$. This shows that constraints corresponding to a set containing solely a terminal, ensure that there is an incoming arc for each terminal.

Figure 13b visualises the set $C_2 := \{\delta^-(U_2) | U_2 = \{v_2, v_3, x_1\}\} = \{(s_1, v_2), (v_6, v_3), (v_1, x_1), (v_5, x_1)\}$, and Figure 13c shows the set $C_3 := \{\delta^-(U_3) | U_3 = \{s_1, v_1, v_2, v_3, x_1\}\} = \{(r, s_1), (v_4, v_1), (v_6, v_3), (v_5, x_1)\}$. In Figure 13c we see that the constraint ensures that x_1 is reached via at least one of the incoming arcs of C_3. The set U_3 can be entered through, for example, s_1 or v_3. Since there is a constraint for each subset containing a terminal but not the root, we ensure a path from the root to each terminal. We immediately see from Figure 13c why we want to exclude r from the subsets; all terminals need to be reached from the root. The subsets do always need to contain a terminal, since not all vertices in the graph need to be reached.



(a) $C_1 = \{(v_5, x_1), (v_1, x_1)\}$

(b) $C_2 = \{(s_1, v_2), (v_6, v_3), (v_1, x_1), (v_5, x_1)\}$

(c) $C_3 = \{(r, s_1), (v_4, v_1), (v_6, v_3), (v_5, x_1)\}$

Figure 13: Example of directed a graph $D = (V, A)$ with non-negative weights on the arcs.

Computing all subsets of a set can be done using backtracking. Algorithm allCorrectSubsets is an algorithm that uses backtracking algorithm correctSubset for each terminal $x \in X$, to compute all subsets C of $V \setminus \{r\}$ that contains x. The allCorrectSubsets Algorithm initiates the set of subsets $\mathcal{C}$ to be the empty set. Then, for each terminal $x \in X$, the algorithm removes x from the set we take subsets of, and reduces the total number of iterations N by one. This is done to avoid subsets that contain multiple terminals to be added multiple times. Then, the algorithm runs the correctSubset Algorithm for input $i = 0$, $Y = \emptyset$, $Z = V'$ and $v = x$. Here, index i counts the number of iterations, Y is the subset that is being constructed, Z is the list we take subsets from, and v is the vertex that must be included in the subset. For each $x \in X$, the correctSubset Algorithm is called N times. In each call, the algorithm considers both adding and not adding element i of the input list, and runs the correctSubset Algorithm for both choices. The index i is also increased by 1. After the algorithm has considered all N elements of Z, we add vertex v to the subset that has been created and the subset is added to $\mathcal{C}$.

One can visualise the correctSubset algorithm as a decision tree, where each decision is the choice of whether to add or not to add a certain element of Z to subset Y. This can be seen in Figure 14, which visualises the correctSubset Algorithm for terminal x_{k-1}. This shows how, in each iteration, the subset either gains a vertex or remains the same. After N iterations, all elements of the input list have been handled. After this, the terminal is added to the subset.

Algorithm allCorrectSubsets: Algorithm that uses backtracking to compute all subsets of the input list $V \setminus \{r\}$ that contain at least one terminal $x \in X$.

Data: vertices $V \setminus \{r\}$, terminals X
Result: set of all subsets C of $V \setminus \{r\}$ such that $C \cap X \neq \emptyset$

```
1 𝒞 ← ∅;
2 N ← n − 1;
3 V′ ← V \ {r};
4 for terminal x in X do
5     remove x from V′;
6     N ← N − 1 ;
7     run the correctSubset Algorithm for i = 0, Y = ∅, Z = V′ and v = x;
```

Algorithm correctSubset: Backtracking algorithm that is used to compute all subsets of input list Z that contain a certain vertex v.

Data: current iteration i, current subset Y, input list Z, vertex v

```
1 if i == N then
2     C ← Y ∪ {v};
3     add C to 𝒞;
4     return;
5 run the correctSubset Algorithm for i = i + 1, Y = Y, Z = Z and v = v;
6 run the correctSubset Algorithm for i = i + 1, Y = Y ∪ {i^th element of Z}, Z = Z and v = v;
```

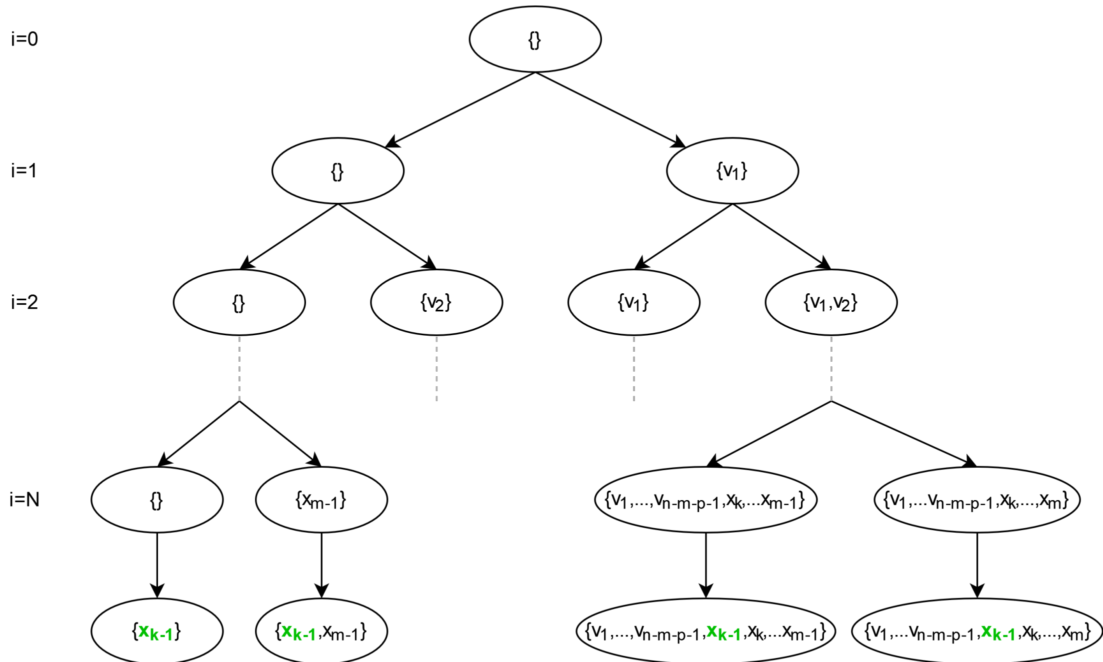


Figure 14: Backtracking algorithm for the $(k-1)^{\text{st}}$ terminal

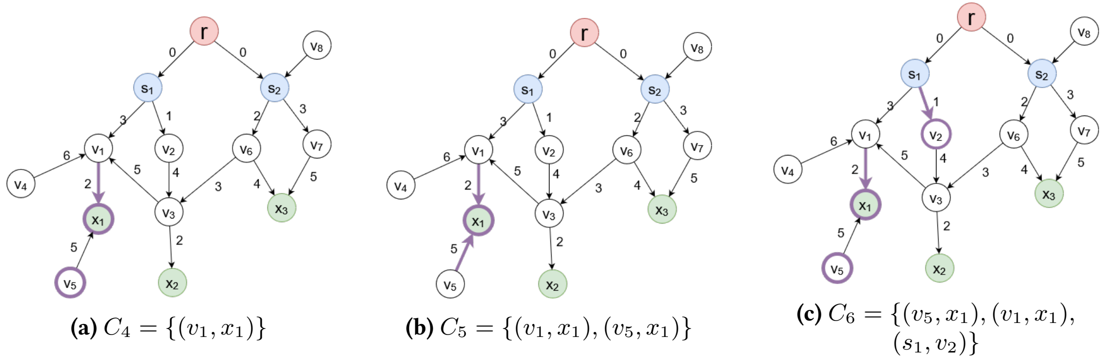
It is important to note that the allCorrectSubsets Algorithm takes exponential time. For the ith terminal, the list V' that is being considered has $n-1-i$ elements. This means that the correctSubset Algorithm is called 2^{n-i} times, where the last iteration consists of adding v to the subset and adding the obtained subset to $\mathcal{C}$. This means that the correctSubset Algorithm is called a total of $\sum_{i=1}^{m} 2^{n-i} = 2^{n-m}(2^m - 1)$ times. Since each call happens in constant time, the allCorrectSubsets Algorithm runs in $O(2^{n-m}(2^m - 1))$

time. This also means that there are $2^{n-m}(2^m - 1)$ subsets, and therefore constraints, in ILP (5).

6.1.1 Constraints based on minimal sets of arcs

Computing the constraints for linear program (5) takes exponential time, and produces an exponential number of constraints. This will make the ILP harder to solve. In an attempt to improve this, we look at which constraints are necessary for the ILP to be feasible.

One method to decrease the number of constraints is considering only minimal sets of incoming arcs to subsets not containing the root and containing at least one terminal. That is: we redefine the set $\mathcal{C}$ to be the set $\{\delta^-(U) | U \subseteq V - \{r\}, U \cap X \neq \emptyset\}$ for which holds that for every $C \in \mathcal{C}$, there exists no $C' \in \mathcal{C}$ such that $C' \subset C$. The effect of this will be shown with an example.



(a) $C_4 = \{(v_1, x_1)\}$ **(b)** $C_5 = \{(v_1, x_1), (v_5, x_1)\}$ **(c)** $C_6 = \{(v_5, x_1), (v_1, x_1), (s_1, v_2)\}$

Figure 15: Example of a directed graph $D = (V, A)$ with non-negative weights on the arcs.

Figure 15 shows the same graph as before. It visualises three different subsets of the vertices and the corresponding sets of incoming arcs. Figure 15a corresponds to $C_4 = \{\delta^-(U_4) | U_4 = \{x_1, v_5\}\} = \{(v_1, x_1)\}$; Figure 15b corresponds to $C_5 = \{\delta^-(U_5) | U_5 = \{x_1\}\} = \{(v_1, x_1), (v_5, x_1)\}$; and Figure 15c corresponds to $C_6 = \{\delta^-(U_6) | U_6 = \{x_1, v_5, v_2\}\} = \{(v_5, x_1), (v_1, x_1), (s_1, v_2)\}$. Note that C_4 is a proper subset of both C_5 and C_6. We can see in (6a) that the constraint corresponding to C_4 makes the ones corresponding to C_5 and C_6 obsolete. We see that constraints corresponding to a non-minimal set of arcs become obsolete because of the constraints corresponding to its proper subsets. Therefore, it is sufficient to only consider minimal sets of incoming arcs.

$$C_4: \quad y_{v_1,x_1} \geq 1 \tag{6a}$$
$$C_5: \quad y_{v_1,x_1} + y_{v_5,x_1} \geq 1 \tag{6b}$$
$$C_6: \quad y_{v_1,x_1} + y_{v_5,x_1} + y_{s_1,v_2} \geq 1 \tag{6c}$$

For the example shown in Figure 15, there is a total of 7168 subsets of $V \setminus \{r\}$ that contain at least one terminal. If we reduce this to the subsets of which the incoming arc sets have no proper subsets in $\mathcal{C}$, only 15 subsets remain. This means that the number of constraints in (5) is reduced from 7168 to 15.

To extract the minimal sets of incoming arcs, we first have to compute all subsets that do not contain the root but contain a terminal, and then check which incoming arc sets have proper subsets. This is done with Algorithm minimalSubsets. This algorithm takes as input a directed graph $D = (V, A)$, a weight function $w : A \to \mathbb{R}_{\geq 0}$ and a set $\mathcal{C}_{\text{old}}$ containing subsets of V. The algorithm copies $\mathcal{C}_{\text{old}}$ into the set $\mathcal{C}$, which will contain only the minimal subsets. The algorithm removes subsets that are not minimal from $\mathcal{C}$ by iterating over all pairs of subsets $(\mathcal{C}(i), \mathcal{C}(j))$. Here, $\mathcal{C}(i)$ is the ith element of $\mathcal{C}$. For each such pair with $i \neq j$, the algorithm checks if $delta^-(\mathcal{C}(i)) \subset delta^-(\mathcal{C}(j))$. If this is the case, $delta^-(\mathcal{C}(j))$ is not minimal, and thus the subset $\mathcal{C}(j)$ is removed from $\mathcal{C}$. Otherwise, $delta^-(\mathcal{C}(j)) \subset delta^-(\mathcal{C}(i))$, and

$\mathcal{C}(i)$ is removed from $\mathcal{C}$. Since it is not possible to remove elements from a set you are iterating over, the algorithm sets elements that need to be removed to null, and removes all null entries of $\mathcal{C}$ at the end of the algorithm.

Algorithm minimalSubsets: Removes all subsets of vertices out of a set which corresponding incoming arc sets are not minimal. Note that the algorithm uses lists rather than sets, but the result should be interpreted as a set.

```
   Data: directed graph D = (V, A), weight function w : A → ℝ≥0, set C_old of subsets of V
   Result: set of subsets of vertices C, where the incoming arc sets of each C ∈ C are minimal
 1 C ← C_old;
 2 for subsets of vertices C in C do
 3     compute the incoming arcs of C;
 4 for i = 0, ..., |C| − 1 do
 5     if C(i) is null then
 6         continue to the next value of i;
 7     for j = 0, ..., |C| − 1 do
 8         if C(j) is null then
 9             continue to the next value of j ;
10         if i ≠ j then
11             if delta⁻(C(i)) ⊂ delta⁻(C(j)) then
12                 C(j) ← null;
13             else
14                 C(i) ← null;
15 Remove all null entries from C;
16 return C ;
```

The sets of incoming arcs of each subset are computed beforehand, to avoid having to compute these sets multiple times for some subsets. The set of incoming arcs of a subset is computed using the findIncoming-Arcs Algorithm. This algorithm iterates over all incoming arcs (u, v) of all vertices $v \in C$, and checks if the source vertex u of (u, v) is in the set C. If u is not in C, the arc comes from outside of C, and is therefore in the set of incoming arcs of C.

Algorithm findIncomingArcs: Returns the set of incoming arcs of a set of vertices.

```
   Data: directed graph D = (V, A), weight function w : A → ℝ≥0, subset C ⊆ V
   Result: δ⁻(C)
 1 for vertex v ∈ C do
 2     for arc (u, v) ∈ δ⁻(v) do
 3         if u ∉ C then
 4             add (u, v) to the set of incoming arcs of C ;
 5 return incomingArcs ;
```

The findIncomingArcs Algorithm checks for each vertex in $subset$ and each arc entering this vertex if the source of the arc is in $subset$. This means that it checks at most each arc $a \in A$. A set C can have size up to $O(n)$, and therefore the complexity of checking if something is in C is $O(n)$. Therefore, the findIncomingArcs Algorithm runs in $O(nm)$ time.

Algorithm minimalSubsets calls findIncomingArcs once for each subset. As stated before, there are $O(2^{n-m}(2^m - 1))$ subsets, giving lines 2-3 a complexity of $O(nm2^{n-m}(2^m - 1))$. Then, the algorithm iterates over all $(\mathcal{C}(i), \mathcal{C}(j))$ pairs, which are $|\mathcal{C}|^2 = 2^{2n-2m}(2^m - 1)^2$ pairs. For each pair, it checks if the

incoming arc set of $\mathcal{C}(i)$ is contained in the incoming arc set of $\mathcal{C}(j)$. This takes $O(n^2)$ time, since both sets can have a maximum of n elements. Therefore, lines 4-14 have a complexity of $O(n^2 2^{2n-2m}(2^m - 1)^2)$. Finally, removing all null entries from $\mathcal{C}$ takes $O(2^{n-m}(2^m - 1))$ time. This gives the minimalSubsets Algorithm a time complexity of $O(nm2^{n-m}(2^m - 1) + n^2 2^{2n-2m}(2^m - 1)^2 + 2^{n-m}(2^m - 1))$ $= O(n^2 2^{2n-2m}(2^m - 1)^2))$.

To compute the minimal subsets, one first needs to compute all subsets. This takes exponential time and memory. We have seen that computing the minimal subsets also takes exponential time. Therefore, even though this technique can significantly decrease the number of constraints, this method will likely not work very well in practise. Therefore, we investigate another method of handling an LP with a large number of constraints.

6.1.2 Adding constraints along the way

Since it costs a lot of time and memory space to compute all constraints beforehand, it is a good idea to look at adding constraints "on the fly". We start out with a much simpler version of our ILP, that is a relaxation and contains none or only a few constraints. This is called the *reduced master problem* (RMP). We let the solver solve the LP, and then check if the solution is feasible for our problem. If this is not the case, we add a constraint to the RMP, excluding this solution. This constraint is called a *cut*. Then, we let the solver solve the new RMP. This is repeated until a feasible solution is found. This solution is the optimal solution for our ILP.

Let the reduced master problem be defined as follows. Furthermore, we define the solution returned by the RMP as $\overline{y}$.

$$\min \quad \sum_{a \in A} w(a) y_a \tag{7a}$$

$$y_a \in \{0, 1\} \tag{7b}$$

There are various ways to compute cuts to add to the master problem. *Combinatorial cuts* are cuts that only exclude the current solution $\overline{y}$. The combinatorial cut for a solution to the master problem $\overline{y}$ is described in (8). This has a term $(1 - y_a)$ for each arc a that is currently used in the solution, and a term y_a for each arc a that is not used in the solution. The cut that is added says that the sum of these terms is at least 1. This means that at least one of the variables is "flipped" from 0 to 1, or from 1 to 0.

$$\sum_{a \in A : \overline{y}_a = 1} (1 - y_a) + \sum_{a \in A : \overline{y}_a = a} y_a \geq 1 \tag{8}$$

A combinatorial cut is added if there is no feasible solution. This can be checked by checking if the chosen arcs form paths from the root to each terminal, for example using Dijkstra's shortest path algorithm. Combinatorial cuts are intuitive and easy to implement. However, they only exclude one infeasible solution. Therefore, it can take an extremely long time before a feasible solution is found. It is better to add cuts that give the master problem more information, and therefore exclude more infeasible solutions.

A method to add more efficient cuts is by trying to find the "most violated constraint". We start by asking the question: does there exist a set of vertices $U \subseteq V \setminus \{r\}$ that contains a terminal such that $\sum_{a \in \delta^-(U)} \overline{y}_a < 1$? If no such set of vertices exists, $\overline{y}$ is a feasible solution. If such a set of vertices does exist, there is a violated constraint and we want to add a cut. Checking if such a cut exists can be done by first computing $F := \min_U \sum_{a \in \delta^-(U)} \overline{y}_a$, where U ranges over all sets of vertices that contain at least one terminal and do not contain the root. If $F < 1$, we have found a set U^* that induces a violated constraint. In this case we add cut (9) to the RMP for the set U^* that induces the "most violated constraint" $F = \sum_{a \in \delta^-(U^*)} \overline{y}_a$.

$$\sum_{a \in \delta^-(U^*)} y_a \geq 1 \tag{9}$$

The problem of finding the set U^* is defined as the *Most violated constraint problem*, as defined in (12).

Definition 12 (Most violated constraint problem). *Let a directed graph $D = (V, A)$ be given. Furthermore, let a function $\overline{y} : A \to [0, 1]$ be given. Define Y to be the set of all subsets of V that do not contain the root but contain at least one terminal, i.e. $Y := \{U \subseteq V \setminus \{r\} | U \cap X \neq \emptyset\}$. The* ***most violated constraint problem*** *is the problem of finding a set $U^* = \arg\min_{U \in Y} \sum_{a \in \delta^-(U)} \overline{y}_a$.*

Of course, computing all constraints for the original ILP (5) is very expensive, and we want to avoid this. Therefore, it is not preferred to simply iterate over all subsets that contain a terminal but not the root. Instead, we seek a more efficient method to find a set U^* that minimises $\sum_{a \in \delta^-(U)} \overline{y}_a$. Given the graph $D = (V, A)$, we construct a weight function $\widetilde{w} : A \to [0, 1]$ that is defined as $\widetilde{w}(a) = \overline{y}_a$ for each $a \in A$. The goal is to find a subset of the vertices that contains a terminal but not the root, for which the sum of the weights of the incoming arcs is smallest. We will see that this is equivalent to finding a *minimum cut*. First, we define an $s - t$ cut.

Definition 13. *Let a weighted directed graph $D = (V, A)$, a weight function $w : A \to \mathbb{R}_{\geq 0}$ on the arcs, a source $s \in V$ and a sink $t \in V$ be given. An* ***s-t cut*** *$C(S_s, T_t)$ is a partition of the vertices V into the sets S_s and T_t, where $s \in S$ and $t \in T$.*

Furthermore, we define the cut-set.

Definition 14. *The* ***cut-set*** *B_C of $C(S_s, T_t)$ is the set of arcs from vertices in S_s to vertices in T_t. That is, $B_C := \{(u, v) | u \in S_s, v \in T_t\}$.*

In order to compare cuts, we define the value of a cut.

Definition 15. *The* ***value*** *$h(C)$ of a cut is the sum of the weights of the arcs of the cut-set $h(C) = \sum_{a \in B_C} w(a)$.*

Finally, we can define the *minimum cut problem*.

Definition 16 (Minimum cut problem). *Let a weighted directed graph $D = (V, A)$, a weight function $w : A \to \mathbb{R}_{\geq 0}$ on the arcs, a source $s \in V$ and a sink $t \in V$ be given. The* ***minimum cut problem*** *is the problem of finding an $s - t$ cut of minimum value. That is, finding an $s - t$ cut $\hat{C}(\hat{S}_s, \hat{T}_t) = \arg\min_{C_{(S_s, T_t)}} \sum_{a \in B_C} \widetilde{w}(a)$*

Theorem 7. *Let a graph $D = (V, A)$ and a weight function $\overline{y} : A \to [0, 1]$ on the arcs be given. Furthermore, let a vertex $r \in V$ and a set of terminals $X \subset V$ be given. U^* is set of vertices that induces a most violated constraint if and only if $\hat{C}(V \setminus U^*, U^*) = \min_{x \in X} C(S_r, T_x)$.*

Proof. Let a directed graph $D = (V, A)$ and a weight function $\overline{y} : A \to [0, 1]$ on the arcs be given. Furthermore, let a vertex $r \in V$ and a set of terminals $X \subset V$ be given.

$\Rightarrow$ Define $Y := \{U \subseteq V \setminus \{r\} | U \cap X \neq \emptyset\}$. Let the solution to the most violated constraint problem be $U^* = \arg\min_{U \in Y} \sum_{a \in \delta^-(U)} \overline{y}_a$. Construct a graph $D' = (V, A)$ with a weight function on the arcs $\widetilde{w} : A \to [0, 1]^q$. This graph has the same vertices and arcs as D. Furthermore, define the set $S := V \setminus U^*$. Let $x \in X$ be a terminal that is contained in U^*. Note that, since U^* does not contain the root, $r \in S$. We now have an $r - x$ cut $C := (S, U^*)$. The cut-set of this cut is $B_C = \{(u, v) | u \in S, v \in U^*\}$, which is the set of arcs entering U^*. Since U^* is the set of vertices in Y for which the sum of the arc weights of the entering arcs is minimal, C is a minimum $r - x$ cut.

$\Leftarrow$ Let $\hat{C}(V \setminus U^*, U^*) := \min_{x \in X} C(S_r, T_x)$ be the minimum of the minimum $r - x$ cuts for terminals $x \in X$. We then know that U^* contains a terminal and does not contain the root. Let x^* be a terminal

such that $x^* \in U^*$. We know that $\hat{C}(V \setminus U^*, U^*)$ is the partition of vertices that separates r and x^* that minimises the sum of the weights of the arcs that start in $V \setminus U^*$ and end in U^*. That means $\hat{C}(V \setminus U^*, U^*) = \min_{x \in X} \min_{C(S_r, T_x)} \sum_{a \in B_C} \overline{y}_a$. Since B_C is the set of arcs that start in S_r and end in $T_x = V \setminus S_r$, we can write $\hat{C}(V \setminus U^*, U^*) = \min_{x \in X} \min_{T_x \subseteq V \setminus \{r\}} \sum_{a \in \delta^-(T_x)} \overline{y}_a$, where T_x is a set that contains x. This means that U^* is a set that contains a terminal and does not contain the root, such that the sum of the weights of the arcs entering U^* is minimal. Define the set $Y := \{U \subseteq V \setminus \{r\} | U \cap X \neq \emptyset\}$. Then we can write $\hat{C}(V \setminus U^*, U^*) = \min_{U \in Y} \sum_{a \in \delta^-(U)} \overline{y}_a$, and $U^* = \arg\min_{U \in Y} \sum_{a \in \delta^-(U)} \overline{y}_a$. Therefore, U^* is the set that induces a most violated constraint. □

We can use Theorem 7 to compute the most violated constraint, by computing the minimum $r - x$ cut for each $x \in X$ and taking the minimum of the results. The minimum cut problem in directed graphs can be solved in $O(qn \log(n^2/q))$ time [Hao and Orlin(1994)], and therefore the most violated constraint problem can be solved in $O(mqn \log(n^2/q))$ time.

6.2 Flow based formulation

Another ILP formulation is based on the flow through the arcs. An example of a flow formulation for an acyclic graph was described by Rothvoß(2011). Since we consider cyclic graphs, the linear program is slightly different. The ILP formulation (10) sends a flow from the root to each of the terminals $x \in X$. There is a decision variable to determine whether flow to a terminal x goes through a certain arc. The weights of the arcs are only taken into account in the objective function; where the sum of the weights of the used arcs is minimised. The model is explained in more detail below.

$$\min \sum_{a \in A} w(a) y_a \tag{10a}$$

$$\sum_{a \in \delta^+(r)} f_{x,a} = 1 \qquad \forall x \in X \tag{10b}$$

$$\sum_{a \in \delta^-(x)} f_{x,a} = 1 \qquad \forall x \in X \tag{10c}$$

$$\sum_{a \in \delta^+(x)} f_{x,a} = 0 \qquad \forall x \in X \tag{10d}$$

$$\sum_{a \in \delta^+(q)} f_{x,a} - \sum_{a \in \delta^-(q)} f_{x,a} = 0 \qquad \forall x \in X \ \forall q \in V \setminus \{\{r\} \cup \{x\}\} \tag{10e}$$

$$\sum_{a \in \delta^-(q)} f_{x,a} \leq 1 \qquad \forall x \in X \ \forall q \in V \setminus \{\{r\} \cup \{x\}\} \tag{10f}$$

$$f_{x,a} \leq y_a \qquad \forall x \in X \ \forall a \in A \tag{10g}$$

$$y_a \leq \sum_{x \in X} f_{x,a} \qquad \forall a \in A \tag{10h}$$

$$y_a \in \{0,1\} \qquad \forall a \in A \tag{10i}$$

$$f_{x,a} \in \{0,1\} \qquad \forall x \in X \ \forall a \in A \tag{10j}$$

For each arc $a \in A$ there is a binary decision variable y_a, that decides whether an arc is used in the result. If $y_a = 1$, arc a is used, and if $y_a = 0$, the arc is not used. Furthermore, each terminal-arc pair (x, a), $x \in X$, $a \in A$, has a binary decision variable $f_{x,a}$. If $f_{x,a} = 1$, arc a is used on the path from the root to terminal x. If $f_{x,a} = 0$, no flow is sent through this arc to terminal x. The set of arcs leaving a vertex v is defined as $\delta^+(v) = \{(v, u)|u \in V\}$. Similarly, the set of arcs entering a vertex v is defined as $\delta^-(v) = \{(u, v)|u \in V\}$. The objective is to minimise the sum of the weights of the arcs that are used in the solution.

For each terminal $x \in X$, Constraint (10b) ensures that the root has exactly one outgoing arc with flow for x. Furthermore, it is ensured by constrains (10c) and (10d) that each terminal $x \in X$ has exactly one incoming arc carrying flow for itself and no outgoing arcs that carry flow for x. Note that, it is possible for a terminal to have outgoing arcs in the solution, if they carry flow for another terminal. A terminal is not always a leaf.

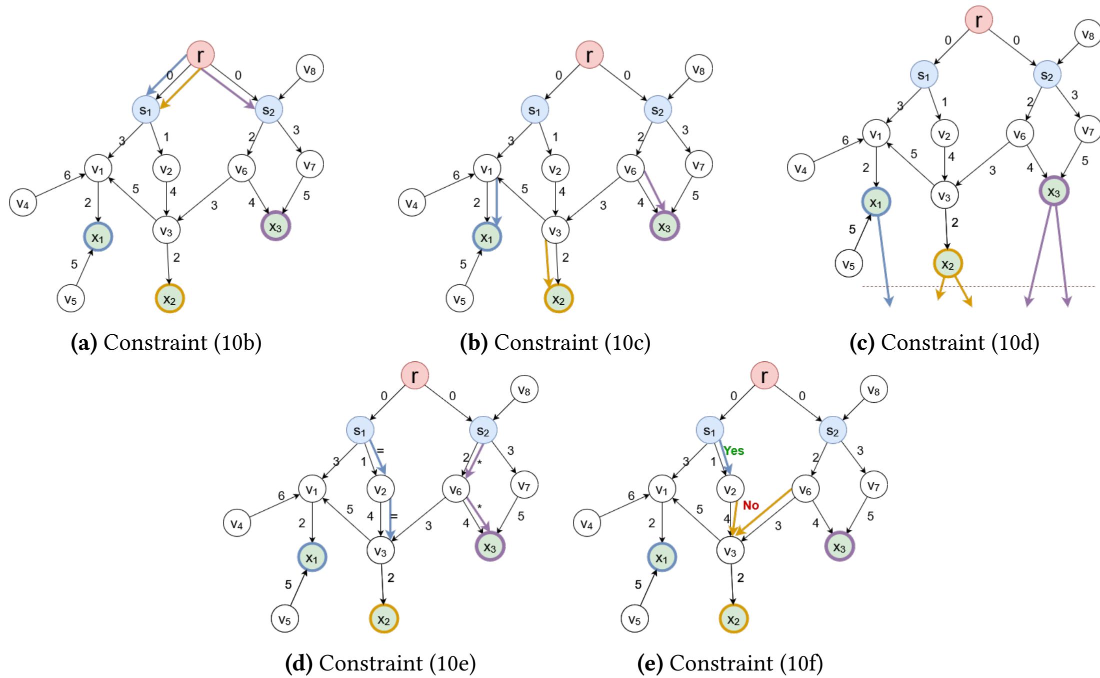
The previously mentioned constraints ensure, together with constraints (10e) and (10f), that there is exactly one path from the root to each terminal. Constraint (10e) ensures that the number of arcs coming into a vertex $q \in v \setminus \{\{r\} \cup \{x\}\}$ that carry flow for x is the same as the number of such outgoing arcs. This also holds for terminals; for any terminal x, the incoming flow for another terminal t is equal to the outgoing flow for t. Since there is exactly one arc leaving the root that carries flow for x, and at most one arc with flow for x can enter a vertex, any vertex $q \in v \setminus \{\{r\} \cup \{x\}\}$ that conducts flow for x can only have exactly one incoming and one outgoing arc that carries flow for x. Otherwise, such a vertex has no flow for x entering or leaving. This means that a path from the root to a terminal cannot contain cycles. With these constraints it is still possible for the result to contain a directed cycle that is disjoint from the

path. However, since the objective function minimises the sum of the arc weights, and the graph does not contain cycles of weight 0, this will never occur in the result.

Constraint (10g) ensures that $f_{x,a}$ can only be one if y_a is equal to one. This means that flow to a terminal x can only go through arc a if arc a is allowed to be used.

Constraint (10h) was added to prevent arcs of weight zero to be present in the result if they are not used to send flow. Let $a \in A$ be an arc with weight zero. For the objective, it does not matter if this arc is added or not. No other constraint prevents y_a from being equal to one, even if $f_{x,a}$ is zero for all terminals s. Note that the problem would not occur if the arc weights were strictly positive.

Figure 16 shows a directed graph with two start vertices and three terminals, where the constraints are visualised. Figure 16a shows that there is exactly one arc with flow leaving the root for x_1, x_2 and x_3. In Figure 16b we see that, for each terminal, exactly one arc with flow for that terminal enters the terminal. Figure 16c shows that no flow for a terminal can leave the terminal. We see that the flow for a terminal entering a regular vertex is the same as the flow leaving it in Figure 16d. Finally, Figure 16e shows that a vertex can have at most one incoming arc with flow for a certain terminal.



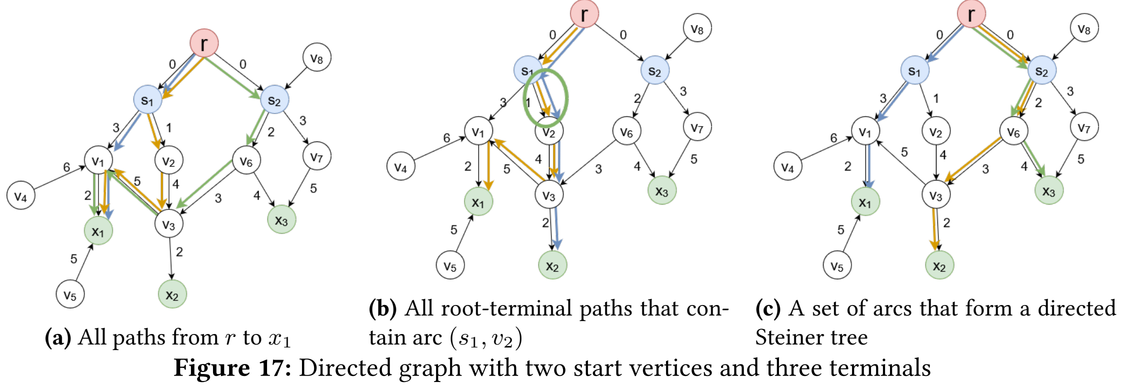
(a) Constraint (10b) **(b)** Constraint (10c) **(c)** Constraint (10d)

(d) Constraint (10e) **(e)** Constraint (10f)

Figure 16: Directed graph with two start vertices and three terminals

6.3 Path based formulation

Finally, we introduce an ILP formulation that is path based. We first define the set $P = \{p_{r,x} | x \in X\}$, containing all paths $p_{r,x}$ from the root to a terminal. Furthermore, we define, for each terminal $x \in X$, the set $P_x = \{p_{r,x}\}$. This is a subset of P that contains all paths $p_{r,x}$ from the root to terminal x. This is visualised in Figure 17, which shows a directed graph with two start vertices s_1, s_2 and three terminals x_1, x_2, x_3.



(a) All paths from r to x_1

(b) All root-terminal paths that contain arc (s_1, v_2)

(c) A set of arcs that form a directed Steiner tree

Figure 17: Directed graph with two start vertices and three terminals

Figure 17a shows all paths from the root to terminal x_1. Next, we define $P_a = \{p_{r,x} | p_{r,x} \text{ contains arc } a\}$ to be the set of paths that contain arc a, for each $a \in A$. Figure 17b shows all paths from r to a terminal that contain arc (s_1, v_2). The goal is to find a set of arcs of minimum weight that defines paths from the root to each terminal, as shown in Figure 17c.

$$\min \sum_{a \in A} y_a w(a) \tag{11a}$$

$$\text{s.t.} \sum_{p \in P_x} z_p \geq 1 \qquad \forall x \in X \tag{11b}$$

$$\sum_{p \in P_a} z_p \leq M y_a \qquad \forall a \in A \tag{11c}$$

$$y_a \in \{0, 1\} \qquad \forall a \in A \tag{11d}$$

$$z_p \in \{0, 1\} \qquad \forall p \in P. \tag{11e}$$

Equivalently to the previously discussed IP formulations, 11 has a binary decision variable y_a for each arc $a \in A$, which decides whether arc a is contained in the solution. Furthermore, there is a binary decision variable z_p for each path $p \in P$. This variable decides whether a path p is present in the solution. The objective is to minimise the sum of the weights of the arcs that are present in the solution.

Constraint (11b) ensures that, for each terminal, there is at least one path from the root to that terminal. Note that there may be more than one path, since a terminal might be on the path to another terminal too.

Furthermore, we introduce big-M Constraint (11c) to guarantee that, if an arc is not contained in the solution, no paths that contain this arc are present in the solution. Here, M is a number larger than the left-hand side will ever be, such that, if $y_a = 1$, the number of paths that pass through arc a is not bounded by M. Therefore, we choose M to be equal to m, since an optimal solution will contain exactly one path for each terminal, and therefore no more than m paths that are used in the solution will contain arc a. This constraint is introduced for each arc $a \in A$.

Note that, since we have assumed that our graph does not contain zero-weight cycles, the solution will never contain a cycle. This is because a cycle will never give an optimal solution.

Finally, we note that we can relax Constraint (11e) to $z_p \in \mathbb{R}_{[0,1]}$, and z_p will still be integral in the solution.

Theorem 8. *If Constraint* (11e) *is relaxed to* $z_p \in [0,1]$ *for all* $p \in P$*, the optimal solution for* (11) *will contain all binary variables* z_p *for* $p \in P$*.*

Proof. Let a solution $(\overline{y}, \overline{z})$ be given, where $\overline{z}_{p_1}$ is not integral for some $p_1 \in P$, i.e. $\overline{z}_{p_1} \in (0,1)$. We will prove that this is not an optimal solution.

Define A_p to be the set of arcs that path p_1 consists of. Let x be the terminal that path p_1 goes to. Because of Constraint (11b), there must be at least one other path p_2 that also leads to x. Then, because of Constraint (11c), for all arcs $a \in A_p \cup A_{p_2}$, the variable $\overline{y}_a$ will be one.

Let another solution $(\widetilde{y}, \widetilde{z})$ be given, where $\overline{y}_a = \widetilde{y}_a$ for $a \notin A_{p_1} \cup A_{p_2}$, and $\overline{z}_p = \widetilde{z}_p$ for $p \neq p_1$ and $p \neq p_2$. Without loss of generality, assume $\widetilde{z}_{p_1} = 1$. Then, $\widetilde{z}_{p_2}$ does not need to be one, since Constraint (11b) is already satisfied. Now, Constraint (11c) forces $\widetilde{y}_a$ to be one for arcs $a \in A_{p_1}$, but not for arcs $a \in A_{p_2}$. Therefore, $\widetilde{y}_a = 0$ for $a \in A_{p_2}$, unless a is used in a path to another terminal. If not all arcs $a \in A_{p_2}$ are contained in a path to another terminal, we know $\sum_{a \in A} \overline{y}_a w(a) > \sum_{a \in A} \widetilde{y}_a w(a)$, and thus $(\overline{y}, \overline{z})$ is not optimal.

Now assume all arcs $a \in A_{p_2}$ are contained in a path p_3 to another terminal x' in solution $(\widetilde{y}, \widetilde{z})$. Then, $\widetilde{y}_a = 1$ for all arcs that are contained in path p_3. We construct a third solution $(\hat{y}, \hat{z})$, where $\widetilde{y}_a = \hat{y}_a$ for $a \notin A_{p_1} \cup A_{p_2} \cup A_{p_3}$, and $\widetilde{z}_p = \hat{z}_p$ for $p \neq p_1$, $p \neq p_2$ and $p \neq p_3$. We let $\hat{z}_{p_3} = 0$, and let $\hat{z}_{p_4} = 1$ for a path p_4 which uses the same arcs as p_3 from vertex x to x', but uses the arcs of p_1 from the root to vertex x. Now, in solution $(\hat{y}, \hat{z})$, not all arcs $a \in A_{p_2}$ are contained in a path to another vertex. Therefore, $\sum_{a \in A} \overline{y}_a w(a) \geq \sum_{a \in A} \widetilde{y}_a w(a) > \sum_{a \in A} \hat{y}_a w(a)$.

We can conclude that an optimal solution to problem (11) will never contain non-integral variables z_p. □

7 Testing the algorithms

For the algorithms described in the previous sections, we have analysed the worst case time complexity, and, for the approximation algorithms, the worst case approximation ratio. However, the average performance for the problem provided by ASML might be a lot better. Therefore, we will test the algorithms on ASML's data and compare the computational results of the different algorithms. The code that was used to test this can be found at `https://github.com/sasvdh/DST.git` (git hash 7d8186e4bfab244e2dfb3df29a3eaca0a1a34df7). In Section 7.1.1 we will discuss the data provided by ASML, and in Section 7.1 we discuss the computational results for the corresponding graph.

Since the data provided by ASML is limited, and they might extend the graph in the future, we will discuss how we can produce more graphs to test the algorithms on. These graphs must have similar properties as the graph we can extract from ASML's data. This is discussed further in Section 7.2.1. Then, the computational results for these graphs are discussed in Section 7.2.2.

7.1 ASML graph

We first discuss the data provided by ASML and how we preprocess the data. Then, we look at how the algorithms discussed in the previous sections perform on ASML's data.

7.1.1 Input data

The context model provided to ASML can be converted to a directed graph, which results in a graph containing 53 vertices and 91 arcs. This graph does not contain any self-loops, but it does contain parallel arcs. After applying the pre-processing that was described in Section 3.2.1, we obtain a graph containing $n = 54$ vertices and $q = 88$ arcs.

It is currently not known yet what the weights of the arcs are going to be. As discussed in Section 3.1, the weight of an arc will be associated to the amount of time it takes to use the corresponding relation in the context model. However, at this moment, these values are not yet known. Therefore, we will test the algorithms with randomly generated arc weights from the range $(0, 1)$. Furthermore, we will test if the algorithms perform differently if the graph is unweighted; i.e. if all arc weights are 1.

The performance of the algorithms does not only depend on the input graph, but also on the start vertices and terminals that are given as input. To analyse how the choice of start vertices and terminals impacts the computational results of the algorithms, we will analyse how each algorithm performs for various numbers of start vertices and terminals. That is, we will randomly compute sets of start vertices and sets of terminals, each of a given size, for various sizes. To do this, we first choose how many start vertices and terminals we want to have, and call these numbers σ and τ, respectively. If $\sigma > n$ or $\tau > n$, we immediately know that it is not possible to compute a feasible combination of start vertices and terminals, and we terminate. Otherwise, we compute for each vertex $v \in V$ the set of vertices that can be reached from v, and combine these into a set R. If $|R| < \tau$, we terminate, since we will not be able to find a feasible set of τ terminals for any set of start vertices. Otherwise, we randomly choose σ vertices from V and let these be the start vertices S. Now, we compute, for each start vertex $s \in S$, the set of vertices that can be reached from s and combine all these vertices in a set Q. From this set we randomly choose τ terminals to form the set X. If $|Q| < \tau$, it is not possible to choose the number of terminals we want for the given set of start vertices. In this case, we compute a new set of start vertices and try again. This is repeated until we have found a valid combination of start vertices and terminals.

The context model that ASML has currently built contains a part of the classes to be considered. However, it could be extended further. Therefore, we will also compute random graphs of various other sizes to simulate how the discussed algorithms would perform on ASML's data if ASML extends their context model. How this is done, is explained in Section 7.2.1.

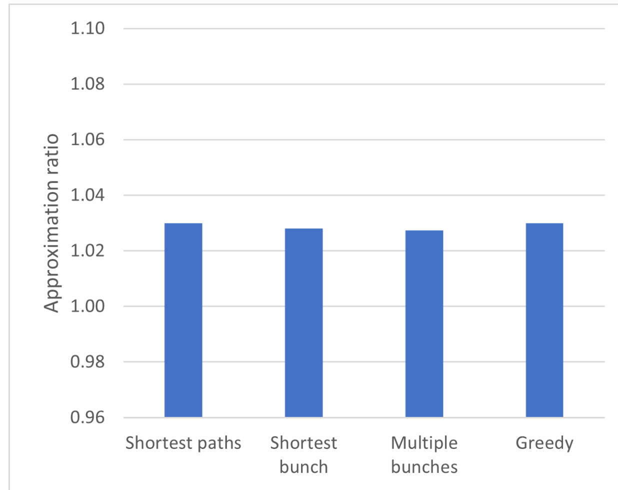
7.1.2 Computational results: random arc weights

First, we look at the approximation ratios of the approximation algorithms for different start vertices and terminals. We varied the number of start vertices and terminals $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, 20, ..., 45, 50\}$, and for each σ and τ we computed 100 sets of start vertices and terminals randomly. Aside from running the approximation algorithms, we also computed the optimal solution using one of the LP based methods. This way we computed the approximation ratio for each run of each algorithm.

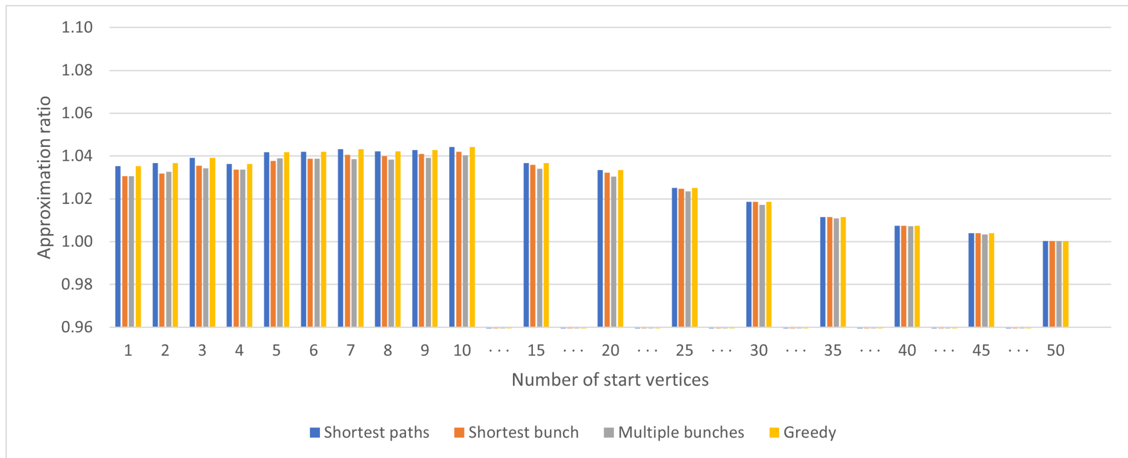
Figure 18 shows the average approximation ratios takes over these 100 randomly generated sets for each $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, 20, ..., 45, 50\}$. We refer to the shortestPathsToTerminals Algorithm as "Shortest paths", the shortestBunch Algorithm as "Shortest bunch", the multipleBunches Algorithm as "Multiple bunches", and the greedyrArborescence Algorithm as "Greedy". In Figure 18a we see the average taken over all these values of σ and τ. We see that the approximation ratios for each of the algorithms are all around 1.04. If you look very closely, you see that the multipleBunches Algorithm performs best, but the difference is very small. We have seen that the shortestPathsToTerminals, shortestBunch and multipleBunches Algorithms each had an approximation factor of m, and the greedyrArborescence Algorithm $(n - 1)$. For the ASML graph, these algorithms all perform a lot better than these upper bounds.

In Figure 18b we see the approximation ratios where the average is takes over all numbers of terminals τ. We see the ratios going down for large numbers of start vertices. This makes a lot of sense, since if many vertices in a graph are start vertices, a terminal is always close to a start vertex. This means there are not many different paths one can take from a start vertex to this terminal, and the algorithms will usually provide a good solution. Also, many of these start vertices will also be terminals, in which case the path from the start vertex to the terminal has length 0. Therefore, it is logical that for $\sigma = 50$ the average approximation factor for all approximation algorithms is very close to 1; 1.00071 to be precise. For $\sigma \in \{1, 2, ..., 10\}$, we see that the approximation ratio first goes up, stays stable, and only starts to decrease after $\sigma = 10$. This is the case for all approximation algorithms. This could be explained by the fact that, if we have only few start vertices, we have less degrees of freedom for choosing the paths to the terminals. If there is only one start vertex, there is only one arc going from the root to a start vertex. This means all paths to the terminals pass through this vertex. Another explanation is that for $\sigma = 1, 2, 3, 4$, less than 50 vertices can be reached. This means that for these numbers of start vertices, the average is only taken over numbers of terminals up to 45. As we can see in Figure 18c, the approximation ratio is worst for large numbers of terminals. Therefore, not taking into account the largest number of terminals for which we test, gives these values of σ an advantage. We see that indeed, if we do not take into account $\tau = 50$ for all values of σ, the chart flattens out a bit. However, we still see the approximation ratios rise from $\sigma = 1$ to $\sigma = 5$, and fall after $n = 10$.

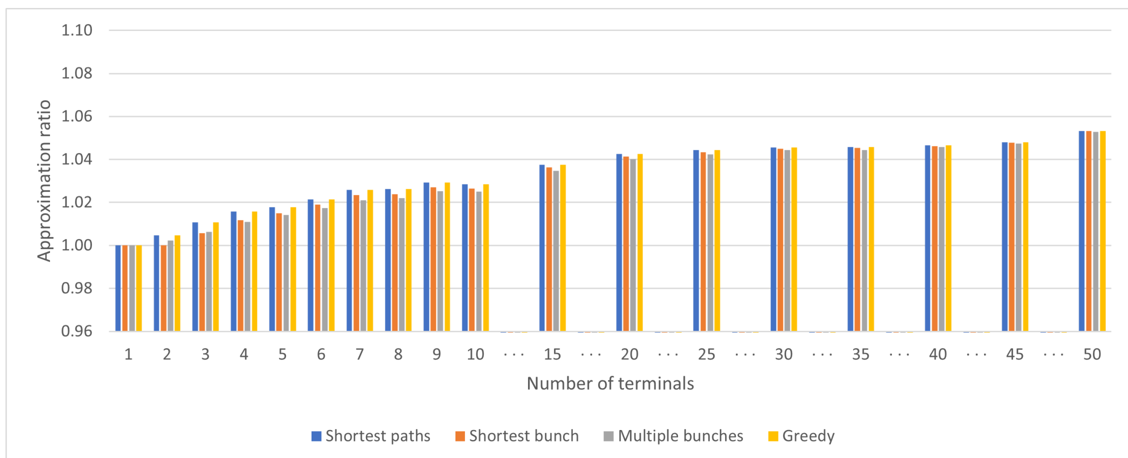
Figure 18c shows the approximation ratios where the average is taken over the number of start vertices σ. We see that the approximation ratio is smallest for a small number terminals, and greatest for a large number of terminals. If there are few terminals, the optimal solution will only contain few r-terminals paths. This means, there is a lower probability of paths overlapping, and therefore the optimal solution will often not be much better than the solutions returned by the approximation algorithms. This explains the behaviour we see in 18c.



(a) Average over all σ and τ



(b) Average over all τ



(c) Average over all σ

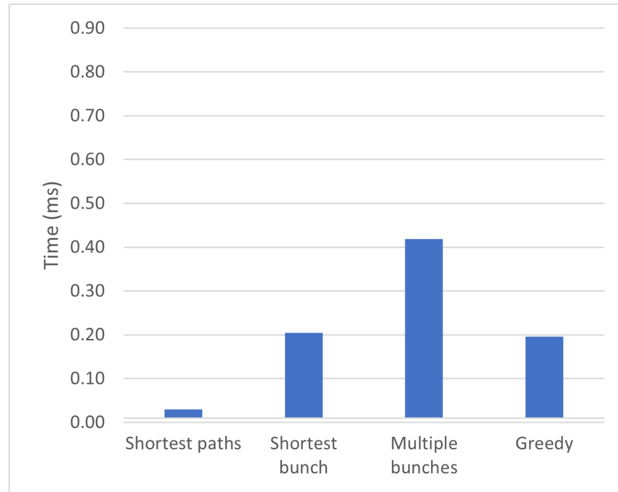
Figure 18: Approximation ratios of the approximation algorithms for various numbers of start vertices and terminals $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, ..., 50\}$. For each number of start vertices and terminals, we took the average of the results for 100 randomly generated sets of start vertices and terminals.

Now, we will analyse the run times of the different algorithms. First of all, we analyse the run times of the approximation algorithms. This is based on numbers of start vertices and number of terminals $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, 20, ..., 45, 50\}$, and 100 randomly generated sets of start vertices and terminals for each pair (σ, τ). For each pair (σ, τ), we take the average of the results for the 100 sets of start vertices and terminals.

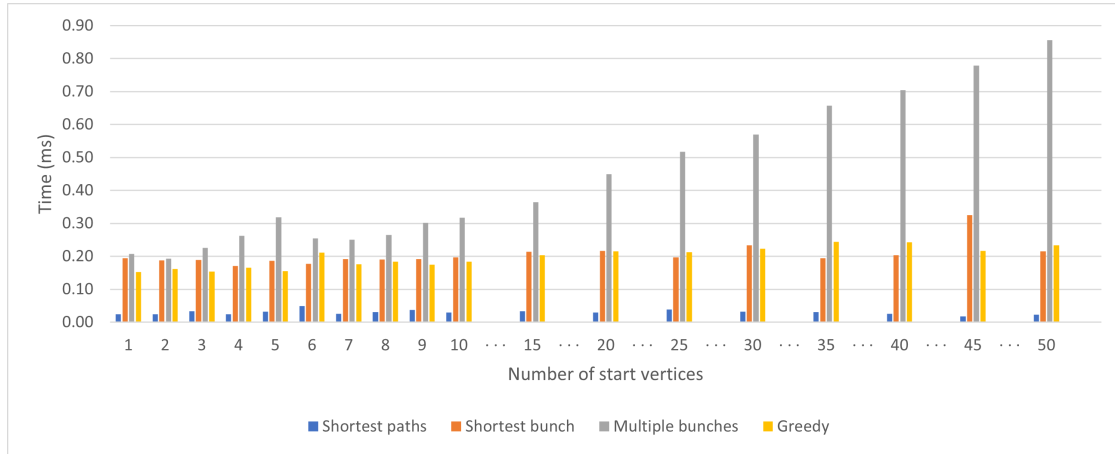
Figure 19a shows the average run time per algorithm, where the average is taken over all σ and τ. We have already seen in the complexity analysis of each algorithm that the shortestPathsToTerminals is fastest, followed by the greedyrArborescence Algorithm, the shortestBunch Algorithm and finally the multipleBunches Algorithm. The same is visible in Figure 19a.

We already saw in the complexity analysis of each algorithm that the only algorithm that is dependent on the number of start vertices is the multipleBunches Algorithm. We see this in Figure 19b too, where we see the this is the only algorithm of which the run time increases as the number of start vertices increases. For the other algorithms, there is some variation in the run time as the number of start vertices increases, but we see no pattern. This is most likely due to run time variability and the fact that this graph and these start vertices and terminals are still a relatively small sample. For the multipleBunches Algorithm, we know that the complexity is $O(n^2 \log(n) + mp + mn(q + n))$. The dependence on p can be seen in Figure 19b; where the run time increases as the number of start vertices gets larger. For a small number of start vertices we do not see a dependence on the number of start vertices yet. This can be explained by the fact that the term containing p will still be small compared to the terms containing n or q in the complexity estimation.

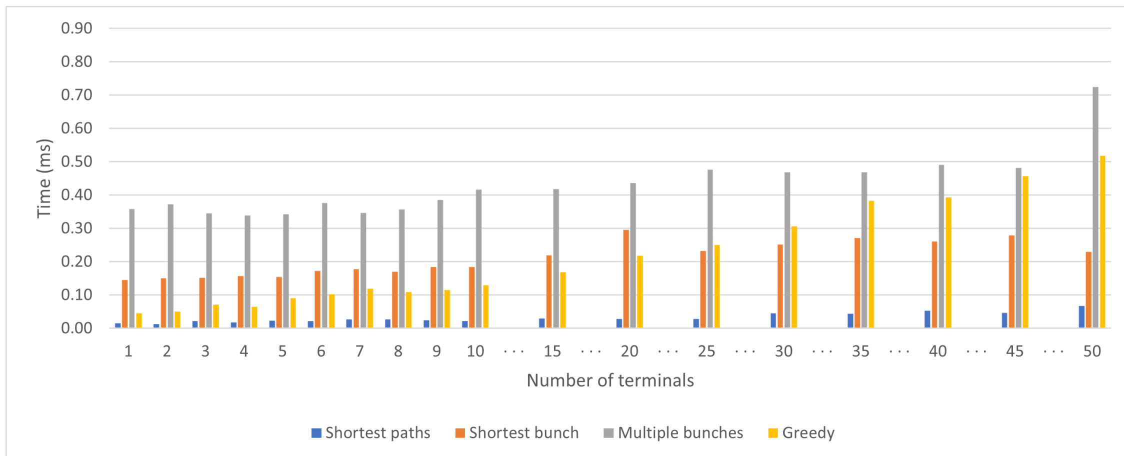
In Figure 19c we see the run times for the approximation algorithms, where the average is taken over all numbers of start vertices. We know from the complexity analyses that the run times of all algorithms are dependent on the number of terminals. We see this in Figure 19c, where the run time of each algorithm increases as the number of terminals increases. For the shortestBunch Algorithm, the run time seems to decrease again at $\tau = 50$. However, this can be due to run time variability, since the difference is rather small.



(a) Average over all σ and τ



(b) Average over all τ



(c) Average over all σ

Figure 19: Run times of the approximation algorithms for various numbers of start vertices and terminals $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, ..., 50\}$. For each number of start vertices and terminals, we took the average of the results for 100 randomly generated sets of start vertices and terminals.

The run time of any algorithm will differ every time one runs it, since many factors related to the system on which an algorithm is run can impact the run time. Therefore, to be able to draw strong conclusions about the run time of an algorithm, it should be tested many times. However, since some of the exact algorithms take very long to execute, it was not possible to do this. Therefore, we run the exact algorithms once for each $\sigma \in \{1, 2, 3, 4, 5, 10, 15, 20, 25\}$ and $\tau \in \{5, 10, 15, 20, 25\}$. We will discuss the results of these tests, keeping in mind that we may not fully rely on these results.

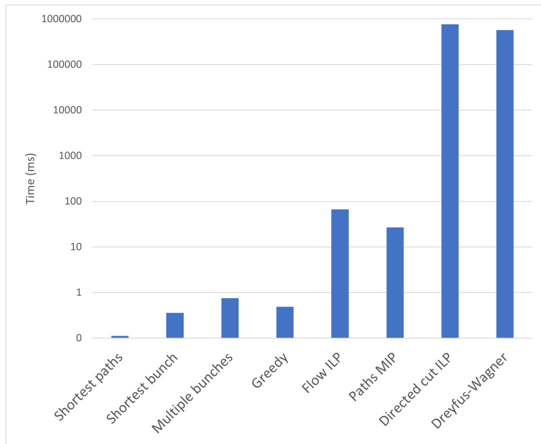
Since the DirectedDreyfusWagner Algorithm and the directed cut approach take very long to complete, there is a time limit of 15 minutes on these methods. For the directed cut approach, we found that generating all constraints beforehand already leads to a memory error for the ASML graph for 1 terminal. Therefore, we generated the constraints on the fly by computing the most violated constraints. Figure 20 shows the results of the runs we did for all exact algorithms.

In Figure 20a we see the average run times of the algorithms discussed in this thesis, where the average is taken over $\sigma \in \{1, 2, 3, 4, 5, 10, 15, 20, 25\}$, $\tau \in \{5, 10, 15, 20, 25\}$, and the results are based on one run. To be able to show all run times in one plot, the y-axis is logarithmic. Where the approximation algorithms all run, on average, in less than a millisecond, the exact algorithms run much slower. On average, the flow based and path based LP approaches solve the problem in 67 and 27 milliseconds, respectively. The directed cut approach and DirectedDreyfusWagner Algorithm both perform much worse; as they both usually reached the time limit of 15 minutes. In Figure 20b we can take a closer look at the run times of the exact methods. The directed cut approach and DirectedDreyfusWagner Algorithm both run off the chart, as they have an average run time of $7.6 \cdot 10^5$ and $5.7 \cdot 10^5$ milliseconds, respectively. This is taking into account the time limit of $9.0 \cdot 10^5$, in which case the algorithm did not provide a solution. The true average run time will therefore likely be much larger.

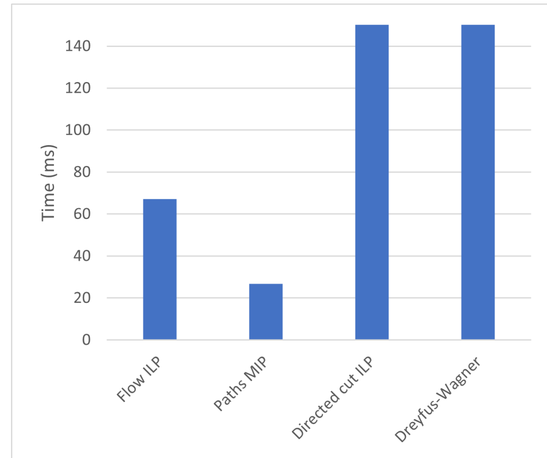
In Figure 20c, we see the run times where the average is taken over all numbers of terminals. We see that, again, the directed cut approach and DirectedDreyfusWagner Algorithm both run off the chart. Both the flow based and path based LP run faster if there are more start vertices. More start vertices will mean that there are more paths possible from the root to the terminals. This makes it easier for the solver to find a feasible solution, and the solver can converge to an optimal solution more quickly. This appears to be beneficial for both LP formulations.

Finally, we look at the average run time for various numbers of terminals, as shown in Figure 20d. We see that the run times of the flow based and path based LP both seem to increase as the number of terminals increases. This can be explained by the fact that both LP formulations will contain more constraints and variables, making the problem harder to solve. It is notable that the flow based LP seems to be solved especially fast when there are 10 terminals. There does not seem to be an immediate explanation for this, and it is most likely caused by the fact that this average is based on a very small sample size.

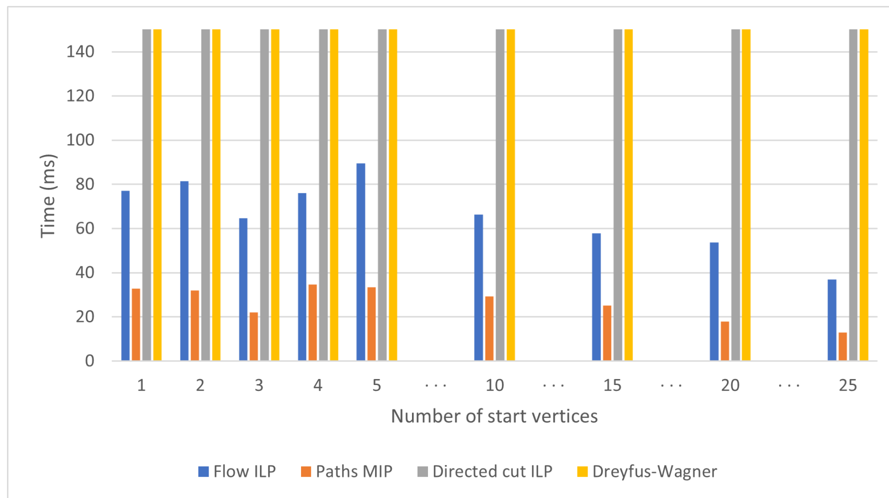
The run time of the DirectedDreyfusWagner Algorithm is not visible in Figure 20d since it is much longer than that of the flow and bath based LP methods. However, when looking at the run times of the various runs for the DirectedDreyfusWagner Algorithm, we note that the run time of this algorithm is heavily dependent on the number of terminals. We have already seen that the time complexity of the Directed-DreyfusWagner Algorithm is exponential in the number of terminals. The algorithm is able to complete within the time limit of 15 minutes for all instances of $\tau = 5$ and $\tau = 10$, but not for any larger values of τ. For $\tau = 5$, the average run time is 171ms. For, $\tau = 10$, the average run time is $1.2 \cdot 10^5$ms. This shows how quickly the run time of the DirectedDreyfusWagner Algorithm blows up. We also see this when we run the algorithm some more times for smaller numbers of terminals. Figure 21 shows the run time of the DirectedDreyfusWagner Algorithm for various different numbers of terminals. Here, all results are based on sets of start vertices of size 5, which were randomly generated at the start of each run. For $\tau = 2$, the run time of the algorithm is 150ms, which is already much slower than the average run times of the flow based an path based LP approaches that were mentioned above. We see that the run time indeed increases exponentially as the number of terminals increases.



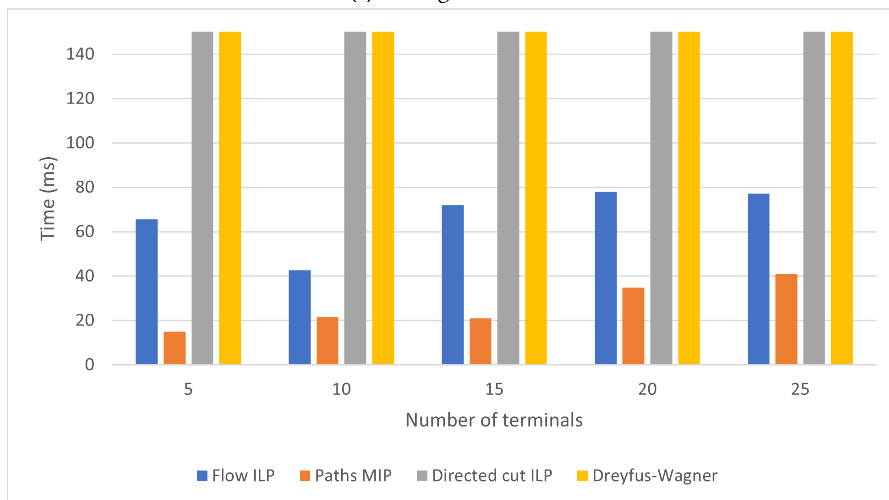
(a) Average over all σ and τ; for all algorithms, with a logarithmic scale



(b) Average over all σ and τ; for all exact approaches



(c) Average over all τ



(d) Average over all σ

Figure 20: Run times of the exact approaches for numbers of start vertices $\sigma \in \{1, 2, 3, 4, 5, 10, 15, 20, 25\}$ and numbers of terminals $\tau \in \{5, 10, 15, 20, 25\}$. For each number of start vertices and each number of terminals, we generated 1 set of start vertices and terminals.

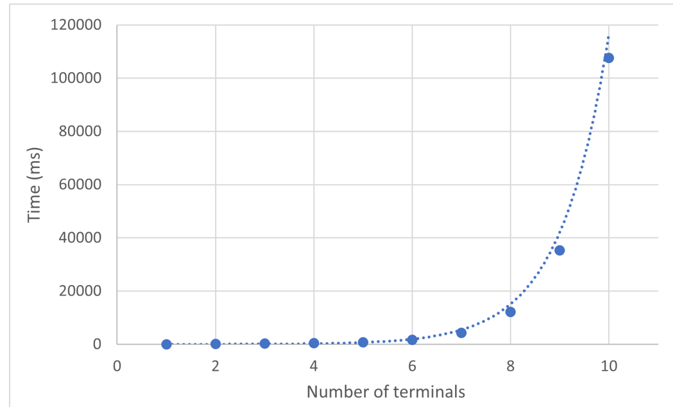


Figure 21: Run times for the DirectedDreyfusWagner Algorithm for $\sigma = 5$, with a different randomly generated set of start vertices for each value of τ.

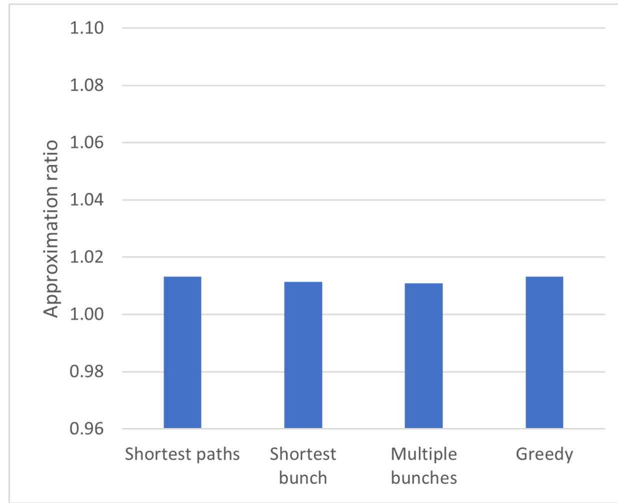
7.1.3 Computational results: unweighted graph

So far, we have looked at a graph with randomly generated weights, since it is still unknown what the weights of the graph are going to be exactly. However, it can be interesting to analyse the performance of the algorithm for an input graph where all arc weights are one. With the exception of the zero weight arcs that are added during pre-processing, all arcs of the graph have the same weight. Since the zero weight arcs do not change any solution, they can be disregarded. Therefore, the algorithms will behave the same on this graph as on an unweighted graph. For simplicity, we will therefore refer to this graph as the unweighted graph.

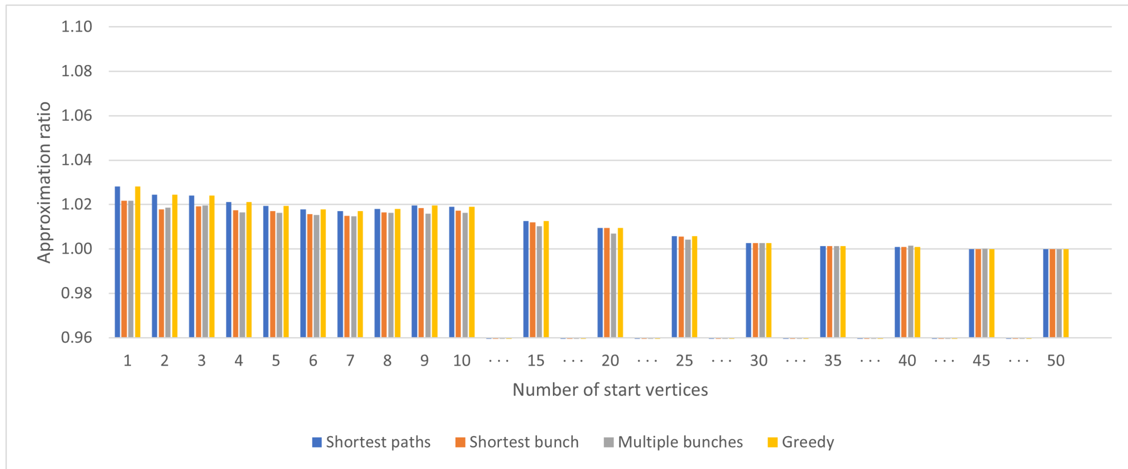
For the approximation ratio, we run the algorithms for the same input as for the weighted case. If we then take the average over all σ and τ, we get Figure 22a. We see that, as we have also seen for the weighted graph, the approximation algorithms all have a similar approximation ratio. However, we do see that these ratios are smaller than those we have seen for the weighted graph. For the weighted graph the approximation ratios were around 1.03, whereas for the unweighted graph they are around 1.01. For unweighted graphs, the problem reduces to finding a subgraph of D that contains paths from r to each terminal and contains a minimum number of arcs. This is a problem with fewer degrees of freedom; which means there is a higher probability that the approximation algorithm computes paths that are similar to the paths of an optimal solution. Therefore, this gives a better approximation ratio.

Figure 22b shows the approximation ratios for the various numbers of start vertices. We see similar behaviour as for the case with random weights. However, the approximation ratio starts out smaller, and goes down faster for the unweighted graph. As explained, the problem concerning the unweighted graph has fewer degrees of freedom. Therefore, increasing the number of start vertices in an unweighted graph reduces the difficulty of the problem quicker than in a graph with random arc weights.

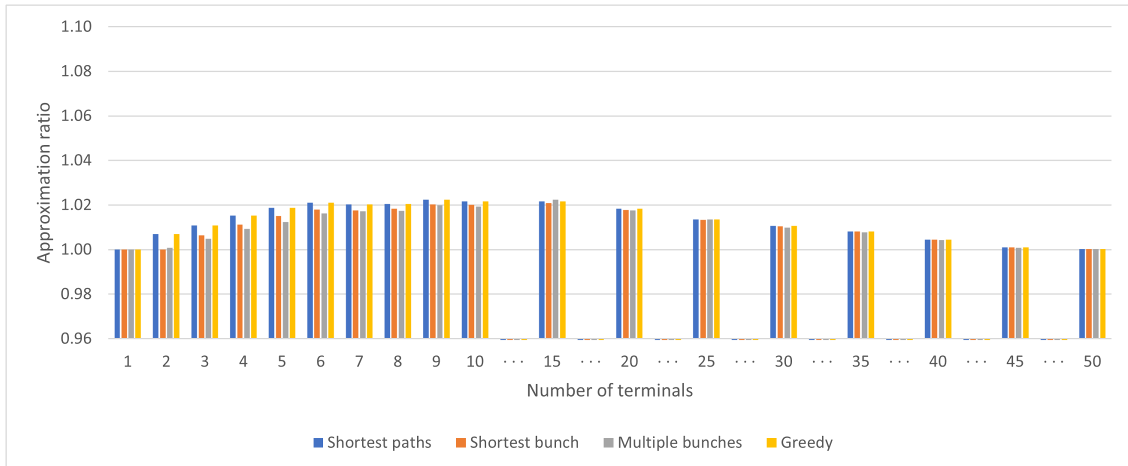
We also look at the approximation ratio for various numbers of terminals, shown in Figure 22c. If we compare this to the case with random weights, we notice two differences. First of all, for the unweighted case, the approximation ratio stays lower than for the graph with random arcs weights. Furthermore, we see that the approximation ratio goes down as the number of terminals increases, whereas for the weighted graph, the approximation ratio went up. The reason why the approximation algorithms can be far off from an optimal solution is that the approximation algorithms look at the shortest path to get somewhere. However, it can often be more efficient to choose a slightly longer path so that the path overlaps with a path to another terminal. For weighted graphs, more terminals make the problem more complicated and approximation algorithms will produce more paths in a “dumb” way. For unweighted graphs, the advantage from choosing overlapping paths is not as big, since all arcs are equally expensive. In fact, if there are more terminals, the solution will contain more arcs and it will be more likely that paths overlap. Therefore, the approximation algorithms perform better on unweighted graphs if there are more terminals.



(a) Average over all σ and τ



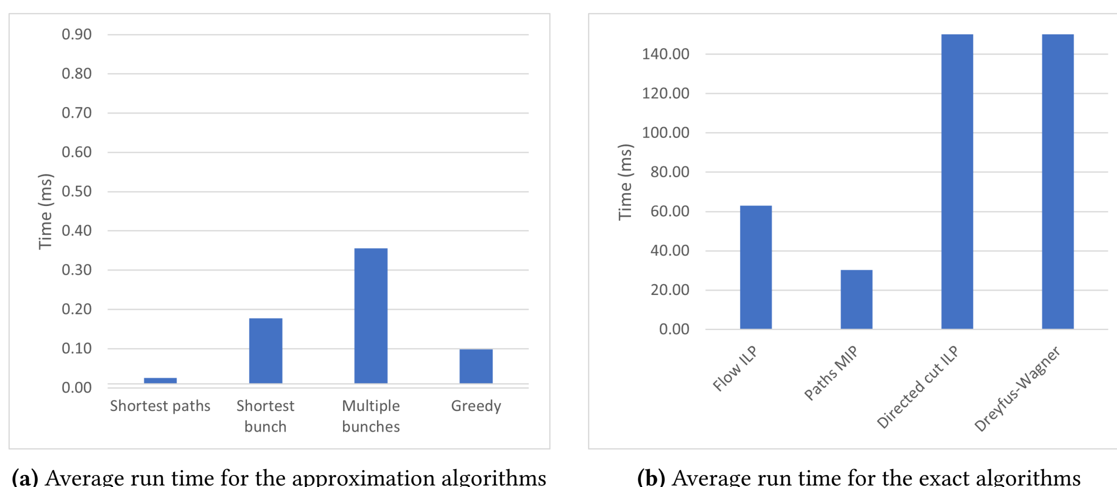
(b) Average over all τ



(c) Average over all σ

Figure 22: Approximation ratios of the approximation algorithms for the unweighted graph for various numbers of start vertices and terminals $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, ..., 50\}$. For each number of start vertices and terminals, we took the average of the results for 100 randomly generated sets of start vertices and terminals.

Figure 23 shows the average run times for the algorithms for an unweighted graph. We ran the approximation algorithms for 100 randomly generated sets of start vertices and terminals, for each pair (σ, τ), where $\sigma, \tau \in \{1, 2, ..., 8, 9, 10, 15, 20, ..., 45, 50\}$. We see that the approximation algorithms run faster for the unweighted case than for the weighted case, but the proportions between the different algorithms are the same. All approximation algorithms perform Dijkstra's algorithm. On an unweighted graph, Dijkstra's algorithm reduces to a breadth-first traversal of the graph. This has a better time complexity than Dijkstra's algorithm on a graph with varying arc weights. It is unclear whether the Java implementation of Dijkstra's algorithm that we used is faster for this reason. However, if this is the case, this could be the reason that the approximation algorithms complete faster on unweighted graphs. Since the exact algorithms take much longer to complete, we ran these once for each (σ, τ) pair with $\sigma, \tau \in \{1, 2, 3, 45, 10, 15, 20\}$. The average run times of these algorithms are very similar to how they were for the weighted graph, and therefore the weights of the arcs do not seem to impact these algorithms.



(a) Average run time for the approximation algorithms

(b) Average run time for the exact algorithms

Figure 23: Average run time for all algorithms for the unweighted graph. The averages for the approximation algorithms are based on values of σ and τ taken from $\{1, 2, ..., 8, 9, 10, 15, 20, ..., 45, 50\}$, where we generated 100 sets of start vertices and terminals randomly for each (σ, τ) pair. The averages for the exact algorithms are based on values of σ in $\{1, 2, 3, 4, 5, 10, 15, 20\}$ and τ in $\{5, 10, 15, 20, 25\}$.

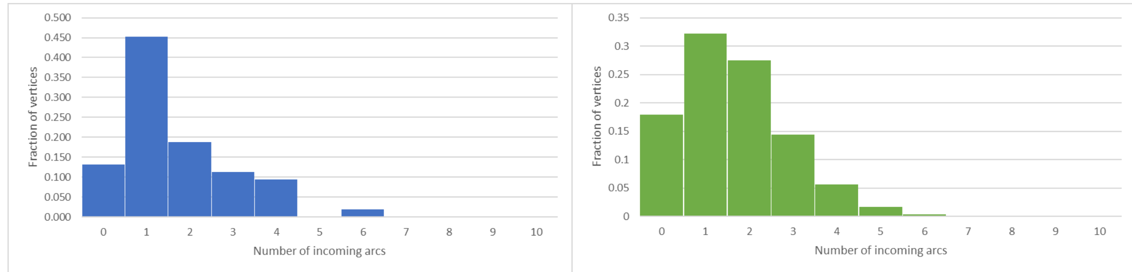
7.2 Randomly generated graphs

Before we look at how the algorithms perform on randomly generated graphs, we discuss how we can compute random graphs that have properties that are similar to the properties of the ASML graph.

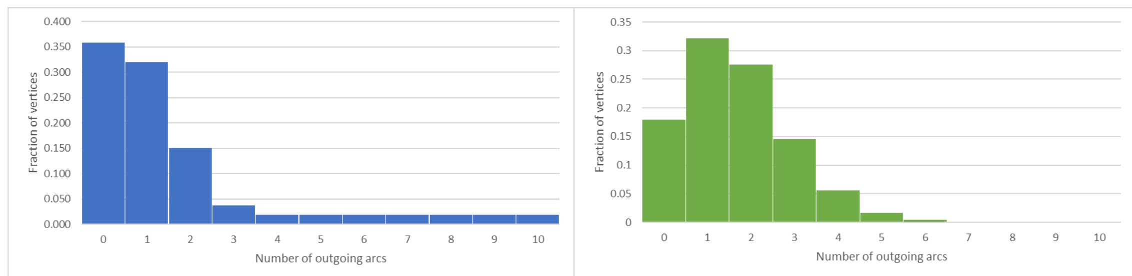
7.2.1 Computing random graphs

To compute random graphs that model ASML's data right, we let the random graphs have the same vertex-arc ratio as the graph obtained from ASML's context model. This graph has a ratio of $53/88$. Therefore, if we construct a graph with n vertices, we let this graph have $q = \lceil n\frac{88}{53} \rceil$ arcs. We construct a graph that does not contain parallel arcs or self-loops, since these would need to be removed anyway. The random graphs are generated using the Erdös-Rényi $G(n, M)$ model, which chooses a graph randomly from the collection of all graphs with n vertices and M arcs [Paul and Alfréd(1959)].

To see if these random graphs are a good model for the ASML graph, we analyse the degree distributions. We look at the indegree (number of incoming arcs) and outdegree (number of outgoing arcs) of each vertex. Then, we calculate what fraction of vertices has a certain indegree or outdegree. Figure 24 shows the results of this. The blue bar charts show the results for the ASML graph, and the green charts show the results for the random graphs. For the random graphs, the result is the average of 1.000 randomly generated graphs of size 10,20,...,100. We see in Figure 24a that the ASML graph has many vertices with indegree 1, whereas the randomly generated graphs also have many vertices of indegree 2. In Figure 24b we see that for both the ASML graph and the random graphs, lower outdegrees are more common. However, for the ASML graph, there are relatively many more vertices with outdegree 0 or 1 than any other outdegrees. From this, we can expect that the ASML graph has fewer paths than the average randomly generated graph. We see that this is indeed true, since the ASML graph has a total of 1342 simple paths, whereas the random graphs with 50 vertices have an average of 10270 simple paths. This means that, even though the shapes of the degree distributions look quite similar, the algorithms might perform differently on the randomly generated graphs than on the ASML graph. Nonetheless, these random graphs can help give us some insight in how the algorithms perform on differently sized graphs.



(a) Indegree distribution for ASML graph (left) and randomly generated graphs (right)



(b) Outdegree distribution for ASML graph (left) and randomly generated graphs (right)

Figure 24: Degree distributions for the ASML graph and the average taken of 1.000 randomly generated graphs of sizes 10,20,...,100. The vertical axis shows the fraction of vertices that have a certain degree, the horizontal axis shows the degree.

7.2.2 Computational results

Now, we analyse the computational results for the randomly generated graphs. We let all of these graphs have random arc weights, taken from range $(0, 1)$. We start by looking at the approximation ratios of the approximation algorithms. We compute 10 random graphs of n vertices, for each $n = 10, 20, ..., 100$. Then, for each of these graphs we compute 50 sets of start vertices and terminals randomly for each pair (σ, τ), for $\sigma, \tau \in \{5, 10, ..., 50\}$.

In Figure 25, we see that the approximation ratio is best for small graphs. For a part, this can be explained by the values of σ and τ for which we test the algorithms. For example, for $n = 10$, we let $\sigma, \tau \in \{5, 10\}$. For both these values, a large part of the graph will be a start vertex. Since the graph is small, there are not many paths possible, and therefore there is a higher probability that the approximation algorithms return a good solution. In general, in a large graph there are more possible solutions, and therefore a higher probability for the approximation algorithms to choose paths that are not in an optimal solution. Therefore, large graphs will, in general, give worse approximation ratios.

Since the ASML graph has 53 vertices, we can compare it to the random graphs of 50 vertices. We see in Figure 25 that the average approximation ratio is worse for the random graphs of 50 vertices than for the ASML graph, which had an approximation ratio around 1.03 for each algorithm. As we had discussed before, the ASML graph contains few simple paths compared to the average random graph with a similar number of vertices and arcs. If we especially look at the average number of paths from the root to the terminals, we see that the ASML graph has an average of 397 paths, and the random graphs have 3370 paths on average. Here, for the ASML graph, we computed 100 sets of start vertices and terminals randomly for each pair (σ, τ) with $\sigma, \tau \in \{5, 10, ..., 50\}$. We computed 10 different random graphs, and for each of this graph we computed 50 sets of start vertices and terminals randomly for each pair (σ, τ) with $\sigma, \tau \in \{5, 10, ..., 50\}$. Because the approximation algorithms only have very few paths to consider for the ASML graph compared to the random graphs, there is a higher probability that they return paths that are contained in an optimal solution. Therefore, the algorithms are more likely to return a solution with a value close to the optimal solution.

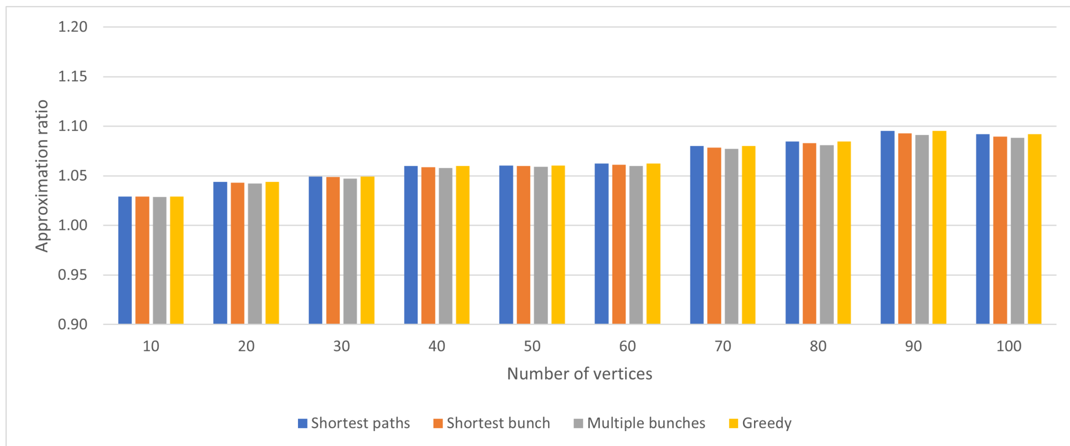


Figure 25: Approximation ratios of the approximation algorithms for random graphs of sizes $n \in \{10, 20, ..., 100\}$. Here, the average is taken over all numbers of start vertices and terminals $\sigma, \tau \in \{5, 10, ..., 50\}$. For each n, we randomly generated 10 graphs. For each of these graphs, we computed 50 sets of start vertices and terminals randomly.

For the approximation algorithms, we look at the average run times for the runs we have done for the approximation ratios. Figure 26a shows the average run times, where the average is taken over all $\sigma, \tau \in \{10, 20, ..., 100\}$, for 50 randomly generated sets of start vertices and terminals for each pair (σ, τ). We see, as expected, that the run time increases as the number of vertices in the graph increases. The shortestPathsToTerminals Algorithm is the fastest in all cases, followed by the greedyrArborescence

Algorithm, the shortestBunch Algorithm, and finally the multipleBunches Algorithm.

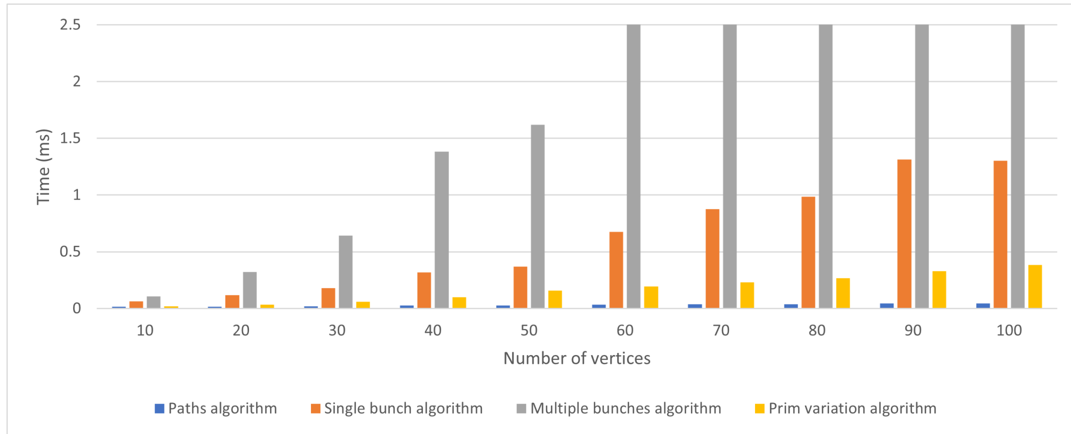
Now, to properly compare the random graphs to the ASML graph, we look at the average run times for graphs of $n = 50$ vertices, shown in Figure 26b, we see that the run times are higher for the random graphs than what we saw for the ASML graph, except for the greedyrArborescence Algorithm. This can again be explained by the structure of the graph. Since the random graphs have, on average, more paths, it will take longer to run Dijkstra's algorithm, since more vertices will be able to be reached from any source vertex. This means that Dijkstra's algorithm will have to explore many vertices, increasing its run time. Therefore, the algorithms that use Dijkstra's algorithm will run slower. The greedyrArborescence Algorithm is the only algorithm where Dijkstra's algorithm is not as significant for the time complexity, and is therefore not slower for the randomly generated graphs. The fact that the greedyrArborescence Algorithm is even faster than for the ASML graph, might be explained by the fact that the structure of the ASML graph is, for some reason, not as suited for this algorithm as the randomly generated graphs.

In Figures 26c and 26d we see the average run times for the exact algorithms. Since some of the exact algorithms take very long to execute, they were only run for one graph of each size $n \in \{10, 20, ..., 50\}$, and each pair (σ, τ) for $\sigma, \tau \in \{5, 10, 15, ..., 50\}$. As explained before, to be able to draw strong conclusions from computational results, one would have to run algorithms many times for various different instances. We keep this in mind when discussing the results.

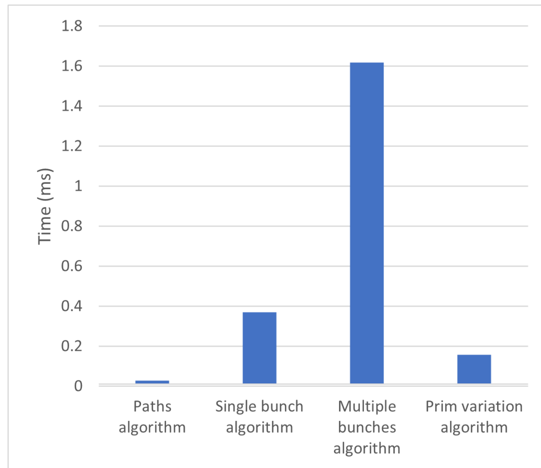
Figure 26c we see the average run times of the exact algorithms, where the average is taken over all n, σ and τ. We see that ratios between the different exact algorithms are similar to how they were for the ASML graph. Furthermore, note that the average run time is longer for the random graphs, despite the fact that this is the average taken over $n = 10, 20, ..., 50$, whereas the ASML graph has 53 vertices. The ASML graph is larger than any of the randomly generated graphs, and still the algorithms complete faster on the ASML graph. This shows that the structure of the graph plays a big role in the performance of the algorithms.

In Figure 26d, we see that, in general, the run time of the algorithms increases as the number of vertices in the graph increases. As expected, the run times of the directed cut LP approach and the DirectedDreyfusWagner Algorithm quickly increase as n increases. We see that for $n = 50$, the run times of the flow based and path based LP approaches are much higher than for the ASML graph, even though the graphs are comparable in size. However, since these algorithms are now only tested on one graph, this may be due to properties of this specific graph.

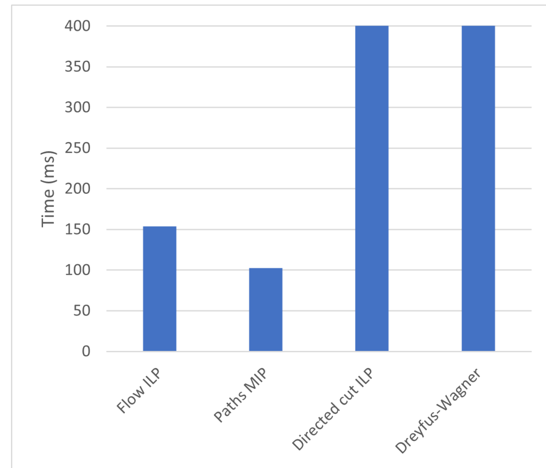
One thing that stands out is that for $n = 40$, the path based LP approach has a relatively very long run time. If we look at the graph that is generated for $n = 40$, we find the average number of paths from the root to the terminals in this graph is much higher than for the other values of n. In fact, there are, on average, four times more paths from the root to the terminals for $n = 40$ than for $n = 50$. This means the path based MIP has many variables, which makes it much harder to solve. This shows that, since the results are only based on one graph for each value of n, we cannot draw too strong conclusions about the results shown in this diagram. However, it does also show us the impact of the structure of the graph on the performance of the various LP methods.



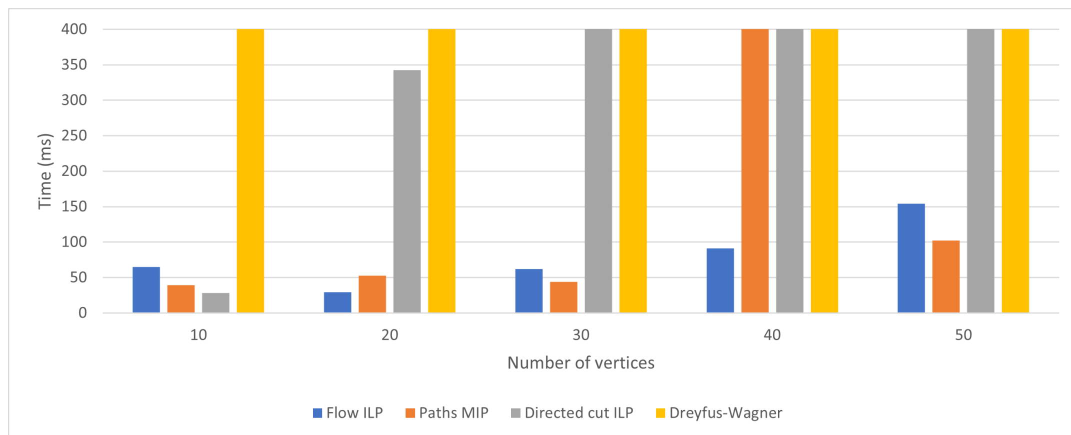
(a) Average run time for the approximation algorithms, where the average is taken over all $\sigma, \tau \in \{10, 20, \ldots, 100\}$



(b) Average run time for the approximation algorithms, where the average is taken over all $\sigma, \tau \in \{10, 20, \ldots, 100\}$, for $n = 50$



(c) Average run time for the exact algorithms, where the average is taken over all $\sigma, \tau \in \{10, 20, \ldots, 50\}$ and $n = 50$



(d) Average run time for the exact algorithms, where the average is taken over all $\sigma, \tau \in \{10, 20, \ldots, 50\}$

Figure 26: Run times of all algorithms for random graphs of various sizes n. For the approximation algorithms, we ran each algorithm for 10 random graphs of size $n \in \{10, 20, \ldots, 100\}$, and 50 sets of start vertices and terminals for each (σ, τ) pair with $\sigma, \tau \in \{10, 20, \ldots, 100\}$. For the exact algorithms, we ran each algorithm once for graphs of size $n \in \{10, 20, \ldots, 50\}$ and one randomly generated set of start vertices and terminals for each pair (σ, τ) with $\sigma, \tau \in \{10, 20, \ldots, 50\}$.

Finally, we run all algorithms once for a randomly generated graph of 1000 vertices, with 50 start vertices and 50 terminals. This indicates how each algorithm behaves on a larger graph. Of course, since this is only based on one run, we cannot draw strong conclusions from it. Nonetheless, it does provide some insight into how each algorithm performs on a large instance.

Table 1 shows the approximation ratios and run times of all approximation algorithms for a randomly generated graph with $n = 1000$, $\sigma = \tau = 50$. We see that these algorithms all terminate in less than a second. The shortestPathsToTerminals Algorithm terminates fastest, in 2 milliseconds. The algorithms all return a solution with the same weight, and therefore all have the same approximation ratio.

	Paths	Single bunch	Multiple bunches	Greedy
Approximation ratio	1.20	1.20	1.20	1.20
Run time	2ms	187ms	389ms	61ms

Table 1: Approximation ratio and run time for the approximation algorithm for a randomly generated graph with $n = 1000$, $\sigma = \tau = 50$

In Table 2 we see the results for the exact algorithms, which were tested on the same graph. The directed cut LP approach and the DirectedDreyfusWagner Algorithm both did not find an optimal solution in less than 6 hours, and were therefore terminated before they returned a solution. The path based LP approach gave a memory error, because the number of paths that had to be computed and stored was too large. The only exact algorithm that returned a solution in time is the flow based LP approach. This algorithm returned an optimal solution in 3 seconds.

	Flow ILP	Path MIP	Directed Cut ILP	Dreyfus-Wagner
Approximation ratio	1	∞	∞	∞
Run time	3.0s	∞	$>$6 hours	$>$6 hours

Table 2: Approximation ratio and run time for the exact algorithm for a randomly generated graph with $n = 1000$, $\sigma = \tau = 50$

8 Discussion

We have discussed several approximation and exact algorithms to solve the 9. We have analysed these algorithms theoretically and tested them on various different instances. In this section, we will discuss the theoretical and practical results of each algorithm. Furthermore, we will touch upon some methods of improving the algorithms discussed in this thesis.

8.1 Approximation algorithms

In this thesis, we have discussed four different approximation algorithms. Table 3 shows the time complexities and approximation factors of each of these algorithms.

Algorithm	Time complexity	Approximation factor
shortestPathsToTerminals	$O(n\log(n) + nm + qm)$	m
shortestBunch	$O(n^2\log(n) + n^2m + nqm)$	m
multipleBunchesApproximation factor	$O(n^2\log(n) + n^2m + nqm + mp)$	m
greedyrArborescence	$O(n^2 + n\log(n) + qm)$	$n-1$

Table 3: Time complexities and approximation factors of the various discussed approximation algorithms

The shortestBunch and multipleBunches Algorithms were developed as improvements of the shortestPathsToTerminals Algorithms. In some cases where the shortestPathsToTerminals performs very poorly, these algorithms can give better results. However, these algorithms do not always performs better than the shortestPathsToTerminals Algorithm; in the worst case, the return the same solution as the shortestPathsToTerminals Algorithm. We see this in Table 3; the approximation factors for these three algorithms are all the same. Since each of these three algorithms is an extension of the previous, each algorithm has a larger time complexity than the previous. The fourth approximation algorithm that was discussed, the greedyrArborescence Algorithm, has a very different approach. This algorithm has the second best time complexity of the four algorithms. However, the solution returned by the greedyrArborescence Algorithm can be up to $n-1$ times as large as the optimal solution. Since the number of terminals can be at most n, this algorithm will, in most cases, have the worst approximation factor.

Since the shortestPathsToTerminals has the best time complexity and approximation factor, it seems that this must be the best of these four algorithms. However, both the time complexity and the approximation factor are measures of how bad an algorithm can perform in the worst case. This means that, in practise, we might see some of the algorithms give better results than the others. Therefore, we tested the algorithms and analysed their performance.

First of all, we tested the algorithms on the graph ASML has provided, with randomly generated weights, for numerous different sets of start vertices and terminals, of various sizes. We saw that the approximation ratio, the value of the solution returned by the algorithm divided by the value of the optimal solution, is on average almost exactly the same for each of these algorithms. The approximation ratios are, on average, all around 1.03. The shortestBunch and multipleBunches Algorithms performed slightly better than the other two algorithms, but often returned the same solution as the shortestPathsToTerminals Algorithm. The fastest algorithm is the shortestPathsToTerminals Algorithm, which only ran for 0.03ms on average. The shortestBunch and greedyrArborescence Algorithms were both similar in run time, at around 0.2ms. The slowest algorithm was the multipleBunches Algorithm, which still only took 0.42ms, on average, to complete. For the input we tested, all algorithms almost always completed in less than a millisecond. We can conclude that these algorithms all perform quite well on this graph. They all complete rather fast, and the approximation ratios are quite close to 1. If we let the weights of the ASML graph all be equal to one, the algorithms performed even better. The average approximation ratios are now around 1.01, and the average run time is lower for each algorithm as well. This means that, if ASML chooses to not add weights to the arcs, the problem can be solved faster and return better solutions. However, this does return a solution that may be less relevant for ASML.

We also tested the algorithms on randomly generated graphs, to see how they perform on other graphs. We found that the approximation algorithms all perform better on smaller graphs, both in terms of approximation ratio and run time. Furthermore, we observed that they all performs worse on the random graphs than on the ASML graph, even if the number of vertices and arcs in the graph is almost the same as for the ASML graph. This holds for both the approximation ratio and run time. From this, we learned that the ASML graph has a certain structure on which these approximation algorithms perform better than on a randomly generated graph.

When testing the approximation algorithms on a random graph of 1000 vertices, we found that the algorithms all still complete in less than half a second. The run time of the shortestPathsToTerminals is even only 2ms. The approximation ratio, however, is now 1.2 for all approximation algorithms. This is much worse than the average approximation ratio for random graphs of $n = 50$ vertices. This shows that, if ASML were to extend their graph, the approximation ratios of these algorithms would very likely also become much worse. If this graph has similar properties to the current graph, the approximation algorithm would probably be at most 1.2, since the average approximation ratio for the current ASML graph is better than that of the random graphs of similar size.

There are some more things that can be investigated for which there was unfortunately no time during this project. For example, the run time of the multipleBunches Algorithm could be improved by computing the bunches for the different start vertices in parallel. As of now, the algorithm iterates over all start vertices and computes a shortest bunch for each start vertex and set of terminals that the start vertex should be connected to. However, using parallel computing, multiple bunches can be computed at the same time. Depending on one's computer, this could significantly reduce the runtime of the multipleBunches Algorithm.

8.2 Exact (combinatorial) algorithm

We have considered one combinatorial algorithm that solves the problem to optimality: the DirectedDreyfusWagner Algorithm. We have shown that the complexity of this algorithm is exponential in the number of terminals. When testing the algorithm, we saw that the run time was indeed exponential in the number of terminals. However, even for a small number of terminals, the run time was already over a hundred times longer than for the approximation algorithms. Then, for a larger number of terminals, the run time very quickly started to blow up.

The only advantage of the DirectedDreyfusWagner Algorithm compared to the approximation algorithms is that it guarantees to return an optimal solution. However, we have seen that there are other exact methods that solve the problem more efficiently: LP based methods.

8.3 LP based solution methods

The first LP based method that we discussed is the directed cut method. An immediate disadvantage of this method is that the number of constraints is exponential. Therefore, the ILP will quickly become very hard to solve as the size of the input graph increases. Therefore, we investigated only adding minimal subsets, which drastically decreased the number of constraints. However, to compute these, we still have to compute the subsets we had to compute before, which still takes exponential time and memory. On top of that, we have to compute which of these subsets is minimal, which again takes exponential time. Hence, this method does not seem to work well in practise. Therefore, we looked at adding constraints on the fly by finding the most violated constraint. This prevents the exponential time complexity and memory usage. Therefore, this is the implementation of the directed cut method that we chose to test.

For the directed cut method, we found that, despite the improvement of generating constraints dynamically, this method was not very efficient. For the ASML graph, this method reached the time limit of 15 minutes for most inputs we provided. Therefore, we could not analyse how this algorithm performs for different inputs. We can conclude, especially after comparing to the other LP based methods, that this method is not preferred.

Next we discussed the flow based and path based LP methods. For the path based method, we showed that if the path variables are continuous, the solution will still be integral. Since this relaxation can speed up the optimisation, we implemented this method with continuous path variables. We found that these LP methods optimise the problem much faster than the directed cut method. For the ASML graph with random weights, the flow based an path based LP methods completed in 67ms and 27ms, respectively. Despite the fact that this is based on only one run for each number of start vertices and terminals, this shows that these methods are much more efficient than the directed cut LP method. We saw that for both methods, the run time decreased when there was a large number of start vertices. Letting all arc weights of the graph be one, did not impact the run times of both algorithms.

When testing the algorithms on the randomly generated graphs, we found that the run time of both algorithms increases as the size of the graph increases. We also found that the run time of the path based LP method heavily depends on the number of paths from the root to the terminals. For the flow based approach, this seems to be irrelevant. When testing the flow based method on a random graph of 1000 vertices, the algorithm returned an optimal solution in 3 seconds. This is 1500 times slower than the fastest approximation algorithm. However, it is the only exact algorithm that we tested that returned a solution at all within 6 hours.

The path based approach resulted in a memory error when testing it on a random graph with 1000 vertices. For the path based LP method, we compute all paths from the root to all terminals beforehand. There is not enough memory to do this for a graph as large as this. This problem could be avoided by computing variables dynamically, for example with column generation. Column generation is an algorithm that is used when an LP contains too many variables to consider all at once. One starts with a smaller sized subproblem, and gradually adds more and more variables to the problem, until an optimal solution has been obtained. Due to the limited time for this project, and since the current ASML graph is rather small, this algorithm has not been implemented. However, if ASML were to extend their graph, this could be an interesting next step.

Furthermore, the run times of the LP methods could be improved by combining these algorithms with the approximation algorithms. An LP can sometimes be optimised faster when an initial solution is provided to the solver. This solution could be computed quickly by one of the approximation algorithms. When an initial feasible solution is provided to an LP solver, this can help the solver converge to an optimal solution faster. However, an LP solver is often able to find a better solution than a feasible solution provided by an approximation algorithm. Nonetheless, since some of the approximation algorithms complete very quickly, this could be worth implementing.

8.4 Problem specific advice

As stated before, we have observed that the ASML graph has a structure on which the approximation algorithms perform especially well. The small number of paths in the graph make the approximation algorithm complete especially fast and give a good approximation ratio. This property could be exploited further to improve the performance of the LP methods.

Furthermore, the structure of the ASML graph can be analysed further to see if we can gain information from this. For example, we can look for certain (repeating) patterns or properties of the graph. This information can then be exploited to develop algorithms specifically tailored to this type of graph. This way, we can develop both approximation algorithms and exact algorithms that perform even better on this graph. While there was no time for exploring this during this project, it can be interesting to focus on this in future research.

9 Conclusion

In this thesis, we discussed efficient methods to solve a problem posed by ASML. After stating the problem description and showing how the problem can be modelled as a directed Steiner tree problem, we showed that the problem is NP-hard. Then, we focused on efficiently solving the directed Steiner tree problem.

First of all, we discussed several approximation algorithms. We concluded that the first algorithm that was discussed, the shortestPathsToTerminals Algorithm, had both the best approximation factor and the best time complexity of the approximation algorithms that were discussed.

Then, we discussed a combinatorial algorithm that solves the directed Steiner tree problem to optimality. However, since the problem is NP-hard, this algorithm takes exponential time to complete. Therefore, we discussed several linear programming methods. Despite the fact that all algorithms minimise the same objective function, each linear program was based on a different idea and therefore has a different set of variables and constraints. The performance of each formulation is heavily dependent on the problem instance, and therefore we needed to test the different methods in order to find which is most efficient for our problem.

By testing all previously mentioned approximation and exact algorithms, we could compare their performances. We tested this on the graph corresponding to ASML's problem, for random positive arc weights and for unit arc weights. To be able to test the algorithms on different instances too, we generated random graphs with the same vertex-arc ratio as the ASML graph. By comparing these to the ASML graph, we found that the ASML graph contains a relatively small number of paths. Because of this, the randomly generated graphs can give different results than the ASML graph, even if the number of vertices and arcs is the same. We kept this in mind, but still tested the algorithms on the randomly generated graphs too, to get an idea of how the algorithms performs on graphs of different sizes.

On all instances we tested, we found that the shortestPathsToTerminals Algorithm is much faster than any of the other approximation algorithms. When we take the average of the approximation ratios of the approximation algorithms for various sizes of the sets of start vertices and terminals, we see that these are almost identical. Even though the other approximation algorithms sometimes compute better solutions, the extra time it takes to compute these solutions does not seem to be worth it for the small improvement this offers. For the ASML graph, the approximation ratios of all approximation algorithms are quite close to one. The algorithms return even better solutions, and complete faster, if all arc weights of the graph are set to one. The approximation ratios are slightly worse on randomly generated graphs than on the ASML graph, and increase as the number of vertices increases. However, even for a graph as large as 1000 vertices, the shortestPathsToTerminals Algorithm returns a solution that is 1.2 times worse than the optimal solution in only a few milliseconds. Therefore, we can conclude that the shortestPathsToTerminals algorithm is the best of the approximation algorithms.

We saw that the run time of the DirectedDreyfusWagner Algorithm is indeed exponential in the number of terminals. However, already for a small number of terminals, the algorithm is slower than two of the linear programming methods. Therefore, this algorithm does not seem to be suitable for our problem.

For an exact solution of the problem, we turned to the linear programming methods. We immediately saw that the directed cut approach resulted in very long run times, even when implementing the improvement described in Section 6.1.2. The most promising LP approaches are the flow and path based methods. Despite these methods being much slower than the approximation algorithms, for the ASML graph they both took less than a tenth of a second to complete on average. The path based method seems to work especially well on the ASML graph, since this algorithm benefits from having a small number of paths in the graph. On average, the path based LP was solved around twice as fast as the flow based LP for the ASML graph. However, for the randomly generated graphs, the difference in run time gets smaller, as these graphs mostly contain significantly more paths. The paths based approach failed for a graph of 1000 vertices, as there was not enough memory to generate all variables.

We can conclude that, for ASML's current graph, the path based LP is the best exact method. If the graph gets bigger, the flow based approach is most reliable. However, by generating variables dynamically, the

path based LP could be implemented in such a way that it also works for larger instances. This could be an interesting focus if the graph were to be extended.

Furthermore, the algorithms could be improved by exploiting the structure of the graph more. We have shown that the performance of many of the algorithms heavily depends on the number of paths in the graph. Therefore, algorithms such as the shortestPathsToTerminals Algorithm and the path based LP approach work especially well. Investigating certain properties of ASML's graph better can help give insight in what type of algorithms perform well on this specific problem. This information can be used to enhance the current algorithms such that they are tailored even better for this problem.

Whether one chooses for an approximation algorithm or an exact algorithm, depends on one's priorities. The shortestPathsToTerminals Algorithm has proven to return a solution very fast. However, it is often not the optimal solution. Even though the returned solutions were often not far from the optimal solution, the value of the solution returned by this algorithm could get as bad as the number of terminals times the value of the optimal solution. If an optimal solution needs to be guaranteed, one of the LP based methods would be the best choice. For a small graph, the current implementation of the path based LP returns an optimal solution fastest. For bigger graphs, one can choose the flow based solution, or investigate implementing an algorithm that improves the path based LP method.

References

[1] Hans Jürgen Prömel and Angelika Steger. *The Steiner tree problem: a tour through graphs, algorithms, and complexity*. Springer Science & Business Media, 2012.

[2] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[3] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[4] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[5] Moses Charikar, Chandra Chekuri, To-Yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed steiner problems. *Journal of Algorithms*, 33(1):73–91, 1999.

[6] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[7] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[8] Stuart E Dreyfus and Robert A Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[9] Bernhard H Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial optimization*, volume 1. Springer, 2011.

[10] Bernhard Fuchs, Walter Kern, and Xinhui Wang. Speeding up the dreyfus–wagner algorithm for minimum steiner trees. *Mathematical methods of operations research*, 66:117–125, 2007.

[11] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[12] Zachary Friggstad, Jochen Könemann, Young Kun-Ko, Anand Louis, Mohammad Shadravan, and Madhur Tulsiani. Linear programming hierarchies suffice for directed steiner tree. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 285–296. Springer, 2014.

[13] JX Hao and James B Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994.

[14] Thomas Rothvoß. Directed steiner tree and the Lasserre hierarchy. *arXiv preprint arXiv:1111.5473*, 2011.

[15] Erdős Paul and Rényi Alfréd. On random graphs i. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.