Technische Universität Berlin
Fakultät Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Algorithmik und Komplexitätstheorie

Funnels

Algorithmic Complexity of Problems on Special Directed Acyclic Graphs

Author
Marcelo Garlet Millani

Supervisors
Prof. Dr. Rolf Niedermeier
Hendrik Molter
Dr. Manuel Sorge

A thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

August 29, 2017

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den

.....
Unterschrift

Zusammenfassung

Wir führen eine Unterklasse von gerichteten azyklischen Graphen (DAGs) ein, und nennen sie *Funnels*. Unser Ziel hierbei ist es, die einfache Struktur von gerichteten Bäume zu bewahren, ohne zu viel von der Ausdruckskraft von DAGs zu verlieren.

Zuerst studieren wir die Klasse aus graphtheoretischer Warte, indem wir manche strukturelle Eingenschaften von Funnels zeigen. Dann wenden wir die Klasse auf konkrete Berechungsprobleme an und überprüfen, ob die betrachteten Probleme im Vergleich zu ihrer Lösung auf DAGs algorithmisch einfacher werden (d.h., in Polynomialzeit lösbar) wenn die Eingabe auf Funnels eingeschränkt wird. Schließlich betrachten wir die Graphklasse auch aus einer parametrisierten Warte, wobei wir vier Abstandsmaße zu Funnel definieren. Für das k-Linkage Problem (auch Vertex-Disjoint Paths genannt) erzielen wir positive parametrisierte Ergebnisse. Das Problem ist in quadratisch Zeit lösbar wenn der Kantenlöschungsabstand zu Funnel eine Konstante ist.

Zum Schluss führen wir Experimente durch, um die praktische Nützlichkeit der beschriebenen Algorithmen zu ermitteln. Wir stellen fest, dass der Kantenlöschungsabstand zu Funnel in der Praxis oft schnell berechnet und noch schneller auch zu einem fast optimalen Wert hin approximiert werden kann. Wir zeigen auch, dass die Eingabegröße einer k-Linkage-Instanz stark reduziert werden kann, falls der Kantenlöschungsabstand zu Funnel gering ist.

Abstract

We introduce a subclass of directed acyclic graphs (DAGs) which we call *funnels*. The purpose of this class is to keep some of the structural simplicity of directed trees without losing too much of the expressiveness of DAGs.

We first analyze this class from a graph-theoretical perspective, proving certain structural properties of funnels. Then, we apply the graph class to concrete computational problems, checking whether they become algorithmically easier (i.e., polynomial-time solvable) when the input is restricted to be a funnel. Finally, we also treat the class from a parameterized complexity point of view, defining four distance measures to funnel. For the k-Linkage problem (also known as Vertex-Disjoint Paths), we obtain positive parameterized results, with the problem becoming solvable in quadratic time when the arc-deletion distance to a funnel of the input DAG is a constant.

We conclude with an experimental evaluation where we analyze the practical usefulness of some of the algorithms described. We conclude that the arc-deletion distance to a funnel of a DAG can often be quickly computed in practice and it can also approximated to a value very close to the optimal one. We also showed that it is possible to considerably reduce the input size of k-Linkage instances with a small arc-deletion distance to a funnel.

Contents

Chapter 1

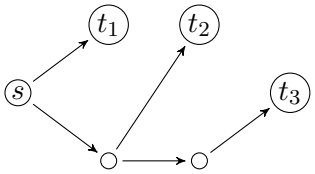
Introduction

Many relevant problems in computer science can be modeled using some kind of network. One can represent relationships between people, the connection between streets in a city, or the dependencies of tasks in a project using networks. The mathematical structure used to represent a network is called a graph. Graphs are composed of vertices which may be connected to each other through edges. For example, one can say that the vertices represent people and an edge between two of them indicates they are friends.

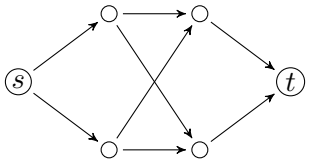
While some problems are adequately formulated with symmetric connections between vertices (i.e., if Ana is a friend of Bob, then Bob is also a friend of Ana), other problems require a notion of direction. When executing a task, for example cooking, it is important to know whether one is supposed to dice the carrots and then cook them, or cook and then dice them. In such cases we want to represent the dependency of tasks through a directed relationship, that is, a not necessarily symmetric connection. When directed edges (also called arcs) are present, the data structure is called directed graph (or digraph).

Both directed and undirected graphs are very powerful with respect to the information which can be modeled with them. One can then expect that solving certain problems where the input is an arbitrary graph to be computationally hard (that is, NP-hard) because of its structural complexity. Yet, sometimes restricting the input to have a simpler structure can make such problems "easy" (that is, solvable in polynomial time). For example, there is no known "efficient" algorithm for finding the longest path on a graph. In fact, the existence of a polynomial-time algorithm for this problem would contradict standard theoretical assumptions and is considered very unlikely. However, only because finding such a path in an arbitrary graph is hard does not mean that it is never easy to find one. If the graph has no cycles, then finding the longest path is easy: Since there is at most one path between any two vertices one can check the length of all paths in polynomial time.

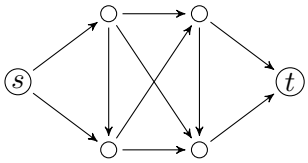
Often we are not interested in solving a problem for any arbitrary graph, but only for those in some restricted domain. The layout of streets in a city, as an example, is restricted to physical constraints and is mostly planar as the construction of overpasses is expensive. If we know that the input graph has certain properties, we can sometimes exploit such properties and efficiently solve otherwise hard problems. When restrict-



(a) In a tree, there is exactly one path from the root s to some leaf t_i. In this example we have three paths from s to some sink.



(b) A funnel has only polynomially many paths between two vertices. Here, there are four paths from s to t.



(c) A DAG can have exponentially many paths connecting two vertices. In this case, there are nine paths from s to t.

Figure 1.1: Comparison between a tree, a funnel and a DAG.

ing the domain of graphs which we are interested in, we are actually thinking about graph classes. A graph class is nothing but a family of graphs who share some specific property. Some of the classical graph classes are planar graphs (can be drawn on a two-dimensional surface without any crossing edges), forests (no cycles), regular graphs (all vertices have the same degree) and hypercube graphs (geometrical interpretation of vertices and edges), among others.

Certain graph classes can be considerably different when defined for digraphs in comparison to undirected graphs. Consider forests. These graphs have less edges than vertices, and they also have the property that there is at most one path between two vertices. On directed acyclic graphs (or DAGs), however, the number of arcs can be quadratic in the number of vertices, and there can be exponentially many paths between two vertices. Even though the defining property of the class is the same (that is, absence of cycles), the properties that follow from it are considerably different on undirected and directed graphs. Some of the properties from forests which are lost on DAGs are, however, desirable for the construction of efficient algorithms. At the same time, certain properties that are gained on DAGs (for example, the existence of multiple paths between two vertices) might be necessary in order to properly model the problem at question. In such cases it is natural to ask oneself if there is some subclass of DAGs which retain certain algorithmically interesting properties from forests without preventing relevant problem instances from being modeled. It is in this context that we consider a subclass of directed acyclic graphs.

The idea behind introducing this class is to retain certain properties from forests which are lost in the directed case. In Chapter 2 we provide a formal definition, but intuitively the digraphs in this class have the property that every path from a source (vertex with no incoming arcs) to a sink (no outgoing arcs) can be uniquely identified by one of its arcs, that is, it has an arc that is not shared with other paths from sources to sinks. One can immediately notice that this property implies that the number of such paths is not greater than the number of arcs. At the same time, it is still possible to have multiple paths connecting two vertices. Hence, it achieves what we wanted: Retaining some properties from forest without destroying too many of DAGs. For reasons that will become clear later, we call the digraphs in this subclass "funnels". A comparison

Parameter	Approximation (DAG)	Exact (DAG)	Exact (Digraph)
Arc-deletion	$\mathcal{O}(n+m)$, factor-3	$3^d(n+m)$	$5^d n(n+m)$
Vertex-deletion	?	$6^d n(n+m)$	?
Funnel height	-	$\mathcal{O}(n+m)$	not defined
Funnel depth	-	$\mathcal{O}(n+m)$	not defined

Table 1.1: Summary of the results of this work about the complexity of computing certain distance measures to funnel.

Problem	Funnel	Arc-deletion	Vertex-deletion
DAG Partitioning	$\mathcal{O}(n^3)$	?	NP-hard
k-Linkage	$\mathcal{O}(n^2)$	$\mathcal{O}(d)$ kernel	?
Max-Leaf Out-Branching	NP-hard	-	-
Path with Forbidden Pairs	$\mathcal{O}(m \cdot n^2)$	?	?

Table 1.2: Summary of the results of this work about the (parameterized) complexity of solving certain NP-hard problems on funnel-like DAGs.

between trees, funnels and DAGs is illustrated in Figure 1.1.

After formally defining the class of funnels, we describe four distance measures to funnel. Some of these measures are later used to develop algorithms which are efficient (i.e. fixed-parameter tractable) if the distance to a funnel of the input is small. We also consider the computational complexity of computing these measures (Table 1.1).

We then analyze the complexity of certain problems when the input is restricted to a funnel (column "Funnel" in Table 1.2). We also motivate why certain real-world data might be properly modeled with this restricted class. This is covered in Chapter 4. By certain NP-hard problems, like Path with Forbidden Pairs, we get straightforward positive results, with the problem becoming polynomial-time solvable. In contrast, Max-Leaf Out-Branching remains NP-hard even if the input is a funnel. We also analyze the parameterized complexity of such problems, in particular for parameters which measure how similar to a funnel a digraph is (columns "Arc-deletion" and "Vertex-deletion" in Table 1.2). For k-Linkage we obtain positive results, showing that the problem is fixed-parameter tractable with respect to arc-deletion distance to a funnel. That is, if we need to remove d arcs in order to turn a DAG into a funnel, then we get an algorithm with a running time $f(d) \cdot \text{poly}(n)$, where n is the input size and f some function.

1.1 Related Work

Graph Theory The definition of graph classes is fairly common. In the book *Digraphs* from Bang-Jensen and Gutin [BJG08] there is a chapter dedicated only to classes of digraphs. The usage of distance parameters is also very widespread in the field of parameterized complexity, with tree-width being a good example of such a parameter, defined by Bertele and Brioschi [BB72].

Ganian et al. [Gan+09] provide a survey with many problems which remain NP-hard on DAGs. They also define two variants of tree-width for digraphs which are also suitable for DAGs (i.e. are not always constant in DAGs).

k-Linkage In the k-Linkage problem (also known as the Vertex-Disjoint Paths problem), the input consists of a digraph and a set of pairs of vertices $P = \{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ and the goal is to find vertex-disjoint paths connecting each s_i to its corresponding t_i.

Fortune, Hopcroft, and Wyllie [FHW80] showed that Vertex-Disjoint Paths is NP-hard on directed graphs even if the number of pairs equals two. They also provided an $\mathcal{O}(knm^k)$ algorithm for the case where the input is a DAG. For $k = 2$ there is an elegant polynomial-time algorithm due to Shiloach and Perl [SP78]. A generalization from this algorithm for an arbitrary k is presented in the book *Digraphs* from Bang-Jensen and Gutin [BJG08]. Schrijver [Sch94] provided an $\mathcal{O}(n^k)$ algorithm for planar DAGs.

Slivkins [Sli10] showed that k-Linkage is W[1]-hard with respect to k on DAGs. It was posed as an open question by Downey and Fellows [DF12] whether the problem is in FPT with respect to k on planar DAGs. This question was answered positevely by Cygan et al. [Cyg+13], who provided an $2^{2^{\mathcal{O}(k^2)}} \cdot n^{\mathcal{O}(1)}$ time algorithm for planar DAGs.

DAG Partitioning The DAG Partitioning problem consists of removing at most k arcs from a DAG such that each connected component has exactly one sink. It was first defined by Leskovec, Backstrom, and Kleinberg [LBK09] and used in order to study the behavior of the news cycle. Alamdari and Mehrabian [AM12] proved that, for any $\epsilon > 0$ there is no $\mathcal{O}(n^{1-\epsilon})$-approximation for DAG Partitioning, unless P = NP. They also presented a polynomial-time algorithm for graphs with bounded pathwidth. Van Bevern et al. [Bev+17] showed that the problem is in FPT with respect to k by giving an algorithm with running time $\mathcal{O}(2^k \cdot (|V| + |A|))$.

Max-Leaf Out-Branching The goal in Max-Leaf Out-Branching is to remove arcs of a DAG in order to transform it into an out-tree with at least k leaves. Alon et al. [Alo+07] showed that the problem is in FPT with respect to k for a class of digraphs which contains DAGs. Bonsma and Dorn [BD07] provided an algorithm with running time $2^{\mathcal{O}(k^3 \log k)} \cdot n^{\mathcal{O}(1)}$ for Max-Leaf Out-Branching on digraphs. Bonsma and Dorn [BD11] improved on this result, giving a $2^{\mathcal{O}(k \log k)} \cdot n^{\mathcal{O}(1)}$-time algorithm for the problem.

1.2 Definitions and Notation

In this section we describe the notation used in this work and formally define certain structures and operations. We recommend reading the subsection about graphs, since they appear quite often, and coming back later to the other subsections as the need arises.

1.2.1 Sets and Sequences

We use braces ({ }) to denote a set of unique, unordered elements. We write $\{a_i\}_{i=1}^n = \{a_1, a_2, \ldots, a_n\}$ to denote a set of n indexed elements. Unlike sets, sequences are ordered and allow repeated elements. They can be represented using tuples, but we often write sequences without any separator between the elements, meaning both $a_1a_2 \ldots a_n$ and $(a_1, a_2, \ldots, a_n)$ describe the same sequence of n indexed elements. The number of elements (not necessarily distinct) of a set or sequence S is given by $|S|$. For sets this value is called *size*, while for sequences it is called *length*.

Since converting a sequence to a set can be done in an intuitive way by simply removing duplicates and ignoring the ordering, we sometimes apply certain set operators to sequences. After implicitly converting the operands to sets, the result remains a set. That is, if $A \subseteq S$ is a set and $p \in S^n$ is a sequence, then $p \cap A$ is the set of all elements of p which are also in A.

Given two sequences $P = a_1a_2 \ldots a_n$ and $q = b_1b_2 \ldots b_m$, their concatenation is given by $PQ = a_1a_2 \ldots a_nb_1b_2 \ldots b_m$. We say that P is a prefix of Q if there is a sequence P' such that $Q = PP'$. Similarly, P is a suffix of Q if there is a P' such that $Q = P'P$. A sequence P is a subsequence of Q (denoted by $P \subseteq Q$) if and only if there are sequences R, S such that $Q = RPS$.

1.2.2 Graphs

An undirected graph $G = (V, E)$ is a tuple with a set V of vertices and a set E of edges where each edge $\{u, v\} \in E$ connects two vertices of V. Formally we have $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$. The neighbors of a vertex are the vertices that are connected through an edge to v. They are given by $N_G(v) = \{u \in V \mid \{u, v\} \in E\}$. As with all operations on graphs, the index G is omitted from the operator if it is clear from the context. The degree of a vertex v is defined as the number of its neighbors. It is formally written as $\deg_G(v) = |N(v)|$.

A directed graph $D = (V, A)$ (or digraph) is a graph where the edges have a direction (and are called arcs instead). That is, the arc (u, v) goes from u to v. Thus $A \subseteq \{(u, v) \mid u, v \in V \text{ and } u \neq v\}$. Since the arcs have a direction, we differentiate between incoming and outgoing neighbors. The inneighbors of a vertex v are denoted by $\text{in}_D(v) = \{u \in V \mid (u, v) \in A\}$. Its outneighbors are given by $\text{out}_D(v) = \{u \in V \mid (v, u) \in A\}$. The indegree of a vertex v is written as $\text{indeg}_D(v) = |\text{in}(v)|$, and its outdegree as $\text{outdeg}_D(v) = |\text{out}(v)|$. We can extract the set of arcs of D with $\mathcal{A}(D)$, and the set of vertices with $\mathcal{V}(D)$.

We extend the definition of set operators for directed and undirected graphs. Let $G = (V, E), H = (U, F)$ be two graphs. We define

$$
\begin{aligned}
H \subseteq G &\Leftrightarrow U \subseteq V \wedge F \subseteq E, \\
H \cup G &= (U \cup V, F \cup E), \\
H \cap G &= (U \cap V, F \cap E), \\
H \setminus G &= (U \setminus V, F \cap (U \setminus V)^2).
\end{aligned}
$$

In particular, we say that H is a subgraph of G if and only if $H \subseteq G$. The induced subgraph of a set U of vertices is a graph that is composed by U and all edges between them that are present in the supergraph. For any function that produces a set of vertices we define the induced subgraph of this function by using square instead of round brackets around the parameter. Formally, let $f : V \to \mathcal{P}(V)$ be a function. Then $f[v] = (f(v), \{\{u, w\} \in E \mid u, w \in f(v)\})$. We also extend the $\setminus$ operator for vertex- and arc-sets. That is, if E' is an arc-set and V' a vertex-set, then $G \setminus E' = (V, E \setminus E')$ and $G \setminus V' = G \setminus (V', \emptyset)$.

Contracting an edge $\{a, b\} \in E$ means adding a vertex v and the required edges such that $N(v) = N(a) \cup N(b)$ and then removing a and b. Contracting an arc $(a, b) \in A$ is defined analogously, with that difference that we instead require $\text{indeg}(v) = \text{indeg}(a) \cup \text{indeg}(b)$ and $\text{outdeg}(v) = \text{outdeg}(a) \cup \text{outdeg}(b)$.

Subdividing an edge $\{a, b\} \in E$ means removing that edge and adding a new vertex c together with the edges $\{a, c\}, \{c, b\}$. On digraphs we add the arcs $(a, c), (c, b)$ when subdividing the arc (a, b). A graph H is a *topological minor* of G if some H' can be obtained from H by subdividing its edges (or arcs) such that H' is isomorphic to some $G' \subseteq G$.

Given two vertices u, v of a graph, a walk from u to v is a sequence of vertices $P = uw_1w_2 \dots v$ where for any $(a, b) \subseteq P$ the edge $\{a, b\}$ exists (or the arc (a, b) in digraphs) on the graph. Note that a walk may contain a single vertex. A walk from u to v is called a path (also (u, v)-path) if no vertex appears more than once. A circuit is a walk where the first and last vertices are the same. A cycle is a circuit where all vertices but the first and last are different.

In digraphs, the vertices which can be reached from a vertex v are given by $\text{out}^*(v)$. The vertices which can reach v are given by $\text{in}^*(v)$. That is $u \in \text{out}^*(v)$ if and only if there is a (u, v)-path, and $u \in \text{in}^*(v)$ if and only if there is a (v, u)-path.

1.2.3 Complexity Theory

When analyzing the time complexity of an algorithm, we use the standard Big-O notation. For a function $f : \mathbb{N} \to \mathbb{N}$ we define

$$\mathcal{O}(f) \coloneqq \{g : \mathbb{N} \to \mathbb{N} \mid \exists n_0 \in \mathbb{N}\, \exists c \in \mathbb{Q}\, \forall n > n_0 : g(n) \leq c \cdot f(n)\}.$$

We frequently use the function definition directly, that is, we write $\mathcal{O}(n^2)$ instead of $\mathcal{O}(f)$ for $f(n) = n^2$.

We model a decision problem as a language $L \subseteq \Sigma^*$, with Σ being some finite alphabet and Σ^* being the set of all words over Σ. We say that an input word $x \in \Sigma^*$ is a yes-instance if and only if $x \in L$. Likewise, x is a no-instance if and only if $x \notin L$.

We classify a language L with respect to the worst-case time complexity of deciding if $x \in L$, for any input x. If there is a deterministic Turing-machine which can decide L in polynomial time (i.e., in $\mathcal{O}(n^c)$ time for some fixed $c \in \mathbb{N}$), then we say $L \in \text{P}$. If there is a non-deterministic Turing machine which can decide L in polynomial time, then $L \in \text{NP}$. There is a long-standing open question of whether $\text{P} = \text{NP}$ or not. It is,

however, widely believed that P $\neq$ NP, an assumption which is often used to exclude the existence of polynomial-time algorithms for some problem.

A language L_1 is reducible in polynomial time to another language L_2 if and only if there is a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that

$$\forall x \in \Sigma^* : f(x) \in L_2 \Leftrightarrow x \in L_1.$$

A language L_1 is said to be NP-hard if every language L_2 can be reduced in polynomial-time to L_1. If $L_1 \in$ NP, it is also said to be NP-complete.

We say that a language L is in coNP if its complement $\Sigma^* \setminus L$ is in NP. Similarly, a language L_1 is coNP-hard if every language in coNP can be reduced in polynomial-time to L_1. From the definition of NP-hardness we have that, if some NP-complete language is in coNP, then NP = coNP. Similarly to the P versus NP question, it is widely believed that NP $\neq$ coNP. For further details on the vast field of computational complexity, refer to standard books on the topic (e.g., [GJ90], [AB09] and [Pap03]).

A problem is said to be fixed-parameter tractable (FPT) with respect to some parameter k if it can be solved in $f(k) \cdot n^c$ time, for some function $f : \mathbb{N} \to \mathbb{N}$ and some constant $c \in \mathbb{N}$. In the context of parameterized complexity we also have the definition of the W-hierarchy of complexity classes, yet they do not have an intuitive definition through a machine model like the NP class. In this work, we only need to know that, if a parameterized problem is W[1]-hard with respect to some parameter k, then it is unlikely to be in FPT with respect to k. For further details, refer to standard books about parameterized complexity (e.g., [DF12], [Nie06] and [Cyg+15]).

We say that a problem L admits a problem kernel with respect to some parameter k if we can compute in polynomial time for any instance x another instance y such that $|y| \leq f(k)$ and $y \in L \Leftrightarrow x \in L$, for some function $f : \mathbb{N} \to \mathbb{N}$.

Chapter 2

Funnels

In this chapter we formally introduce the central topic of this work, namely a subclass of directed acyclic graphs (DAGs) which we call *funnels*. In Section 2.1 we provide a formal definition of this class together with some examples of DAGs which satisfy the definition. We then show in Section 2.2 different characterizations of this class.

2.1 Formal Definition

Undirected acyclic graphs (that is, forests) are known to have a fairly simple structure, which often allows polynomial-time algorithms for otherwise NP-hard problems (for example, Independent Set and Longest Path). Directed acyclic graphs, however, have a more complicated structure, and certain problems which are polynomial-time solvable on forests become NP-hard on DAGs (for example, certain variations from scheduling [PY90]). In a sense, there is a huge gap between forests and DAGs, which raises the natural question of whether some natural graph classes exist inside this gap.

We define funnels in such a way that some structural properties of forests are kept without taking too much of the expressiveness of DAGs. Our goal is to restrict the structure of the input in order to solve problems which are NP-hard on DAGs more efficiently (e.g. in polynomial time). To this end, we limit in this class the number of paths that may exist between two vertices (which can be exponential in DAGs). Requiring every path between two vertices to be unique would be possibly too restrictive, and in the case of a single source such DAGs would be just directed trees. Instead, we require each path going from a source to a sink to be uniquely identified by one of its arcs. These arcs are said to be *private* since there is only one path from a source to a sink which goes through them. This immediately gives us an upper bound for the number of paths between two vertices, which obviously cannot be greater than the number of arcs in the digraph. However, it is still possible to have multiple paths between two vertices.

Before proceeding to the definition of funnels, we first define what it means for an arc to be private. From now on we use the term *source-sink path* to refer to any path which starts at some source and ends at some sink.

Figure 2.1: Example of a funnel (left) and a DAG which is not a funnel. Private arcs are marked as dashed lines. The DAG on the right is not a funnel because all arcs in an (s_1, t)-path are shared. Removing one arc turns it into a funnel.

Definition 2.1.1 (Private arc)**.** Let $D = (V, A)$ be a DAG. An arc $a \in A$ is said to be *private* if there is only one source-sink path which contains a.

We can now define funnels formally. An example of a funnel can be seen in Figure 2.1. The reason for the name, as we will substantiate in Proposition 2.2.1 below, is that, as soon as a vertex has an indegree greater than one, all of its successors must have outdegree at most one. In a sense, these vertices funnel paths towards a sink.

Definition 2.1.2 (Funnel)**.** A DAG D is a *funnel* if every source-sink path has at least one private arc.

From this definition we obtain a simple property of funnels, namely the fact that the number of source-sink paths is linearly bounded (as opposed to an exponential function in general DAGs). This property comes from the fact that each such path has a private arc, and the amount of private arcs is clearly bounded on the total number of arcs in the DAG.

Corollary 2.1.3. *Every funnel $D = (V, A)$ has at most $|A|$ different source-sink paths.*

2.2 Equivalent Definitions

We now consider some different ways of characterizing a funnel. Some of them give us an efficient method to check if a DAG is a funnel, while others are useful for further proofs.

2.2.1 Degree Characterization

Definition 2.1.2 does not give us a very efficient way of checking whether a digraph is a funnel or not. A simple implementation which counts how many paths go through each arc (and stops if some path has no private arc) would take $\mathcal{O}(|A|^2)$ time. A more efficient way of making this check is to use an alternative definition of funnels which considers the degrees of the vertices. The idea is to show that, if two different paths meet at some point and diverge later on, then none of the arcs on either path is private.

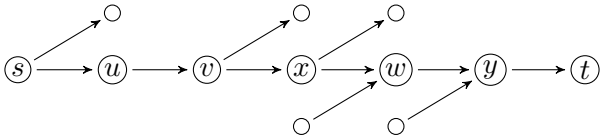


Figure 2.2: Illustration of the vertices used in the proof of Proposition 2.2.1. We argue that the arc (x, w) is private.

Also refer to Figure 2.2 while reading the proof below. Recall that a DAG induces a total ordering of its vertices, and so the first or last vertex of some set is always well-defined.

Proposition 2.2.1. *A DAG $D = (V, A)$ is a funnel if and only if for each vertex $v \in V$:*

$$\text{indeg}(v) > 1 \Rightarrow \forall u \in \text{out}^*(v) : \text{outdeg}(u) \leq 1. \tag{2.1}$$

Proof. We first show that a DAG that satisfies Inequality (2.1) is a funnel. The idea is to identify the private arcs and to argue that each path must contain at least one of those arcs.

Let $D = (V, A)$ be a DAG which satisfies Inequality (2.1), let $s \in V$ be a source and $t \in V$ be a sink such that some (s, t)-path exists, and let v be the first vertex in $\text{out}^*(s)$ with $\text{outdeg}(v) > 1$. If no such vertex v exists, then there is only one (s, t)-path in D and all outgoing arcs from s are private. Otherwise, we know that $\forall u \in \text{in}^*(v) \setminus \{s\} : \text{indeg}(u) = 1$, otherwise 2.1 would not hold. This means that there is exactly one (s, v)-path. Let P be some (s, t)-path that goes through v and let w be the first vertex in this path with $\text{indeg}(w) > 1$. If no such w exists, then there is only one (v, t)-path and all arcs after v are private, as required. Otherwise we consider a vertex x such that the arc (x, w) is in P. We know $\text{indeg}(x) = 1$, which implies there is only one (v, x)-path. Since the (v, x)-path as well as the (s, v)-path are unique and $\forall y \in \text{out}^*(w) : \text{outdeg}(y) = 1$, the arc (x, w) is private for the (s, t)-path that contains it. Thus, D is a funnel.

We now show that every funnel satisfies Inequality (2.1). We do this by contradiction, showing that every arc of some (s, t)-path is present in at least one other source-sink path if Inequality (2.1) does not hold.

Let $D = (V, A)$ be a funnel. Without loss of generality we can assume that D has only one source s and only one sink t. If this is not the case, we add a single source and arcs from it to all previous sources, and we also add a single sink and arcs from the previous sinks to the new one. This basically adds one vertex to the beginning and another to the end of the sequence that defines each path, and if any arc was private for a path it will remain private.

Assume towards a contradiction that Inequality (2.1) is not true. This means there is some vertex $u \in V$ with $\text{indeg}(u) > 1$ and that there is some other vertex $w \in \text{out}^*(u)$ with $\text{outdeg}(w) > 1$. Let w be the first such vertex. Then there are at least $\text{indeg}(u)$ many (s, u)-paths and $\text{outdeg}(w)$ many (w, t)-paths. Since there is at least one (u, w)-path (possibly without arcs), this implies that every arc in the induced subgraph $\text{in}^*[u]$ is shared by $\text{outdeg}(w)$ many paths, every arc in $\text{out}^*[w]$ is shared by $\text{indeg}(u)$ many paths and all arcs in a (u, w)-path are shared by $\text{indeg}(u) \cdot \text{outdeg}(w)$ many

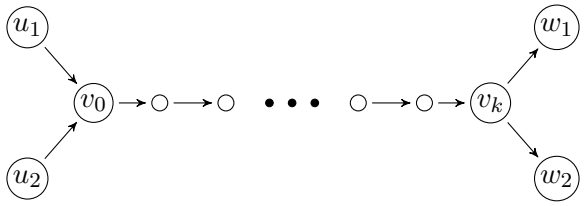


Figure 2.3: Forbidden subgraphs for funnels.

paths. Hence, all arcs in an (s,t)-path which goes through u and w are shared. This contradicts the fact that D is a funnel. □

From Proposition 2.2.1 implies following corollary by setting u to v in Inequality (2.1).

Corollary 2.2.2. *If $D = (V, A)$ is a funnel then*

$$\forall v \in V : \text{outdeg}(v) \leq 1 \vee \text{indeg}(v) \leq 1.$$

2.2.2 Forbidden Subgraphs and Minors

While the degree characterization is useful for algorithms, it can make the mathematical notation for proofs somewhat complicated. That is, it is harder to formally argue about a substructure which violates the funnel property when using only the degrees of the vertices on the DAG. In many cases, arguing over some forbidden subgraph or minor is much simpler and makes proofs easier to understand. Fortunately, using the previous characterization it is easy to define a family of forbidden subgraphs for funnels. The condition in Proposition 2.2.1 implies that a DAG D is not a funnel if there is some path $v_1 v_2 \dots v_k$ with $\text{indeg}(v_1) > 1$ and $\text{outdeg}(v_n) > 1$. In order to show that the absence of such a path is both sufficient and necessary for a DAG to be a funnel, we first define $D_k \coloneqq (V_k, A_k)$ where (see Figure 2.3 for a graphical representation)

$$\begin{aligned} V_k &= \{u_1, u_2, v_0, w_1, w_2\} \cup \{v_i\}_{i=1}^{k} \text{ and} \\ A_k &= \{(u_1, v_0), (u_2, v_0), (v_k, w_1), (v_k, w_2)\} \cup \{(v_i, v_{i+1})\}_{i=1}^{k-1}. \end{aligned}$$

With this we define an infinite family of forbidden subgraphs $\mathcal{F} \coloneqq \{D_i\}_{i=0}^{\infty}$. We can then prove the following statement.

Proposition 2.2.3. *A DAG D is a funnel if and only if $\forall C \subseteq D : C \notin \mathcal{F}$.*

Proof. We prove this by contraposition. That is, we show that $\exists C \subseteq D : C \in \mathcal{F}$ if and only if D is not a funnel. Both directions can be easily shown using Proposition 2.2.1.

Let $C \subseteq D$ be a subgraph of D such that $C \in \mathcal{F}$. By definition of C it contains some vertex v_0 with $\text{indeg}_C(v_0) = 2$ and another vertex $v_k \in \text{out}^*_C(v_0)$ with $\text{outdeg}_C(v_k) = 2$. This implies $\text{indeg}_D(v_0) > 1$ and $\text{outdeg}_D(v_k) > 1$. From Proposition 2.2.1 we know D is not a funnel.

Now assume D is not a funnel. From Proposition 2.2.1 we know there is some vertex v with $\text{indeg}_D(v) > 1$ and another vertex $u \in \text{out}^*_D(v)$ with $\text{outdeg}_D(u) > 1$. Let $u_1, u_2 \in \text{in}_D(v)$ and $w_1, w_2 \in \text{out}_D(u)$ be four distinct vertices. Let $vv_1v_2 \dots v_{k-1}u$ be a (v, u)-path. We set $v_0 \coloneqq v$ and $v_k \coloneqq u$, obtaining the forbidden subgraph D_k if $u \neq v$, and D_0 otherwise. Hence, D contains a subgraph from $\mathcal{F}$. □

From the characterization by forbidden subgraphs we obtain another property of funnels, namely that they are *hereditary*. That is, every subgraph of a funnel is also a funnel. This property follows directly from the fact that the subgraph relationship is transitive. Hence, we obtain the following corollary.

Corollary 2.2.4. *Funnels are hereditary, that is, if D is a funnel then every $C \subseteq D$ is also a funnel.*

Proof. Assume C is not a funnel, but D is. By Proposition 2.2.3 $\exists E \subseteq C : E \in \mathcal{F}$. Hence $E \subseteq C \subseteq D \Rightarrow E \subseteq D$, which contradicts the fact that D is a funnel. □

Similarly, instead of characterizing funnels by an infinite family of forbidden subgraphs we can also use the concept of forbidden topological minors. Recall that subdividing an arc (a, b) means removing it and adding a vertex c together with the arcs $(a, c), (c, b)$. Subdividing a digraph successively subdividing some of its arcs. Funnels contain two forbidden topological minors, namely D_0 and D_1.

Proposition 2.2.5. *A DAG D is a funnel if and only if it does not contain D_0 or D_1 as a topological minor.*

Proof. We use the characterization with forbidden subgraphs from Proposition 2.2.3 in this proof. It is enough to show that any $D_i \in \mathcal{F}$ can be obtained by subdividing D_0 or D_1 multiple times, and that any subdivision of D_0 and D_1 contains some digraph of $\mathcal{F}$ as a subgraph.

We first show that we can generate $\mathcal{F}$ by subdividing D_0 and D_1. Let $D_i \in \mathcal{F}$. If $i \leq 1$ then D_i obviously contains itself as a topological minor. If $i > 1$, then by subdividing the arc (v_0, v_1) from D_1 a total of $i - 1$ times, we obtain D_i. Hence all digraphs in $\mathcal{F}$ can be generated by D_0 and D_1 through subdivisions.

Now we show that any subdivision of D_0 and D_1 contains some digraph from $\mathcal{F}$. Since subdividing arcs does not change the degrees of the affected vertices, from Proposition 2.2.1 we know any subdivision of D_0 is not a funnel and, hence, contains some digraph from $\mathcal{F}$ as a subgraph.

For any subdivision D'_1 of D_1 we know $\text{indeg}_{D'_1}(v_0) = 2$, $\text{outdeg}_{D'_1}(v_1) = 2$ and $v_1 \in \text{out}^*_{D'_1}(v_0)$. Again, from Proposition 2.2.1 we know D'_1 is not a funnel, and hence contains a subgraph from $\mathcal{F}$. □

2.2.3 Constructive Characterization

Although it can be useful to check whether a certain DAG is a funnel (for example, if we want to choose between a fast algorithm that only works for funnels and a slower

Fork Merge

Figure 2.4: Illustration of the different operations which create a funnel.

one which always works), it is sometimes more convenient to guarantee beforehand that the input is a funnel. That is, if we use some procedure to generate a funnel from certain data, we might be interested in guaranteeing that, regardless of how the data looks like, the produced DAG is always a funnel. To this end, we give two operations which allow us to construct any funnel and that also always produce a funnel. The operations are illustrated in Figure 2.4.

We describe each operation as a condition and an effect. The condition is just a Boolean formula which needs to be satisfied in order for the operation be applicable. In order to avoid brackets we mix symbolic operators ($\vee, \wedge$) with the respective verbal variants (or, and). The symbolic operators have a higher precedence, that is, they are evaluated before the verbal variants. For example, when we write "$p \vee q$ and $r \vee s$" we mean that both $(r \vee s)$ and $(p \vee q)$ need to be true.

The effect of an operation describes how the initial funnel D is modified by each operation. We use an auxiliary table $F : V \to \{\texttt{true}, \texttt{false}\}$ which describes whether a vertex v is "forkable" ($F(v) = \texttt{true}$) or not ($F(v) = \texttt{false}$). That is, if $F(v) = \texttt{true}$ then v is allowed to have outdegree greater than one. The core idea of the operations is that, as soon as a vertex has its indegree increased to a value greater than one, the vertex itself and all of its successors must have outdegree at most one. In order to keep the rules efficient, we prevent adding incoming arcs to vertices which have already forked, even if their outdegree is one. Thus, we do not need to check the degree of the successors of a vertex when adding an arc.

We can construct a funnel as follows. First, create a trivial funnel $D = (V, A)$ containing only some sources s_i and set $F(s_i) \coloneqq \texttt{true}$. Then, iteratively select some vertex $v \in V$ and execute one of the following operations (if the conditions are met).

The Fork operation adds one outneighbor to a vertex. This is allowed if the vertex has outdegree zero or if it is allowed to have outdegree greater than one, that is, if its predecessors all have indegree at most one (recall the degree characterization from Proposition 2.2.1).

Fork(v, u)
Condition: $F(v) \vee \text{outdeg}(v) = 0$ and $u \notin V$
Effect: $D \coloneqq (V \cup \{u\}, A \cup \{(v, u)\})$, $F(u) \coloneqq F(v)$

The Merge operation adds an arc between two existing vertices, causing one of them to have an indegree greater than one and, thus, no longer being forkable. We can apply this operation to a vertex v whenever $\forall u \in \text{out}^*(v) : \text{outdeg}(u) \leq 1$. This happens when v has no successors or when it is no longer forkable.

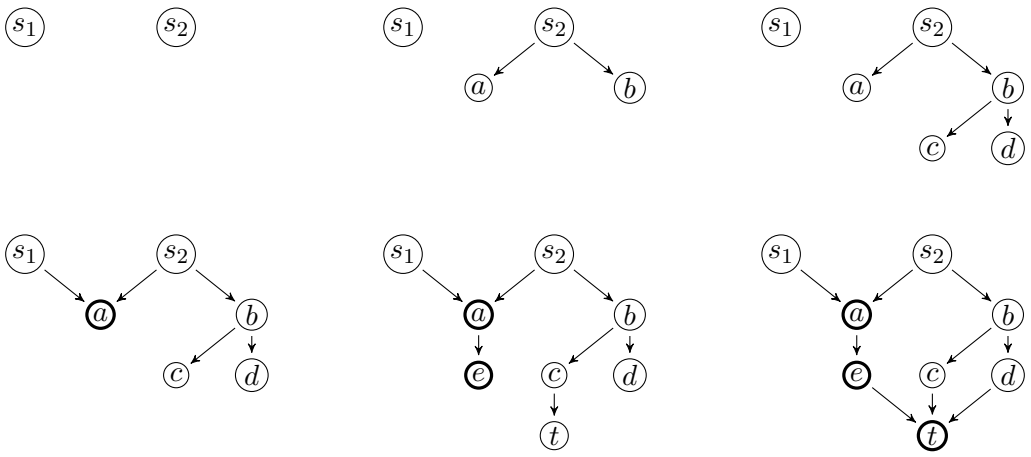


Figure 2.5: Step-by-step construction of a funnel using Fork and Merge operations, as described in Example 2.2.1. Vertices such that F is false are marked in bold. Note that, if the vertex e were created before adding the arc (s_1, a), it would not be possible to add such an arc afterwards using a Merge.

Merge(v, u)
Condition: $\neg F(v) \vee \text{outdeg}(v) = 0$ and $F(u) \vee \text{outdeg}(u) = 0 = \text{outdeg}(v)$
Effect: $D \coloneqq (V, A \cup (u, v))$, $F(v) \coloneqq \texttt{false}$

Example 2.2.1. We reconstruct the funnel from Figure 2.1 using the operations defined above. The step-by-step procedure is illustrated in Figure 2.5. We start with $D := (\{s_1, s_2\}, \{\})$. Then we execute the following operations in order: Fork(s_2, a), Fork(s_2, b), Fork(b, c), Fork(b, d), Merge(a, s_1), Fork(a, e), Fork(c, t), Merge(a, t) and Merge(d, t).

From the operations above one can see that there are two types of vertices in a funnel: Those which are "forkable" and those which are not. Later, we use this property when developing algorithms to compute the distance of a DAG to a funnel.

It is fairly straightforward to see that applying a Fork or a Merge to a funnel will produce a funnel. In order to show that these rules construct any funnel, we provide a method which reconstructs its input. This result are formally described below.

Proposition 2.2.6. *A DAG is a funnel if and only if it can be generated with* Fork *and* Merge *operations.*

Proof. We start by showing that the Fork and Merge operations always produce a funnel. We do this by induction, that is, we show that applying any of these rules to a funnel always produces a funnel. Since we always start with a trivial funnel, this guarantees that the procedure described above is correct. Note, however, that we need a function F in order to apply the operations, which is not given by the input digraph. Even though we could describe how to compute F for any funnel, it is simpler to restrict the proof to show that applying one of the operations on a funnel generated by the

procedure produces a funnel. This is no real restriction because later we show that every funnel can be generated by the above defined operations.

Let $D = (V, A)$ be a funnel generated by the rules above and F the auxiliary table internally used by the procedure. We show that applying any operation will produce a funnel D'. We first observe for every $v \in V$ that $F(v)$ implies $\text{indeg}(v) \leq 1 \wedge \forall u \in \text{in}^*(v) : F(u)$, and that $\neg F(v)$ implies $\text{outdeg}(v) \leq 1$.

Assume we apply $\textsc{Fork}(v, u)$, with v, u satisfying the conditions of the operation, and obtain D'. Clearly, $\text{indeg}_{D'}(u) = 1$ and $\text{outdeg}_{D'}(u) = 0$. For v, we have $\text{indeg}_D(v) = \text{indeg}_{D'}(v)$. If $\text{indeg}_{D'}(v) > 1$, we know that $\text{outdeg}_{D'}(v) = 1$, otherwise v would not satisfy the condition $\text{outdeg}_D(v) = 0$ since $F(v) = \texttt{false}$ before applying the operation. If $\text{indeg}_{D'}(v) \leq 1$, then its outdegree is irrelevant. Since the vertices in $\text{out}^*_D(v)$ were not modified in D', the new digraph remains a funnel.

Now assume we apply $\textsc{Merge}(v, u)$ instead, with v, u satisfying the conditions of the operation, and obtain D'. If $F(u)$, then we can add the arc (u, v) without destroying the funnel property since we have $\text{outdeg}(v) \leq 1$ and $\forall w \in \text{out}^*(v) : \text{outdeg}(w) \leq 1$. If $\text{outdeg}_D(u) = 0$, then we can add the arc (u, v) as long as $u \notin \text{out}^*(v)$. However, since we also require that $\text{outdeg}(v) = 0$ or $F(u)$, we know that $u \notin \text{out}^*(v)$. Hence, the new digraph is also a funnel.

We now show that we can construct any funnel using Fork and Merge operations. Let $D = (V, A)$ be a funnel. We reconstruct D iteratively as follows. First, consider some topological ordering of the vertices of D. Now initialize $D' = (V', A')$ with all sources of D.

Following the previously established ordering where v_i is the i-th vertex, execute $\textsc{Merge}(v_i, u)$ for all $u \in \text{in}_D(v_i)$ if $\text{indeg}_D(v_i) > 1$ and $(u, v_i) \notin A'$. To check if we can execute the Merge, we first note that $\text{outdeg}_{D'}(v_i) = 0$. Furthermore, if any $\textsc{Merge}(u, w)$ was executed with u and some $w \in V'$, then $\text{outdeg}(u) = 0$, otherwise $\text{outdeg}_D(u) > 1$ and $\text{indeg}_D(u) > 1$, which is not possible since D is a funnel. Hence, it is possible to execute the desired $\textsc{Merge}(v_i, u)$. If $\text{indeg}_D(v_i) \leq 1$, then the necessary incoming arcs, if any, are already present due to the topological ordering and no Merge is required. Afterwards, execute $\textsc{Fork}(v_i, u)$ for all $u \in \text{out}_D(v_i) \setminus V'$. Since, at this point, $\text{outdeg}_{D'}(v_i) = 0$, this is possible.

Every arc $(v, u) \in A$ is created either by executing $\textsc{Fork}(v, u)$ or $\textsc{Merge}(u, v)$. Thus, after executing both operations for each vertex, all arcs and vertices from D will be created and $D' = D$. □

2.3 Summary

The constructive characterization gives us some insight about the structure of a funnel. Recall that an out-tree is a directed tree where all arcs point outwards from the root, and an in-tree is one where all arcs point towards the root. One can see that a funnel can be partitioned into two induced subgraphs: One is an out-forest (where $F(v) = \texttt{true}$) and the other is an in-forest. With a simple argument it is possible to give an upper bound on the number of arcs in a funnel. It is also easy to see from the

constructive proof below that the bound is sharp, that is, that for any even $n \geq 2$ there is a funnel with n vertices and $n^2/4 + n - 2$ arcs. If n is odd, we can construct a funnel with $(n+1)(n-1)/4 + n - 2$ many arcs.

Proposition 2.3.1. *Let $D = (V, A)$ be a funnel. Then $|A| \leq |V|^2/4 + |V| - 2$.*

Proof. Let D be generated by Fork and Merge operations and let F be the auxiliary table used by them. Let $V = X \uplus Y$ where $\forall v \in X : F(v)$ and $\forall v \in Y : \neg F(v)$. Clearly, the vertices in X form an out-forest, while those in Y form an in-forest. This gives us at most $|X| - 1$ arcs between vertices in X, and at most $|Y| - 1$ arcs between vertices in Y. Furthermore, there are at most $|X| \cdot |Y|$ arcs from vertices in X to vertices in Y and we know from the construction that there are no arcs from Y to X. Hence, $|A| \leq |X| \cdot |Y| + |X| + |Y| - 2$. This value is maximized when $|X| = |Y| = |V|/2$, which gives us the bound $|A| \leq |V|^2/4 + |V| - 2$. □

If we consider that a DAG has at most $\binom{n}{2} = |V|(|V| - 1)/2$ arcs, we see from Proposition 2.3.1 that a funnel can have roughly half as many arcs as a DAG. This implies that funnels are not necessarily sparse (unlike trees).

We close this section with an overview of how we can characterize funnels by summarizing the results from Propositions 2.2.1, 2.2.3, 2.2.5 and 2.2.6.

Theorem 2.3.2. *Let $D = (V, A)$ be a DAG. The following statements are equivalent:*

1. *D is a funnel.*
2. *For every source s and every sink t in D every (s,t)-path has at least one private arc.*
3. *D can be generated by Fork and Merge operations.*
4. *For any vertex $v \in V : \text{indeg}(v) > 1 \Rightarrow \forall u \in \text{out}^*(v) : \text{outdeg}(u) \leq 1$.*
5. *No subgraph of D is contained in $\mathcal{F} = \{D_i\}_{i=0}^{\infty}$, where*

$$\begin{aligned} D_k &= (V_k, A_k), \\ V_k &= \{u_1, u_2, v_0, w_1, w_2\} \cup \{v_i\}_{i=1}^{k} \text{ and} \\ A_k &= \{(u_1, v_0), (u_2, v_0), (v_k, w_1), (v_k, w_2)\} \cup \{(v_i, v_{i+1})\}_{i=1}^{k-1}. \end{aligned}$$

6. *D does not contain D_0 or D_1 (defined above) as a topological minor.*

Chapter 3

Closeness to a Funnel

While sometimes one might be only interested in solving a problem where the input is a funnel, other times this restriction is too strong. However, it can nevertheless be the case that the digraphs of interest are "almost" funnels. Similarity measures are usually called distance parameters, providing some concrete definition of how far away a DAG is from a funnel. There are multiple ways of defining this distance, and in this section we study some of them.

Two of the parameters are rather canonical, namely arc- and vertex-deletion distance. The other two, funnel height and funnel depth, are based on the structure of a funnel. We also provide algorithms to compute each parameter.

3.1 Vertex-Deletion Distance

One of the most straightforward parameters to consider is vertex-deletion distance. The motivation behind such parameter is that if removing only a few vertices from a DAG turns it into a funnel, then the DAG itself should be structurally similar to a funnel. In other words, if a problem is easy on funnels, then one can hope that adding just a few vertices which violate funnel properties should not make the problem considerably harder. We now define this parameter formally.

Definition 3.1.1 (Vertex-deletion distance to a funnel)**.** Let D be a digraph and V be the smallest vertex-set such that $D \setminus V$ is a funnel. Then the vertex-deletion distance to a funnel of D is $|V|$.

Lewis and Yannakakis [LY80] proved a very general theorem about the NP-hardness of several vertex-deletion distance parameters. In order to understand the statement of the theorem we first define two concepts. A digraph property Π is said to be *nontrivial* if it is satisfied by infinitely many digraphs, and not satisfied also by infinitely many. The property is *hereditary* if, for any digraph satisfying the property, all induced subgraphs also satisfy the property.

The theorem of Lewis and Yannakakis [LY80] states that for any nontrivial and hereditary digraph property Π solving the vertex-deletion problem is NP-hard. Hence,

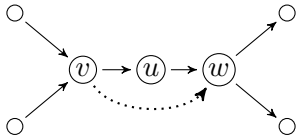


Figure 3.1: Example application of Reduction Rule 3.1.2. The dotted arc corresponds to the result of the contraction of (v, u).

it is NP-hard to compute the vertex-deletion distance to a funnel. For this reason it is interesting to consider FPT-time algorithms for this problem. We first propose a data reduction rule which simplifies the input digraph. It consists of removing vertices which cannot contribute to the solution. The reduction rule is illustrated in Figure 3.1.

Reduction Rule 3.1.2. If there is a path vuw in D with $\text{indeg}(u) = \text{outdeg}(u) = 1$, $\text{indeg}(w), \text{outdeg}(v) \leq 1$, then contract the arc (v, u).

Since we use Reduction Rule 3.1.2 again later on in a similar context, we prove here its correctness in a more general case. In order to prove that applying Reduction Rule 3.1.2 does not change the vertex-deletion distance to a funnel of a digraph, we first show that it does not destroy any forbidden topological minor F of a digraph class if the reduction rule is not applicable in F and the class in question is closed under *extrusion*. Extruding a vertex v with $\text{indeg}(v) = 0$ means adding a vertex u together with the arc (u, v). If $\text{outdeg}(v) = 0$ then we add the arc (v, u) instead. If $\text{indeg}(v) = 0 = \text{outdeg}(v)$ then we either add the arc (v, u) or (u, v), but not both. A digraph class is closed under extrusion if extruding any source or sink also produces a digraph of said class.

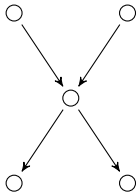
Lemma 3.1.3. *Let Π be a digraph property characterized by a set $\mathcal{F}$ of forbidden topological which is closed under extrusion and is such that Reduction Rule 3.1.2 is not applicable to any $F_i \in \mathcal{F}$. A digraph D satisfies Π if and only if after applying Reduction Rule 3.1.2 to D it satisfies Π.*

Proof. We first show that applying the reduction rule does not destroy any minor F_i. Let F be some forbidden topological minor for Π. Let D' be the resulting digraph after applying the reduction rule to D and let $H \subseteq D$ such that H is isomorphic to some subdivision of F. If u is not a vertex of H, then H is also a subgraph of D' and D' contains F as a topological minor.

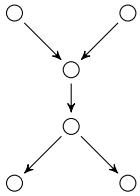
If the entire path vuw is contained in H, then it has to correspond to some subdivided arc of F, since Reduction Rule 3.1.2 is not applicable in F. Hence, the arc (v, w) in D' corresponds to one less subdivision of the same arc in F, and so D' also contains F as a minor.

Finally, if only vu or uw is present in H, then in D' there is a subgraph isomorphic to H which contains the vertices v and w and the arc (v, w). Hence, D' also contains F as a minor.

Now assume D' contains F as a topological minor. Let $H' \subseteq D'$ be isomorphic to some subdivision of F. If H' does not contain the arc (v, w), then it is also present in D.



(a) Single core



(b) Double core

Figure 3.2: Example of both families of forbidden subgraphs.

Otherwise, by subdividing this arc we obtain a digraph H which is a subgraph of D. Since H' contains F as a minor, so does H. □

We note that Reduction Rule 3.1.2 is not applicable in any forbidden topological minor for funnels, and so we can apply this lemma when proving the correctness of the reduction rule.

Proof (Correctness of Reduction Rule 3.1.2). We show that applying the reduction rule does not change the vertex-deletion distance to a funnel of a DAG. We first show that a vertex-deletion set for the reduced DAG is also a valid set for the original one, that is, it also produces a funnel.

Let $D = (V, A)$ be the initial digraph and $D' = (V', A')$ be the resulting digraph after applying the rule and U' be a vertex-set such that $D' \setminus U'$ is a funnel. If $\{v, w\} \cap U' = \emptyset$, then contracting the arc (u, w) in $D \setminus U'$ produces a digraph which is isomorphic to $D' \setminus U'$. By Lemma 3.1.3 this implies that $D \setminus U'$ is also a funnel. If $\{v, w\} \cap U' = \{v, w\}$, then u is an isolated vertex in $D \setminus U'$ and does not participate in any forbidden minor. If $\{v, w\} \cap U' \neq \{v\}$, then clearly $\hat{D} = D \setminus (U' \cup \{u\})$ is equal to $D' \setminus U'$ and, hence, is also a funnel. Adding back the vertex u together with the arc (u, w) to $\hat{D}$ will not produce a forbidden minor, since adding an arc from a source to another does not destroy the funnel property. Hence, $D \setminus U'$ is a funnel. An analogous argument holds when $\{v, w\} \cap U' \neq \{w\}$.

Now let $U \subseteq V$ be a vertex-set such that $D \setminus U$ is a funnel. If $u \notin U$ then $U \subseteq V'$ and $D' \setminus U$ is a subgraph of $D \setminus U$ and, thus, a funnel. Otherwise, set $U' \coloneqq (U \setminus \{u\}) \cup \{w\}$. We argue that $D' \setminus U'$ is a funnel. Clearly, $D'' = D \setminus (U \cup \{w\})$ is a funnel, since the class is closed under deletion. Furthermore, D'' is isomorphic to $D' \setminus U'$. Hence, $D' \setminus U'$ is also a funnel. □

Note that the correctness proof also works if the input is not a DAG. Since DAGs are characterized by a single forbidden topological minor (namely, a cycle of length two) and are closed under the extrusion of sources and deletion of vertices, we can apply Reduction Rule 3.1.2 due to Lemma 3.1.3.

We can now proceed to the algorithm which computes the vertex-deletion distance to a funnel of a DAG. If Reduction Rule 3.1.2 is not applicable, then we can identify two families of subgraphs where at least one vertex has to be removed. A digraph of

the first family, which will be referred to in this section as *single core*, is composed of a single vertex with in- and outdegree greater than one together with all of its neighbors.

A digraph of the second family, here called *double core*, contains two vertices u, v such that $\text{outdeg}(u) = \text{indeg}(v) = 1$, $\text{outdeg}(v) > 1$ and $\text{indeg}(u) > 1$.

Digraphs of both families are illustrated in Figure 3.2. Clearly, if a DAG contains either a single core or a double core as a subgraph then it is clearly not a funnel. Before proceeding to the algorithm, we also show that it is sufficient to look for subgraphs of these two families.

Lemma 3.1.4. *Let D be a DAG where Reduction Rule 3.1.2 is no longer applicable. Then D is a funnel if and only if D contains neither a single nor a double core as a subgraph.*

Proof. One can easily see that none of the digraphs of either family are funnels. Since being a funnel is an hereditary property, if D is a funnel then it does not contain any single or double core.

We show the other direction by contraposition. That is, we show that if Reduction Rule 3.1.2 is no longer applicable and D is not a funnel, then D has a single or double core. The idea is to use the properties from Proposition 2.2.1. Let $D = (V, A)$ be a DAG which is not a funnel where Reduction Rule 3.1.2 is not applicable. Since D is not a funnel, we know from Reduction Rule 3.1.2 that there is some vertex $v \in V$ such that $\text{outdeg}(v) > 1 \wedge \text{indeg}(v) > 1$ or $\text{indeg}(v) > 1 \wedge \exists u \in \text{out}^*(v) : \text{outdeg}(u) > 1$. If $\text{outdeg}(v) > 1 \wedge \text{indeg}(v) > 1$ then v together with its neighbors correspond to a single core. Otherwise if $\text{indeg}(v) > 1 \wedge \exists u \in \text{out}^*(v) : \text{outdeg}(u) > 1$ then we argue that u is the outneighbor of v. First note that $\text{outdeg}(v) = 1$, and so v has only one outneighbor u. Furthermore, if $\text{outdeg}(u) = 1$, then Reduction Rule 3.1.2 is applicable, a contradiction to the initial assumption for D. Since we know that there is some vertex in $\text{out}^*(v)$ with outdegree greater than one, $\text{outdeg}(u)$ has to be greater than zero. Thus $\text{outdeg}(u) > 1$ and v, u together with their neighbors form double core. □

Our algorithm for computing the vertex-deletion distance to a funnel of a DAG uses a search tree whose size is a function of the size k of the vertex-deletion set. The search tree is constructed as follows. First, note that in a single core we must either remove the central vertex, all but one of its inneighbors or all but one of its outneighbors. For a double core we also need to consider the possibility of removing either of the central vertices. In both cases, however, we decrease the parameter by at least one. Furthermore, if a single or double core has more than $k+1$ inneighbors, then we know it is not possible to remove all but one of them since this would require removing more than k vertices. This implies that the number of branches we have for each single and double core is bounded by a function in k. Formally, this leads to the following branching rule.

Branching Rule 3.1.5. If there is some vertex $v \in V$ such that $\text{indeg}(v) > 1$ and $\text{outdeg}(v) > 1$, then branch into three cases: Remove all but one inneighbor of v, remove all but one outneighbor of v or remove v. If there are two vertices $v, w \in V$ such that $\text{indeg}(v) > 1$, $\text{outdeg}(w) > 1$ and $(u, w) \in A$, then branch into four cases: Remove all but one inneighbor of v, remove all but one outneighbor of w, remove v or remove w.

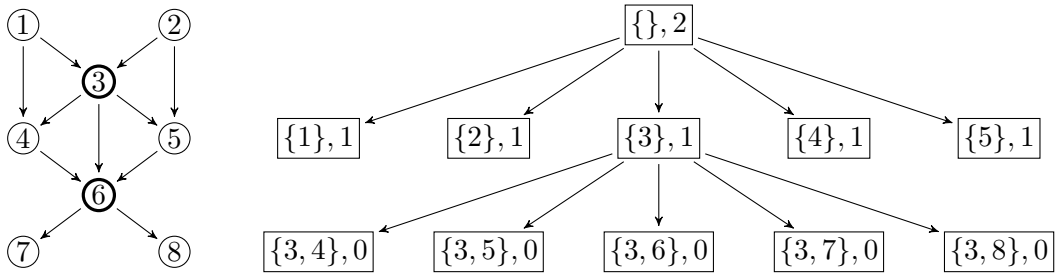


Figure 3.3: On the left side is the input DAG. Vertices which correspond to a vertex-deletion set are marked in bold (3 and 6). On the right side is a fragment of the search tree described in Theorem 3.1.6. Each box contains the current vertex-deletion set and the parameter k. In the first step, the single core centered at 3 was chosen. Note that the possibility of removing 6 was not considered on the first branching, but only after removing 3.

The search-tree is then obtained by exhaustively applying Branching Rule 3.1.5. In order to analyze the time complexity of the search-tree algorithm described above we first consider a simpler one. Instead of considering all in- and outneighbors of a vertex or vertex pair, we consider only two. The argument that at least one of the considered vertices needs to be removed still holds. Furthermore, the cases described above are also tested in this simplified variant. Hence, it is also correct and an upper bound of its running time is also an upper bound for the search tree constructed from Branching Rule 3.1.5. An example execution of the algorithm used in the proof is given by Figure 3.3.

Theorem 3.1.6. *Let $D = (V, A)$ be a DAG and $k \in \mathbb{N}$. Finding some $V' \subseteq V$ with $|V'| \leq k$ such that $D \setminus V'$ is a funnel can be done in $\mathcal{O}(6^k \cdot |V|^2)$ time.*

Proof. We describe an algorithm which finds such V' if one exists. It starts with an empty V'. Applying Reduction Rule 3.1.2 requires removing one vertex and two arcs, and adding an arc. Doing this for every vertex takes $\mathcal{O}(|V|^2)$ time.

If all degrees are precomputed, finding a single or double core can be done in linear time. If neither a single nor a double core exists, then Lemma 3.1.4 assures that the digraph is a funnel. We then output "`yes`" if $k \geq 0$. We output "`no`" whenever $k < 0$ during the branching.

If we find a single core, we pick two inneighbors and two outneighbors of the central vertex. We then branch on the possibilities of removing one of the neighbors or the central vertex, and adding it to V'. This amounts to five branches, each decreasing k by one. For double cores we pick the vertices analogously, but now we need to consider two central vertices, giving us six branches. This leads to a search tree with 6^k leaves. After removing a vertex we need to apply Reduction Rule 3.1.2 again. Removing a vertex takes at most $\mathcal{O}(|V|^2)$ operations. Thus, checking each leaf of the search tree can be done in $\mathcal{O}(6^k \cdot |V|^2)$. □

3.2 Arc-Deletion Distance

Similar to vertex-deletion distance, the arc-deletion distance to some class is also a canonical parameter to be considered. One motivation for considering both parameters is that deleting vertices is a more "aggressive" operation in the sense that the DAG is more strongly modified when compared to only deleting arcs. Hence, one can often expect to find more positive (e.g. FPT) results with respect to the weaker (and, thus, larger) parameter than with the stronger (and smaller) one.

In this section, we consider different algorithms which compute the arc-deletion distance to a funnel of a DAG. We start in Section 3.2.1 with a search tree strategy very similar to the one used in Section 3.1, obtaining again an FPT algorithm with respect to the solution size. Unlike the other algorithms in this section, this one works on digraphs as well. We then proceed to a constant-factor approximation algorithm in Section 3.2.2. Finally, we develop in Section 3.2.3 another FPT algorithm with respect to solution size, but with a better running time in comparison to the one in Section 3.2.1. This algorithm, however, has the disadvantage that it only works on DAGs.

3.2.1 An Arc Strategy

We first define formally what is the arc-deletion distance to a funnel of a digraph.

Definition 3.2.1 (Arc-deletion distance to a funnel)**.** Let D be a digraph and A the smallest arc-set such that $D \setminus A$ is a funnel. Then the arc deletion distance to a funnel of D is $|A|$.

We describe a branching strategy which is very similar to the one used in Section 3.1 for the search-tree algorithm for the vertex-deletion distance to a funnel of a DAG. We again consider two families of forbidden subgraphs and branch on all possibilities of eliminating these subgraphs. With some additional steps, we also obtain an algorithm which computes the arc-deletion distance to a funnel of a digraph. To achieve this, one needs to solve two problems simultaneously: Finding a feedback arc-set (that is, a set of arcs whose removal turns the digraph into a DAG) and destroying the forbidden topological minors for funnels. A simple approach would be to try all possible feedback arc sets and then compute the distance to a funnel for each of them. This would not, however, yield an FPT-time algorithm since there can be $|V|^k$ many feedback arc-sets of size k. In order to solve this problem in FPT-time, we first try to remove all arcs necessary to destroy the forbidden minors for funnels. Then we compute a feedback arc-set. We only need to argue why it is not necessary to consider all arc-sets which destroy the forbidden minors for funnels in the digraph. We first show that if a digraph D does not contain any of the forbidden minors for funnels, then all cycles are disjoint, that is, they do not share any arcs.

Lemma 3.2.2. *Let D be a digraph contains neither a single core nor a double core as a topological minor. Then every arc of D is contained in at most one cycle.*

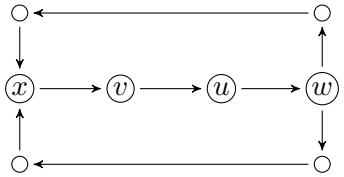


Figure 3.4: Whenever two cycles share an edge, a double core is present as a topological minor. In this case, the path $xvuw$ corresponds to this minor.

Proof. Assume towards a contradiction there are two distinct cycles C_1, C_2 such that both contain the arc (v, u). Since C_1, C_2 are distinct, it follows that $\exists w \in \text{out}^*(u) : \text{outdeg}(w) > 1$, as otherwise there would be only one cycle from u to itself. Similarly, $\exists x \in \text{in}^*(v) : \text{indeg}(v) > 1$. Thus, the path $x \dots w$ has a single or double core as minor, contradicting our assumption that this is not the case. □

Lemma 3.2.2 implies that a digraph attains a very simple structure if it does not contain the forbidden subgraphs for funnels. This structure allows us to find a feedback arc-set in polynomial time, a problem which is NP-hard in general. In order to find cycles more easily, we apply the following reduction rule.

Reduction Rule 3.2.3. If there is some vertex v with $\text{indeg}(v) = 0$ or $\text{outdeg}(v) = 0$, then remove v.

The basic idea of the polynomial-time algorithm for computing a feedback arc-set is to find a cycle and remove one of its arcs. Since every arc is in at most one cycle, it does not matter for the solution size which arc from a cycle we remove. Hence, we can greedily find cycles and remove arcs with a depth-first search, as formalized below.

Lemma 3.2.4. *Let $D = (V, A)$ be a digraph which does not contain single or double cores as topological minors. Then one can find a minimum-sized arc-set $A' \subseteq A$ such that $D \setminus A'$ is a DAG in $\mathcal{O}(|V|^2 + |V| \cdot |A|)$ time.*

Proof. First, exhaustively apply Reduction Rule 3.2.3. We do this in $\mathcal{O}(|V| + |A|)$ time by starting at the sources and sinks, removing them and proceeding to their neighbors. In order to find a cycle, we can apply depth-first search on a vertex. If no cycle exist, output $\emptyset$. Otherwise, pick some arc of this cycle into the solution, remove it from the digraph and apply Reduction Rule 3.2.3 again. We repeat this procedure at most $|V|$ times, yielding a running time of $\mathcal{O}(|V| \cdot (|V| + |A|)$.

From Lemma 3.2.2 we know that the cycles do not share any arc, and so for each cycle we have to pick exactly one of its arcs in order to destroy it. Since it is not possible to destroy multiple cycles by removing a single arc, it is irrelevant which arc of the cycle is taken into the solution. □

In order to complete the definition of the FPT algorithm for computing the arc-deletion distance to a funnel of a digraph, we need to describe how we destroy the topological minors for funnels. The idea is to identify single and double cores, and then branch on all possibilities of destroying them.

Branching Rule 3.2.5. If there is some vertex $v \in V$ with $\text{indeg}(v) > 1$ and $\text{outdeg}(v) > 1$, then branch into all possibilities of deleting all but one of its outgoing arcs or all but one of its incoming arcs.

If there are two vertices $v, u \in V$ with $(v, u) \in A$, $\text{indeg}(v) > 1$ and $\text{outdeg}(u) > 1$, then branch into all possibilities of deleting all but one of the outgoing arcs of u, all but one of the incoming arcs of u or the arc (v, u).

We can now combine the previous results, building an FPT algorithm with respect to solution size for computing the arc-deletion distance to a funnel of a digraph. This is done on the theorem below. We will also use Reduction Rule 3.1.2, and so we restate it here and prove its correctness for the arc-deletion problem.

Reduction Rule (repeated)**.** If there is a path vuw in D with $\text{indeg}(u) = \text{outdeg}(u) = 1$ and $\text{indeg}(w), \text{outdeg}(v) \leq 1$, then contract the arc (v, u).

Proof (Correctness of Reduction Rule 3.1.2). Let $D' = (V', A')$ be the digraph obtained from $D = (V, A)$ after applying the reduction rule once. Let Π be a hereditary digraph property which is characterized by a set $\mathcal{F}$ of forbidden topological minors such that Reduction Rule 3.1.2 is not applicable to any digraph in F. Let B' be an arc-set such that $D' \setminus B'$ satisfy Π. We show that there is some $B \subseteq A$ such that $D \setminus B$ also satisfies Π and $|B| \leq |A|$.

If $(v, w) \notin B'$, then by Lemma 3.1.3 $D \setminus B'$ satisfies Π. Otherwise, $D'' = (D \setminus \{u\}) \setminus B'$ is isomorphic to $D' \setminus B'$. By assumption, extruding u from w in D'' does not destroy the property Π. Doing so produces a digraph which is isomorphic to $D \setminus B$, where $B = (B' \setminus \{(v, w)\}) \cup \{(v, u)\}$. Hence, $D \setminus B$ satisfies Π as well.

Now assume there is some $B \subseteq A$ such that $D \setminus B$ satisfies Π. If no arc of u is in B, then by Lemma 3.1.3 $D' \setminus B$ also satisfies Π. Otherwise, assume $(v, u) \in B$ and let $B' = (B \setminus \{(v, u)\}) \cup \{(v, w)\}$. Because Π is hereditary, $D'' = (D \setminus B) \setminus \{u\}$ also satisfies the property. And because D'' is isomorphic to $D' \setminus B'$, this implies that $D' \setminus B'$ also satisfies Π. The case where $(u, v) \in B$ follows analogously. Thus, we conclude that applying Reduction Rule 3.1.2 to a digraph does not change its arc-deletion distance to Π. □

Theorem 3.2.6. *Let $D = (V, A)$ be a digraph. One can find an arc-set $A' \subseteq A$ such that $D \setminus A'$ is a funnel in $\mathcal{O}(5^d \cdot (|V|^2 + |V| \cdot |A|))$ time, where $d = |A'|$.*

Proof. We first delete all single and double cores from D by applying Reduction Rule 3.1.2 exhaustively and then using Branching Rule 3.2.5. In the worst case we need to branch into five possibilities, yielding a search tree with 5^k leaves. This search tree clearly finds the smallest set A' such that $D \setminus A'$ contains neither a single nor double core as a subgraph. From Lemma 3.1.4 we also know that $D \setminus A'$ also contains neither single nor double cores as topological minors.

Whenever we reach a leaf of the search tree, we compute a feedback arc set for the resulting digraph. From Lemma 3.2.4 we know this is doable in $\mathcal{O}(|V|^2 + |V| \cdot |A|)$ time. Since we might have to compute this for every leaf, the total running time of the algorithm is $\mathcal{O}(5^k \cdot (|V|^2 + |V| \cdot |A|))$. □

We note here that, unlike the vertex-deletion problem, we do not know whether computing the arc-deletion distance to a funnel of a digraph is NP-hard. The problem does not seem to have enough structure to allow the construction of gadgets, but at the same time it is not local enough to allow for some greedy algorithm to compute an optimal solution. Hopefully, the algorithms in subsequent subsections provide some insight to the complexity of the problem, yet a concrete answer is unknown to us.

3.2.2 A Factor-3 Approximation

When computing an arc-set whose removal turns the DAG into a funnel it is not always necessary to find the smallest such set. Sometimes it suffices to have a arc-set which is small enough. For example, when the arc-set is used in order to solve another problem having a smaller arc-set often improves the running time but is not required for correctness. By not requiring the optimal solution to be found, it is often possible to obtain an algorithm with a better running time. In this section we develop a linear-time factor-3 approximation for computing the arc-deletion distance to a funnel of a DAG.

Our approximation algorithm for arc-deletion distance to a funnel works in two steps. First, we assign each vertex v a Fork or Merge label. This decides whether v can have indegree or outdegree greater than one. For this labeling we use a greedy strategy, that is, we try to minimize the number of arcs to be removed when only considering v. This guarantees that, if the approximation algorithm guesses wrongly, in the optimal solution many arcs need to be removed. We use this property later on to show that the approximate solution is only by a constant factor worse than the optimal solution. Formally, we assign a label to a vertex v using the following rule.

$$\text{label}_D(v) := \begin{cases} \textsc{Fork}, & \text{if } \text{outdeg}_D(v) > \text{indeg}_D(v) \\ \textsc{Merge}, & \text{if } \text{outdeg}_D(v) < \text{indeg}_D(v) \\ \textsc{Fork}, & \text{if } \text{outdeg}_D(v) = \text{indeg}_D(v) \wedge \exists u \in \text{in}(v) : \text{label}_D(u) = \textsc{Fork} \\ \textsc{Merge}, & \text{otherwise.} \end{cases}$$

Since we only need the incoming neighbors to break ties, the label of each vertex can be computed by following some topological ordering of the DAG. After assigning labels to all vertices, we need to guarantee that the digraph is compatible with our choice. In other words, we not only need to remove outgoing arcs from Merge vertices and incoming from Fork vertices, but also arcs from Merge to Fork vertices. This step can be done in a fairly arbitrary way. If a Fork vertex has two incoming arcs from other Fork vertices, then it is irrelevant which of the arcs is kept. Likewise, arcs going from a Merge to another Merge vertex can also be kept in an arbitrary way without making the solution any worse.

The approximation algorithm for arc-deletion distance is described in Algorithm 1. We assume the label function has being computed for all vertices and its value is stored in an array. An example of the execution of the algorithm is given at Figure 3.5.

Algorithm 1 Approximation for the arc-deletion distance to a funnel of a DAG.

```
1: function ArcDeletionSet(DAG D = (V, A))
2:     A' := ∅
3:     for all v ∈ V do
4:         w := ⊥                          ▷ The arc (v, w) will be kept if w ≠ ⊥.
5:         if label(v) = Merge then
6:             for all u ∈ out(v) do
7:                 if label(u) = Merge then
8:                     w := u              ▷ A single arc to another Merge vertex can be kept.
9:             X := {(v, u) | u ∈ out(v) ∧ u ≠ w}
10:        else if label(v) = Fork then
11:            for all u ∈ in(v) do
12:                if label(u) = Fork then
13:                    w := u              ▷ A single arc from another Fork vertex can be kept.
14:            X := {(u, v) | u ∈ in(v) ∧ label(u) = Fork ∧ u ≠ w}
15:        A' := A' ⊎ X
16:    return A'
```

Before we analyze the approximation factor, we first show an interesting property of the approximation algorithm. We show that if the vertices are labeled as in an optimal solution, then the algorithm above also outputs an optimal arc-set. In a sense, this means that the difficult part of the problem lies in finding the correct labeling. Once this labeling is found, we can compute an arc-deletion set in linear time.

Proposition 3.2.7. *Let $D = (V, A)$ be a DAG and $A' \subseteq A$ a minimum-sized arc-set such that $D' = D \setminus A'$ is a funnel. If $\forall v \in V : \text{label}_D(v) = \text{label}_{D'}(v)$, then $|\texttt{ArcDeletionSet}(D)| = |A'|$.*

Proof. We first consider two simple cases for an arc $(v, u) \in A$. We argue that in both cases any optimal solution and `ArcDeletionSet`take the same decision with respect to this arc.

The first case is when $\text{label}(v) = \textsc{Merge}$ and $\text{label}(u) = \textsc{Fork}$. This arc has to be both in A' as well as in the solution given by `ArcDeletionSet`, which we call from now on B. It is clearly in B since it was added to the solution on Line 9. Since $\text{label}(v) = \textsc{Merge}$ we know $\exists w \in \text{in}^*_{D'}(v) : \text{indeg}_{D'}(w) > 1$. Because $\text{label}(u) = \textsc{Fork}$, then necessarily $\text{outdeg}_{D'}(u) > 1$, otherwise u would also be marked as Merge. Hence, $(v, u) \in A'$, otherwise a forbidden subgraph would be present and D' would not be a funnel. The other simple case is when $\text{label}(v) = \textsc{Fork}$ and $\text{label}(u) = \textsc{Merge}$. Clearly removing this arc will not destroy any forbidden subgraph, since $\forall w \in \text{in}^*_{D'}(v) : \text{label}(w) = \textsc{Fork}$. Since the approximation algorithm does not remove the arc, it is neither present in B nor in A'.

For the remaining cases we cannot guarantee that exactly the same decision was taken with respect to the arc. What we do instead is to consider a set of arcs which is

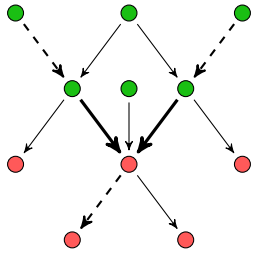


Figure 3.5: Example of the execution of `ArcDeletionSet`. Vertices which received the label FORK are green, and those who received the MERGE label are red. The approximation algorithm will return the dashed arcs, while there is an optimal solution (bold arcs in the center) of size two.

not removed and argue that exactly one of them is not removed by the approximation, and at most one is not removed by an optimal solution.

If $\text{label}(v) = \textsc{Merge} = \text{label}(u)$, let $X = \{(v, w) \mid w \in \text{out}_D(v)\}$. We know that $|X \setminus A'| \leq 1$, since otherwise $\text{outdeg}_{D'}(v) > 1$ which contradicts $\text{label}(v) = \textsc{Merge}$. We also know that the algorithm will set w to u in Line 8, and potentially to another value later. In either case, the arc (v, w) will not be present in B. Thus, $|X \setminus B| = 1 \geq |X \setminus A'|$. An analogous argument holds for the case where $\text{label}(v) = \textsc{Fork} = \text{label}(u)$.

Since those were all cases and in all of them `ArcDeletionSet`deletes at most as many arcs as the optimal solution, we can conclude that $|B| = |A'|$. □

We now give a guarantee of the quality of the solution found by `ArcDeletionSet`. Since the algorithm does not always yield an optimal solution, it is useful to have an upper-bound of how much larger the solution found can be in comparison to an optimal solution. This can then be used in other algorithms which rely on the fast computation (e.g., polynomial time) of a arc-deletion set which is not too large (e.g., off by a constant factor).

Lemma 3.2.8. *Let $D = (V, A)$ be a DAG, $A' \subseteq A$ be a minimum-sized arc-set such that $D \setminus A'$ is a funnel, and B be the output of* `ArcDeletionSet`*. Then $|B| \leq 3 \cdot |A'|$.*

Proof. We prove this by mapping each arc removed by the approximation to one arc from the optimal solution. We then show that at most three arcs are mapped to the same arc. That is, we construct a function $f : B \to A'$ such that $\forall a \in A' : \left|f^{-1}(a)\right| \leq 3$, where $f^{-1}(a) = \{x \in B \mid f(x) = a\}$ is the inverse image of f. We do this by iterating through all vertices and analyzing which arcs are removed by the approximation.

Let $v \in V$ be a vertex. We first observe that if $\text{label}(v) = \textsc{Fork}$, then the approximation only removes incoming arcs; when $\text{label}(v) = \textsc{Merge}$ it only removes outgoing arcs. Even though other arcs may be removed in the end, this only happens when the algorithm reaches other vertices. Hence, we do not need to consider all adjacent arcs of a vertex. We then make a case distinction based on the in- and outdegree of a vertex.

If $\text{outdeg}(v) \neq \text{indeg}(v)$ and both are greater than one, then an optimal solution has to remove at least $\min\{\text{indeg}(v) - 1, \text{outdeg}(v) - 1\}$ many arcs. The approximation

(a) The arc (v, u) is removed both by the approximation as well as in an optimal solution.

(b) The arc (v, u) is not removed by the approximation if $\text{label}(u) = \textsc{Merge} = \text{label}(v)$.

Figure 3.6: Argumentation of why `ArcDeletionSet`is at most a factor-three approximation. Even if all dashed arcs are removed, a factor of three is achieved. The dotted arc corresponds to an optimal solution.

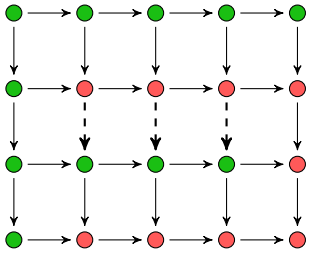
removes at most $\min\{\text{indeg}(v), \text{outdeg}(v)\}$ many, that is, at most one more than optimal. Hence, at most two arcs from the approximation are mapped to the same optimal arc, while the rest is mapped one-to-one to the remaining arcs from the optimal solution.

If $\text{outdeg}(v) \neq \text{indeg}(v)$ and one of the degrees equals one, then an arc is removed only if the labels of the vertices differ. Consider the case where $\text{outdeg}(v) = 1$, which implies $\text{indeg}(v) > 1$ and $\text{label}(v) = \textsc{Merge}$ and assume the arc was removed. Then $\text{label}(u) = \textsc{Fork}$, for $u \in \text{out}(v)$, and $\text{outdeg}(u) > 1$. If additionally $\text{indeg}(u) > 1$, then the mapping of this arc is covered by the previous case and we do not need to consider it now. Otherwise, if $\text{indeg}(u) = 1$, then $\text{in}^*[v] \cup \text{in}^*[u]$ forms a forbidden subgraph for funnels, and so at least one of its arcs is removed by the optimal solution. When considering v and u, the approximation only removes the arc (v, u). Hence, we can map (v, u) to one of the arcs removed by the optimal solution. The case where $\text{indeg}(v) = 1$ and $\text{label}(v) = \textsc{Fork}$ works analogously.

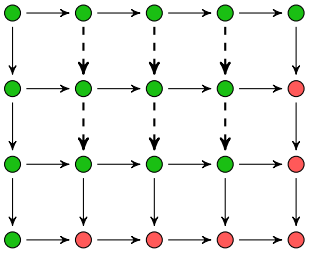
If $\text{outdeg}(v) = \text{indeg}(v)$, then we need to consider two further cases. First, if $\text{label}(v) = \textsc{Fork}$, we know by the definition of label that $\exists u \in \text{in}(v) : \text{label}(u) = \textsc{Fork}$. Hence, the arc (u, v) will not be removed by the approximation, and so it removes exactly $\text{indeg}(v) - 1$ arcs, which is the minimum amount of arcs that must be removed from v. Thus, each arc removed by the approximation is mapped to a different optimal arc. In the second case $\text{label}(v) = \textsc{Merge}$. The optimal solution has to remove at least $\text{outdeg}(v) - 1$ many arcs, whereas the approximation removes at most $\text{outdeg}(v)$. Hence, we map two arcs to the same one from the optimal solution, while the rest is mapped one-to-one to the remaining arcs.

In every case, an arc of the optimal solution receives at most four arcs from the approximation: Two for each endpoint. However, we argue that the case where a total of four arcs are mapped to the same one never happens.

Assume towards a contradiction that there is some arc $(v, u) = a \in A'$ such that $\left|f^{-1}(a)\right| = 4$. First note that there are only two cases where two arcs from the approximation are mapped to the same optimal arc: When in- and outdegree are different and greater than one; or when they are equal and the label is Merge. In both cases this only happens when all considered arcs are removed (i.e., all outgoing or all incoming). If the



(a) Optimal labeling for a grid with five columns and four rows. The optimal solution consist of removing three arcs.



(b) Labeling found by the approximation algorithm. A total of six arcs are removed by the algorithm.

Figure 3.7: Example of a DAG where the approximate solution is two times as large as an optimal solution. Green vertices receive the label FORK, while red ones receive MERGE. The solution of each algorithm is given by the dashed arcs.

in- and outdegree differ, then either both the optimal solution and the approximation remove incoming arcs or both removing outgoing arcs, as doing otherwise would result in a one-to-one mapping. Hence, only the cases where $\text{label}(v) = \text{MERGE}$ are relevant. These cases are illustrated in Figure 3.6.

If $\text{label}(u) = \text{FORK}$, then $(v, u) \in B$ as well. Thus, it is mapped to itself both when considering v as well as u. Since we are counting this mapping twice, $\left|f^{-1}(a)\right| \leq 3$.

If $\text{label}(u) = \text{MERGE}$, then not all outgoing arcs from v are removed, since keeping (v, u) is allowed. Hence $\left|f^{-1}(a)\right| \leq 3$.

In all cases $\left|f^{-1}(a)\right| \leq 3$, so $|B| \leq 3 \cdot |A'|$. □

Theorem 3.2.9. *Let $D = (V, A)$ be a DAG. An arc-set $A' \subseteq A$ of size $|A'| \leq 3d$ such that $D \setminus A'$ is a funnel can be computed in $\mathcal{O}(|V| + |A|)$ time, where d is the arc-deletion distance to a funnel of D.*

Proof. We start by computing a topological ordering of the vertices of D. This can be done in $\mathcal{O}(|V| + |A|)$. We then apply `ArcDeletionSet`, obtaining the arc-set A'. Since this algorithm iterates through all vertices exactly once, and for each vertex v the number of steps required is linear on its degree, it runs in $\mathcal{O}(|V| + |A|)$ time. If we implement label as an array, we can decide in constant time what is the label of a vertex. By Lemma 3.2.8 we know $|A'| \leq 3d$. Hence, the problem of determining arc-deletion distance to a funnel of a DAG admits a factor-3 approximation computable in $\mathcal{O}(|V| + |A|)$ time. □

Even though we showed the approximation algorithm yields a solution which is at most three times larger than an optimal one, it is still unclear whether this bound is sharp. That is, is there any DAG where a factor of three is indeed achieved? While we do not know the answer to this question, we provide in Example 3.2.1 a way of constructing arbitrarily large DAGs where a factor of two is achieved by the approximation.

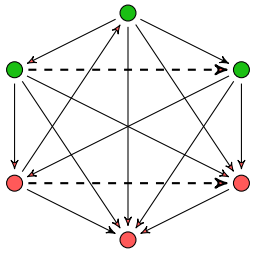


Figure 3.8: In tournament DAGs, the approximation algorithm finds an optimal solution. In this case, only two dashed arc are removed by the approximation. Green vertices receive the label Fork, while red ones receive the label Merge.

Example 3.2.1. Let $D = (V, A)$ be a directed grid with width w and height h. The grid is defined as follows

$$V = \{1, 2 \dots w\} \times \{1, 2 \dots h\} \text{ and}$$
$$A = \{((x, y), (x + 1, y)) \mid (x, y) \in V, x < w\} \cup \{((x, y), (x, y + 1)) \mid (x, y) \in V, y < h\}.$$

Directed grids constructed in this form have $(w-2)(h-2)$ vertices with in- and outdegree equal to two. Since each arc can destroy at most two forbidden subgraphs, we need to remove at least $\lceil (w-2)(h-2)/2 \rceil$ arcs in a grid. By removing $(w-2)$ arcs of each second row, we destroy all single-cores of the grid and require only $\lceil (w-2)(h-2)/2 \rceil$ arcs (refer to Figure 3.7). Because this amount equals the lower bound, it is also the size of an optimal solution.

The approximation algorithm, however, will attribute every single-core in the grid the label Fork. It will then have to remove one incoming arc of every single core, totaling $(w-2)(h-2)$ removed arcs. Hence, on grids the approximation removes twice as many arcs as an optimal solution.

While on directed grids the approximation factor is not very good, on tournament DAGs `ArcDeletionSet`always finds an optimal solution. This is shown on the example below (see Figure 3.8).

Example 3.2.2. Let $D = (V, A)$ be a tournament DAG with n vertices, where n is even. That is,

$$V = \{1, 2, \dots n\} \text{ and}$$
$$A = \{(v, u) \mid 1 \leq v \leq u \leq n\}.$$

From Proposition 2.3.1 we know a funnel with n vertices has at most $n^2/4 + n - 2$ arcs. Since a tournament DAG has $n(n-1)/2$ arcs, the arc-deletion distance to a funnel of D is at least $n(n-1)/2 - n^2/4 - n + 2 = (n^2 - 6n + 8)/4$ arcs. The approximation algorithm sets the label of the first $\lceil n/2 \rceil$ to Fork since their indegrees is not larger than their outdegrees. The remaining vertices receive the label Merge. The number of

arcs removed by the approximation is then given by

$$\sum_{v=3}^{\lceil n/2 \rceil} (v-2) + \sum_{v=\lceil n/2 \rceil + 1}^{n-2} (n-v-1) = \frac{n^2 - 6n + 8}{4}.$$

Hence, the solution from the approximation is optimal. The case where n is odd works similarly.

3.2.3 A Labeling Strategy

Other than the approximation algorithm from Section 3.2.2, the FPT algorithm in Section 3.2.1 does not label vertices and branches instead on the arcs that need to be removed. However, from Proposition 3.2.7 we know we actually only need to correctly label the vertices in order to find an optimal arc-deletion set. In a sense, the difficulty of the problem lies in finding this labeling, and not in removing the arcs themselves.

What we now do is to consider another algorithm which tries different labelings of the vertices. That is, we construct a function $L : V \to \{\textsc{Fork}, \textsc{Merge}\}$ and then apply `ArcDeletionSet` to it. A naive approach would be to guess, for each vertex, whether it is a Fork or a Merge vertex. This would imply testing $2^{|V|}$ different labelings, which would not give us an FPT algorithm with respect to the arc-deletion distance to a funnel of the input DAG.

In order to again bound the size of the search-tree with a function on the arc-deletion distance d to a funnel, we need to guarantee that, whenever the algorithm needs to branch, then, regardless of the decision it makes, some arc has to be removed. This implies that the search-tree has a height upper-bounded by d, and if the number of branches per step is constant, we obtain again an FPT algorithm for computing the arc-deletion distance to a funnel of a DAG.

One advantage of using labels instead of simply removing arcs is to gain certain local information about the vertices. That is, if we know that a vertex has a Merge label, then, after removing the arc-set found, all of its successors must also have a Merge label. This locality allows us to define certain data reduction rules which improve the running time of the algorithm. We first consider a reduction rule which allows us to label vertices without removing any arcs. The main idea is to consider what the algorithm `ArcDeletionSet` does when removing arcs from each vertex. Since Proposition 3.2.7 tells us that the algorithm computes an optimal arc-set if the labeling is optimal, we basically need to argue that the labels we set with this reduction rule do not increase the number of arcs removed by `ArcDeletionSet`. The various cases from Reduction Rule 3.2.10 below are illustrated in Figure 3.9.

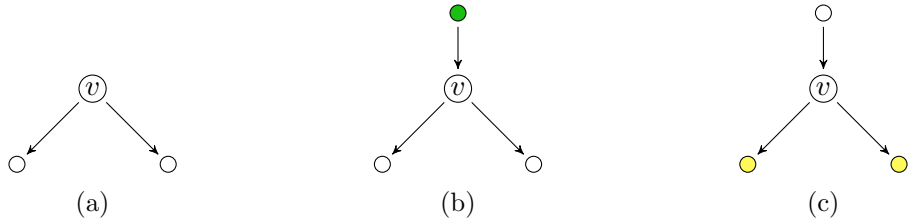


Figure 3.9: The three cases considered by Reduction Rule 3.2.10 when attributing the label Fork to a vertex v. Green vertices already have the label Fork, and yellow ones already have some label. In (a) we have $\text{indeg}(v) = 0$; in (b) v has only one inneighbor, and it has the label Fork; in (c) all labels of the outneighbors are known. The cases for setting $L(v) \coloneqq$ Merge are analogous.

Reduction Rule 3.2.10 (Set Label)**.** Let $v \in V$ be a vertex. Set $L(v) \coloneqq$ Fork if at least one of the following is true:

- $\text{indeg}(v) = 0$,
- $\text{indeg}(v) = 1$, $u \in \text{in}(v)$ and $L(u) =$ Fork, or
- $\text{outdeg}(v) > 1$, $\text{indeg}(v) = 1$ and $\forall u \in \text{out}(v) : L(u) \neq \bot$.

Set $L(v) \coloneqq$ Merge if at least one of the following is true:

- $\text{outdeg}(v) = 0$,
- $\text{outdeg}(v) = 1$, $u \in \text{out}(v)$ and $L(u) =$ Merge, or
- $\text{outdeg}(v) = 1$, $\text{indeg}(v) > 1$ and $\forall u \in \text{in}(v) : L(u) \neq \bot$.

Proof (Correctness of Reduction Rule 3.2.10). Recall the definition of the label function from Section 3.2.2, restated below.

$$\text{label}_D(v) \coloneqq \begin{cases} \text{Fork}, & \text{if } \text{outdeg}_D(v) > \text{indeg}_D(v) \\ \text{Merge}, & \text{if } \text{outdeg}_D(v) < \text{indeg}_D(v) \\ \text{Fork}, & \text{if } \text{outdeg}_D(v) = \text{indeg}_D(v) \wedge \exists u \in \text{in}(v) : \text{label}_D(u) = \text{Fork} \\ \text{Merge}, & \text{otherwise.} \end{cases}$$

Clearly, in a funnel the function label attributes every source a Fork label and every sink a Merge label. Since destroying sinks and sources is not possible, Reduction Rule 3.2.10 labels these vertices optimally.

Let v be a vertex with $\text{indeg}(v) = 1$, let $u \in \text{in}(v)$ be its only predecessor and assume $L(u) =$ Fork. If we set $L(v) \coloneqq$ Fork, then `ArcDeletionSet` will not remove any arc when considering v. If some outgoing arc (v, w) is removed, then necessarily $L(w) =$ Fork. Hence, if we instead set $L(v) \coloneqq$ Merge we also need to remove this arc, and potentially more. This implies that it is never worse to set $L(v) \coloneqq$ Fork in this case. An analogous argument holds for the case where $\text{outdeg}(v) = 1$ and $L(u) =$ Merge for the only successor u of v.

Finally, let v be a vertex where $\text{outdeg}(v) = 1$, $\text{indeg}(v) > 1$, and $\forall u \in \text{in}(v) : L(u) \neq \bot$. Since, by assumption, all outneighbors of v already have their labels set and satisfied, we only need to consider the label of v and of its only predecessor u. If $L(u) = \textsc{Fork}$ in an optimal solution, then we know by the previous case that it is optimal to set $L(v) \coloneqq \textsc{Fork}$. If $L(u) = \textsc{Merge}$ in an optimal solution, then we need to remove the arc (u, v) or some outgoing arc of v. That is, we need to remove at least one arc of v. By setting $L(v) \coloneqq \textsc{Fork}$, we know we need to remove exactly one arc of v. Hence, doing so is optimal. An analogous argument also holds for the last case where we set $L(v) \coloneqq \textsc{Merge}$. □

In order to simplify the branching rule defined later, we describe a partial simulation of `ArcDeletionSet` based on the assigned labels, but which does not necessarily satisfy the label of a vertex. We basically remove all arcs that would be removed by `ArcDeletionSet`, ignoring arcs from and to vertices which currently have no label.

Reduction Rule 3.2.11 (Satisfy Label)**.** Let v be some vertex where $L(v) = \textsc{Fork}$ and $\text{indeg}(v) > 1$. If $\exists u \in \text{in}(v) : L(u) = \textsc{Fork}$, then take the arcs $\{(x, v) \mid x \in \text{in}(v) \wedge x \neq u\}$ into the solution. Otherwise, take $\{(x, v) \mid x \in \text{in}(v) \wedge L(x) = \textsc{Merge}\}$ into the solution.

Let v be some vertex where $L(v) = \textsc{Merge}$ and $\text{outdeg}(v) > 1$. If $\exists u \in \text{out}(v) : L(u) = \textsc{Merge}$, then take the arcs $\{(v, x) \mid x \in \text{out}(v) \wedge x \neq u\}$ into the solution. Otherwise, take $\{(v, x) \mid x \in \text{out}(v) \wedge L(x) = \textsc{Fork}\}$ into the solution.

Clearly, the arcs removed by Satisfy Label would also be removed by `ArcDeletionSet` if all vertices had a label. Hence, if the labels are correct, then Satisfy Label only removes arcs that are present in an optimal arc-deletion set.

We can now proceed to the main part of the search-tree algorithm for computing the arc-deletion distance to a funnel of a DAG, namely the branching rule. As before, we identify vertices where we always need to remove at least one arc in order to turn the input DAG into a funnel. The crucial difference is that now we might know the labels of the neighbors of a vertex, allowing us to consider less possibilities to branch into. In particular, if we know all the labels of the inneighbors of a vertex, we only need to branch over the label and the outgoing arcs. We again need to consider single cores for branching, yet due to the previously defined data reduction rules we no longer need to consider double cores directly. Indeed, we can optimally decide on the label of one of the vertices in the double core through the data reduction rules.

The branching rule is divided into two parts. First, we branch on the label a vertex receives, without removing any arcs. It is important, however, to only take vertices where we necessarily need to remove at least one arc, otherwise we would not be able to bound the size of the search tree. The second rule consists of branching, if necessary, on the arcs that need to be removed from a vertex. Because we now know the labels of certain vertices, we might be able to optimally decide without further branching which arcs to remove after setting the label of a vertex.

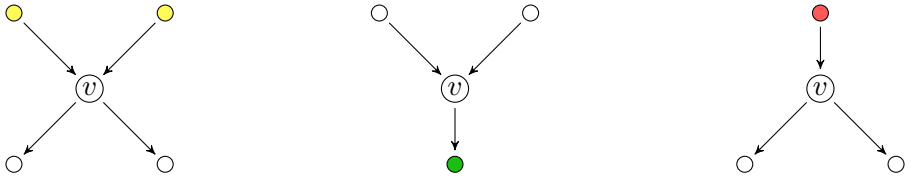


Figure 3.10: The three cases considered by Branching Rule 3.2.12 when branching on the label of a vertex v. Green vertices have the label Fork, red ones have the label Merge and yellow vertices have any label. In (a) we know the labels of the inneighbors of v, and thus do not require further branching if we set $L(v) \coloneqq$ Fork. In (b) we need to remove the only outgoing arc of v if we set $L(v) =$ Merge, and all but one incoming arc if we set $L(v) =$ Fork. Case (c) is analogous to (b).

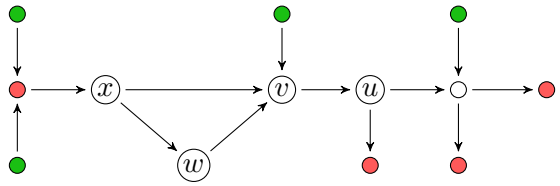


Figure 3.11: A DAG where Satisfy Label and Set Label are not applicable. Label Branch cannot be applied to v since u does not have a label, yet it can be applied to $x \in \text{in}^*(w)$.

Branching Rule 3.2.12 (Label Branch)**.** If there is some vertex v such that $\forall w \in \text{in}(v) : L(w) \neq \bot$ or $\exists w \in \text{in}(v) : L(w) =$ Fork, branch into two possibilities: Set $L(v) \coloneqq$ Fork; Set $L(v) \coloneqq$ Merge.

If there is some vertex v such that $\forall w \in \text{out}(v) : L(w) \neq \bot$ or $\exists w \in \text{out}(v) : L(w) =$ Merge, branch into two possibilities: Set $L(v) \coloneqq$ Fork; Set $L(v) \coloneqq$ Merge.

Branching Rule 3.2.13 (Arc Branch)**.** If there is some vertex v with $L(v) =$ Fork and $\text{indeg}(v) > 1$, branch into all possibilities of removing all but one incoming arc of v. If there is some vertex v with $L(v) =$ Merge and $\text{outdeg}(v) > 1$, branch into all possibilities of removing all but one outgoing arc of v.

From the definition of Label Branch it is, at first glance, unclear that we actually destroy all forbidden topological minors in the DAG. In particular, the branching rule requires certain labels of the neighbors of a vertex to be known, but this information is not necessarily available for every vertex in every forbidden minor. We argue that, if the label of some inneighbor u of a vertex is not known, then we can apply the branching rule on some vertex in $\text{in}^*(u)$. Since we can label all sources with Set Label, we necessarily arrive at some vertex where the label of all inneighbors is known, and then we can apply Label Branch. This is formalized in the lemma below.

Lemma 3.2.14. *Let D be a DAG. If Branching Rules 3.2.12 and 3.2.13 and Reduction Rules 3.2.10 and 3.2.11 are not applicable, then D is a funnel and all vertices have a label.*

Proof. We first note that, as soon as we set the label of a vertex, we immediately satisfy it by either applying Satisfy Label or by branching with Arc Branch. Since satisfying all labels turns D into a funnel, we only need to show that all vertices have a label if Label Branch, Set Label and Satisfy Label are not applicable. Arc Branch only removes arcs and does not set labels, hence we do not need to consider it further in this proof.

We first show that, if there is some forbidden subgraph $D' = (V', A') \subseteq D$ and Set Label and Satisfy Label are not applicable, then Label Branch is applicable. Let $v, u \in V'$ be two (not necessarily distinct) vertices in D' such that $\text{indeg}_{D'}(v) > 1$, $\text{outdeg}_{D'}(u) > 1$ and all vertices between v and u (if any) have in- and outdegree one in D. If D is not a funnel, we can always find such a forbidden subgraph by following the successor of v and the predecessor of u.

If $\forall w \in \text{in}_D(v) : L(w) \neq \bot$, then we can apply Label Branch (if $\text{outdeg}_D(v) > 1$) and we are done, or we know $L(v) = \textsc{Merge}$ due to Set Label. Since all vertices between v and u have in- and outdegree one, we also know from the latter case that there is some arc (x, y) in the (v, u)-path such that $L(x) = \textsc{Merge}$ and $L(y) = \bot$. Note that it cannot happen that $L(y) = \textsc{Fork}$ since Satisfy Label is not applicable. We also know $\text{outdeg}_D(y) > 1$, since Reduction Rule 3.2.10 is not applicable. This implies that Label Branch is applicable on y.

We now consider the remaining case where $\exists w \in \text{in}_D(v) : L(w) = \bot$, which is illustrated in Figure 3.11. We show that we can find some vertex in $\text{in}^*(w)$ where we can apply Label Branch. Consider the longest (x, w)-path that only contains vertices in $\text{in}^*(w)$ which do not have a label. If $x = w$, then we can apply Label Branch on w since $\forall y \in \text{in}(x) : L(y) \neq \bot$. Otherwise we know $\text{outdeg}(x) > 1$, since Set Label is not applicable. Thus, we can apply the branching rule on x.

Since only these two cases are possible, and in both we can apply Label Branch, it follows, by contraposition, that D is a funnel and all vertices have a label if Label Branch, Set Label and Satisfy Label are not applicable. □

By combining the previous reduction and branching rules, we can construct a search-tree algorithm for computing the arc-deletion distance to a funnel of a DAG. Since the size of the search-tree is bounded on the solution size, we obtain an FPT algorithm with respect to the arc-deletion distance to a funnel.

Theorem 3.2.15. *Let $D = (V, A)$ be a DAG. Finding an arc-set $A' \subseteq A$ such that $D \setminus A'$ is a funnel is doable in $\mathcal{O}(3^d(|V| + |A|))$ time, where $d = |A'|$.*

Proof. We first exhaustively apply Set Label to D. For each vertex we have two auxiliary counters for the number of in- and outneighbors whose labels are known. Hence, testing if the labels of all in- or outneighbors of a vertex are known can be done in constant time, and assigning a label to a vertex v takes $\mathcal{O}(\text{indeg}(v) + \text{outdeg}(v))$ time. We iterate through all vertices once, revisiting a vertex only if one of its neighbors received a label. Since we visit a vertex v at most $\text{indeg}(v) + \text{outdeg}(v)$ times, we can exhaustively apply this reduction rule in $\mathcal{O}(|V| + |A|)$ time.

We then search for a vertex where Label Branch is applicable. We first iterate through all vertices, collecting in a set S those where Label Branch is applicable. By again using

the same two counter as before, we can check in constant time if the branching rule is applicable to a vertex or not. If it is, we add the vertex to S. When $S = \emptyset$, we can stop the algorithm. Every vertex is visited at most $\text{indeg}(v) + \text{outdeg}(v)$ times, and applying the rule to a vertex takes $\mathcal{O}(\text{ indeg}(v) + \text{outdeg}(v))$ time. In total, applying this branching rule takes $\mathcal{O}(|V| + |A|)$ time. By applying Arc Branch and Satisfy Label immediately after Label Branch, we will need to remove at least one arc in each case of Label Branch. Label Branch branches into two possible labels and, in the worst case, it also branches on removing at most d from $d+1$ arcs. If we analyze a branching strategy which considers two arcs in every step, branching into removing one or the other, we obtain a search tree with 2^d leaves. We additionally need to branch on the label of a vertex. However, we only need to apply Arc Branch in one of the branches of Label Branch. Consider a vertex v to which Label Branch is applied. If $\forall w \in \text{in}(v) : L(w) \neq \bot$ or $\exists w \in \text{in}(v) : L(w) = \textsc{Fork}$, then Satisfy Label can satisfy the label of v if we set $L(v) \coloneqq \textsc{Fork}$. The other case of the rule follows analogously, with no further branching being necessary if we set $L(v) \coloneqq \textsc{Merge}$. This gives us a total running time of $\mathcal{O}(3^d(|V| + |A|))$.

By Lemma 3.2.14 we obtain a funnel after exhaustively applying Label Branch, Satisfy Label and Set Label. Since the search tree explores every possible labeling of the considered vertices, the labeling of an optimal solution is also considered. While setting the labels, we simulate the decisions of `ArcDeletionSet` by applying Satisfy Label. Hence, it follows from Proposition 3.2.7 that a solution of size at most d is found if it exists. □

As stated before, the algorithm described here only works on DAGs. On digraphs we can no longer guarantee that, if the reduction rules are no longer applicable, then either the digraph a funnel or Label Branch is applicable. The reason for this is that digraphs do not necessarily have sinks or sources, and so we cannot always find a vertex v where the labels of all inneighbors of v are known. Without this property, we end up with an algorithm with the same worst-case running time as the one described in Theorem 3.2.6.

3.2.4 A Lower Bound

In practice, the running time of a search-tree algorithm is longer if the input is a no-instance. To reject the input, all leaves of the search tree need to be evaluated, making the algorithm reach the worst-case running time. In order to improve the running time in practice, often pruning rules are used. These rules allow an entire subtree to be evaluated without having to compute all of its leaves. For example, if we know we need to remove at least k arcs in order to turn the input DAG into a funnel, and we ask for a arc-set of size at most d, then if $d > k$, we can stop the search tree and answer "`no`".

We describe in this section an algorithm which computes a lower bound for the arc-deletion distance to a funnel of a DAG. Since funnels are characterized by forbidden subgraphs, one natural lower bound consists of collecting such subgraphs. Because we need to destroy all forbidden subgraphs in order to turn the input into a funnel, the

Algorithm 2 Compute a lower bound for the arc-deletion distance to a funnel.

```
1: function LowerBound(DAG D = (V, A))
2:     d̂ := 0
3:     for all v ∈ V do
4:         if indeg(v) > 1 ∧ outdeg(v) > 1 then
5:             d := min{indeg(v), outdeg(v)}
6:             A' := take(d, {(u, v) | u ∈ in(v)})
7:             A' := A' ⊎ take(d, {(v, u) | out(v)})
8:             D := D \ A'
9:             d̂ := d̂ + d − 1
10:        else if indeg(v) > 1 ∧ outdeg(v) = 1 then
11:            u := out(v)[0]
12:            while indeg(u) = 1 = outdeg(u) do
13:                u := out(u)[0]
14:            if outdeg(u) > 1 ∧ indeg(u) = 1 then
15:                A' := {(v, out(v)[0]), (in(u)[0], u)} ⊎ take(2, {(w, v) | w ∈ in(v)})
16:                A' := A' ⊎ take(2, {(u, w) | w ∈ out(u)})
17:                D := D \ A'
18:                d̂ := d̂ + 1
19:    return d̂
```

number of arcs we need to remove in order to destroy the collected subgraphs is a lower bound to the arc-deletion distance to a funnel of a DAG.

For the lower bound to be useful in practice, it needs to be quickly computable, preferably in polynomial time. Since we do not know any polynomial-time algorithm for computing the arc-deletion distance to a funnel of an arbitrary DAG, we need to consider forbidden subgraphs where this distance can be computed efficiently. Furthermore, if we require the collected subgraphs to be arc-disjoint, then we can consider each subgraph individually when computing its arc-deletion distance to a funnel. In the proposition below we formalize how to obtain a lower bound for the arc-deletion distance to a funnel of a DAG.

Proposition 3.2.16. *Let D be a DAG and $F_1, F_2 \dots F_n \subseteq D$ be arc-disjoint subgraphs of D which are not funnels. Let d_i be the arc-deletion distance to a funnel of F_i. Then the arc-deletion distance to a funnel of D is at least $\sum_{i=1}^{n} d_i$.*

Proof. We can only destroy a forbidden subgraph by removing some of its arcs. Arcs outside the subgraph do not help in destroying it. Since all F_i are arc-disjoint, arcs used to destroy some F_i cannot be used to destroy another F_j. Hence, an arc-deletion set that turns D into a funnel must contain at least $\sum_{i=1}^{n} d_i$ many arcs. □

Clearly, if we set $F_1 := D$, we obtain by Proposition 3.2.16 a lower bound which is equal to the real distance. This is, in itself, useless, since in order to compute the lower bound we need to be able to compute the arc-deletion distance to a funnel of D. We are

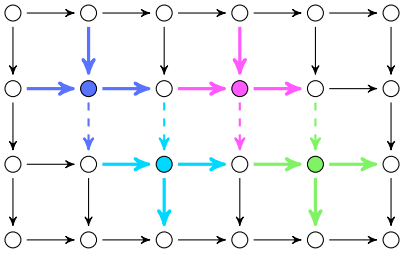


Figure 3.12: A collection of forbidden subgraphs which give a lower bound for the arc-deletion distance to a funnel of a grid. The arcs used by each subgraph are colored according to the subgraph they belong to. In order to destroy each subgraph we collected we need to remove one arc. To turn the grid into a funnel, it suffices to remove the four dashed arcs. In this case, the lower bound met the actual distance to a funnel.

actually interested in finding forbidden subgraphs in which computing the arc-deletion distance to a funnel is easy. This can be achieved by searching for vertices with both in- and outdegree greater than one. Such vertices can be easily found in linear time (Lines 4 to 9 in Algorithm 2). For a vertex v found we then take $x = \min\{\text{indeg}(v), \text{outdeg}(v)\}$ inneighbors and x outneighbors arbitrarily, as long as the respective arcs are not already taken. The vertex v together with the chosen neighbors form a forbidden subgraph. By marking each arc with a Boolean value which says whether it is in use or not, we can find a maximal set of such forbidden subgraphs in $\mathcal{O}(|V| + |A|)$ time, and compute the arc-deletion distance to a funnel of each of them also in linear time.

Another type of DAG for which we can compute its arc-deletion distance to a funnel efficiently is composed by two vertices v, u such that $\text{indeg}(v) > 1$,$\text{outdeg}(v) = 1$, $\text{outdeg}(u) > 1$, $\text{indeg}(u) = 1$ and $u \in \text{out}^*(v)$. If we additionally require that every vertex in a (v, u)-path to have both in- and outdegree equal to one, we can turn such a DAG into a funnel by simply removing the outgoing arc of v. When taking these subgraphs into our collection, we only need to take two inneighbors of v, two outneighbors of u and all vertices between the unique (v, u)-path. Finding such subgraphs can also be done efficiently by first identifying a vertex with $\text{indeg}(v) > 1$ and $\text{outdeg}(v) = 1$ (Lines 10 to 18 in Algorithm 2). We then follow the unique successor u of v, stopping if $\text{outdeg}(u) > 1$ or $\text{indeg}(u) > 1$. In the case that $\text{outdeg}(u) > 1$, we found a forbidden subgraph. We visit each vertex v at most $1 + \text{indeg}(v)$ times, obtaining a running time of $\mathcal{O}(|V| + |A|)$ for computing a maximal set of such arc-disjoint subgraphs.

We consider now the same example where the approximation algorithm from Section 3.2.2 achieved a factor of two, namely grids. Interestingly, the lower bound proposed here finds the exact arc-deletion distance to a funnel on grids. We do not know, however, how to check this in polynomial time for the general case. That is, checking if there is an arc-deletion set of the same size as the lower bound. When formulating the arc-deletion problem as a decision problem, the lower bound acts like a certificate for "`no`" instances, allowing us to reject an input if the lower bound is too large.

Example 3.2.3. Let $D = (V, A)$ be a directed grid with h rows and w columns. For

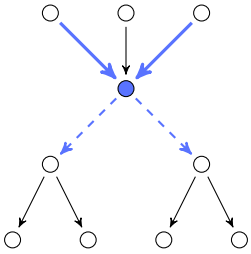


Figure 3.13: A DAG where the lower bound found by Algorithm 2 is not optimal. The colored arcs correspond to a subgraph found by the algorithm, implying a lower bound of one, yet we need to remove both dashed arcs in order to turn the DAG into a funnel.

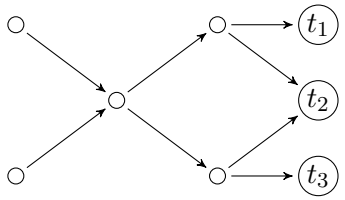
simplicity, we assume w and h are even. Each row of D contains $w-2$ vertices with both in- and outdegree greater than one. If we iterate through the vertices first by row and then by column, Algorithm 2 finds $(w-2)/2$ single cores in each row (except for the first and last ones, refer to Figure 3.12). Hence, in total the algorithm finds $(h-2)(w-2)/2$ single cores. For each of them we need to remove one arc, and so the lower bound equals $(h-2)(w-2)/2$, which is also the arc-deletion distance to a funnel of the grid.

The lower bound given here is not always optimal. In fact, we do not know if for every DAG D there is some set F of subgraphs D such that the arc-deletion distance to a funnel of D is the sum of the arc-deletion distances to funnel of each subgraph in F, with the additional restriction that we can compute the arc-deletion distance to a funnel of the DAGs in F in polynomial time. One example where this set F is not know is shown in Figure 3.13. If such an set F always exist, however, then the problem of deciding whether the arc-deletion distance to a funnel of a DAG is at most d would be in coNP. The size of F is clearly polynomial in the size of the input DAG, and if we find some set F such that the sum of the arc-deletion distances to funnel of every subgraph in F is greater than d, then we can reject the input. If the problem were indeed in coNP, then this would explain why we could not show that it is NP-complete. If it were at the same time NP-complete and in coNP, then this would imply that NP=coNP, which is assumed not to be the case.

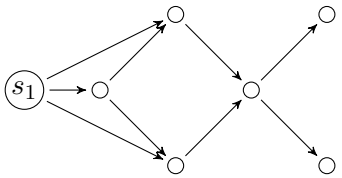
3.3 Funnel Depth and Height

While arc- and vertex-deletion parameters are natural distance measurements, they have the disadvantage of quickly growing together with the input size: If we simply duplicate a DAG, we also duplicate its arc- and vertex-deletion distance to a funnel. From an algorithmic point of view, however, simply duplicating an instance should not make it much harder. For this reason, we define here two parameters which represent a more global distance to a funnel of a DAG.

The idea is to view the DAG as having multiple layers, where the innermost one is a funnel. The number of layers is then the distance parameter. When considering a path from a source to a sink, this distance parameter tells us how many steps we need to take



(a) A DAG with funnel-height two. If we start at t_1 or t_3 then we already have a funnel, but if we start at t_2 we need to take one further step backwards in order to reach a funnel.



(b) A DAG with funnel-depth three. If we start at s_1, then after at most two steps we reach a funnel. Since in general a DAG can have more than one source, the choice of s_1 counts as a step.

Figure 3.14: Examples of funnel depth and height. In a sense, both parameters measure the number of steps, starting at some source or sink, we need to take in order to reach a funnel.

in order to reach a funnel. For an algorithm, this parameter could indicate how many steps of a path are "hard" to compute.

Here we consider two variants of the distance parameter. In the first one, the layers appear before the funnel (that is, they start at the source vertices). On the second one, they come after the funnel. We call the distance parameter in these cases funnel depth and funnel height, respectively. They are formally defined below (an example can be seen in Figure 3.14).

Definition 3.3.1 (Funnel depth)**.** Let D be a DAG with sources $\{s_i\}_{i=1}^n$. The funnel depth of D is given by $\mathrm{fd}(D)$. If D is a funnel, then $\mathrm{fd}(D) \coloneqq 0$. Otherwise,

$$\mathrm{fd}(D) \coloneqq 1 + \max\{\mathrm{fd}(\mathrm{out}^*[s_i] \setminus \{s_i\})\}_{i=1}^n.$$

Definition 3.3.2 (Funnel height)**.** Let D be a DAG with sinks $\{t_i\}_{i=1}^m$. The funnel height of D is given by $\mathrm{fh}(D)$. If D is a funnel, then $\mathrm{fh}(D) \coloneqq 0$. Otherwise,

$$\mathrm{fh}(D) \coloneqq 1 + \max\{\mathrm{fh}(\mathrm{in}^*[t_i] \setminus \{t_i\})\}_{i=1}^m.$$

In a sense, the funnel depth of a DAG captures the maximum number of "hard" decisions an algorithm has to take when constructing some path from a source to a sink. An alternative way of understanding the parameter is to consider the private arcs of a source-sink path. While in a funnel we can uniquely identify any source-sink path through one arc, in a DAG D we need at most $\mathrm{fd}(D) + 1$ arcs: After taking $\mathrm{fd}(D)$ arcs we know we reached a funnel. The same is true for the funnel height. This property implies that a DAG $D = (V, A)$ with maximum degree Δ has at most $\Delta^{\mathrm{fd}(D)} \cdot |A|$ many paths. Hence, for some problems we can obtain a simple brute-force XP algorithm with respect to funnel height or funnel depth simply by testing all paths of a DAG.

We can quickly compute both funnel height and funnel depth by following the topological ordering of the DAG. For funnel depth, all sinks can be marked with 0. We then follow the topological ordering backwards, until we find a vertex v such that $\mathrm{out}^*[v]$ is

not a funnel. This vertex is then marked with one, and for all of its predecessors we just need to take one plus the maximum of the values of the outneighbors. An analogous algorithm works for funnel height. Hence, both funnel height and funnel depth can be computed in $\mathcal{O}(|V| + |A|)$ time.

Chapter 4

Applications of Funnels

In this chapter we analyze the usefulness of funnels for the construction of algorithms. We start by considering problems which are NP-hard on DAGs and ask if they become polynomial-time solvable if we restrict the input DAG to be a funnel. If we do find a polynomial-time algorithm for a problem, then we also investigate the parameterized complexity of said problem with respect to some distance to a funnel measurement.

In Section 4.1 we consider the DAG Partitioning problem, which can be used to study the behavior of the news cycle, and provide a polynomial-time algorithm for it when the DAG is a funnel. In Section 4.2 we prove that the Max-Leaf Out-Branching problem remains NP-hard even when restricted to funnels. Finally, we study the k-Linkage problem in Section 4.3, providing an FPT algorithm with respect to the arc-deletion distance to a funnel of the input DAG.

4.1 DAG Partitioning

Leskovec, Backstrom, and Kleinberg [LBK09] developed a method of tracking how news articles spread over the web, and applied their method in practice. To this end, they constructed a digraph from millions of news articles where each vertex is some relevant statement (for example a quote) and there is an arc from a vertex v to another vertex u if with certain probability v came from u. From this digraph they extracted which topics were present in each article in such a way that it was possible to identify which articles belonged to the same topic.

They achieved this by removing arcs from the digraph such that each weakly connected component had one sink, which correspond to the statement from which all other vertices on the same connected component derived from. They called this problem DAG Partitioning. Since they did not formally define the problem as a decision problem, we use the definition due to van Bevern et al. [Bev+17].

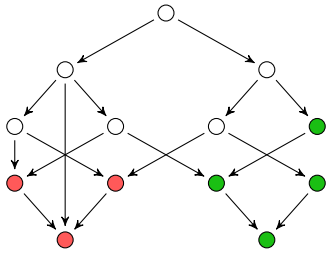


Figure 4.1: Motivating example for DAG Partitioning on funnels. Colored vertices correspond to articles whose origin is known. These vertices always have outdegree at most one. For articles whose origin is unknown, we make a decision tree, with the outneighbors corresponding to possible originators. Such a construction gives us a funnel.

DAG Partitioning [Bev+17]

Input: A directed acyclic graph $D = (V, A)$ with positive integer arc costs $\omega : A \to \mathbb{N}$ and a positive integer $k \in \mathbb{N}$.

Question: Is there a set $S \subseteq A$ with $\sum_{a \in S} \omega(a) \leq k$ such that each weakly-connected component in $D' = (V, A \setminus S)$ has exactly one sink?

The problem was considered from a parameterized perspective by van Bevern et al. [Bev+17]. They developed a $\mathcal{O}(2^k \cdot (|V| + |A|))$ algorithm together with linear-time reduction rules. Alamdari and Mehrabian [AM12] provided an NP-hardness reduction from 3-SAT to DAG Partitioning. From this reduction we can observe some NP-hardness results also for some distance to a funnel parameters described in Chapter 3. Their reduction produces a DAG where the vertex-deletion distance to a funnel is one and which has funnel height and depth equal to one. The arc-deletion distance to a funnel is, however, unbounded.

In this section we propose a polynomial-time algorithm for DAG Partitioning restricted to funnels. It might be realistic to assume that the input DAG is a funnel if it is constructed in the following manner, illustrated in Figure 4.1. We use a database of statements whose origin is known, either due to manual verification or from some previous execution of the algorithm. Because each vertex should only reach one sink, the database corresponds to an in-forest, which is exactly the Merge region of a funnel. For each statement whose origin we want to find we construct a decision tree. This decision tree connects one statement with others through some mutually exclusive criterion. For example, we could consider who said the statement, in which context it was spoken, if it happened before or after an event, and so on. At some point the possible originators of a vertex will be narrowed down to some of the statements on the database.

Perhaps a more concrete example of when the DAG might be a funnel is if we want to determine the origin of some archaeological finding. We might know the origin of certain objects, which will form the database. For a newly found object we might ask where and when it was constructed, what kind of technology was used in its manufacture, which religious marking it has and so on. After constructing the decision tree the object is linked to other objects on the database which share similar properties.

Before describing the algorithm we first consider two data reduction rules proposed by van Bevern et al. [Bev+17].

Reduction Rule 4.1.1 ([Bev+17])**.** If there is an arc (v, w) such that w can reach exactly one sink $t \neq w$ and v can reach multiple sinks, then delete the arc (v, w) and if there is no arc (v, t), then add it with cost $\omega(v, w)$, otherwise, set $\omega(v, t) \coloneqq \omega(v, t) + \omega(w, t)$.

Reduction Rule 4.1.2 ([Bev+17])**.** If, for some sink t, the set L of non-sink vertices that can reach only t is nonempty, then delete all vertices in L.

If the input digraph is a funnel, after applying both reduction rules it acquires a very simple structure similar to that of a directed tree. We formalize this with the following proposition.

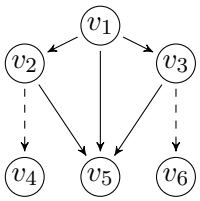
Proposition 4.1.3. *Let $D = (V, A)$ be a funnel where Reduction Rules 4.1.1 and 4.1.2 are no longer applicable. Then all vertices with indegree greater than one are sinks.*

Proof. We argue that, if there is some vertex with indegree greater than one which is not a sink, then we can apply one of the reduction rules. Assume towards a contradiction that there is some vertex $v \in V$ with $\operatorname{indeg}(v) > 1$ and $\operatorname{outdeg}(v) > 0$. Since D is a funnel, we also know that $\operatorname{outdeg}(v) = 1$ and every $w \in \operatorname{out}^*(v)$ has outdegree at most one. Hence, v can reach only one sink t. Let $w \in \operatorname{in}^*(v)$ be an arbitrary vertex which can reach at least two sinks and has the shortest distance to v. If no such w exists, then every vertex in $\operatorname{in}^*[v] \cup \operatorname{out}^*[v]$ can reach exactly one sink, and Reduction Rule 4.1.2 is applicable. If w exists, then we know there is some $w' \in \operatorname{out}(w)$ such that w' can reach only t. Thus, Reduction Rule 4.1.1 is applicable. In either case, one of the reduction rules is applicable, which is a contradiction to the fact that Reduction Rules 4.1.1 and 4.1.2 are not applicable. Hence, all vertices in D with indegree greater than one must be sinks. □

Corollary 4.1.4. *Let $D = (V, A)$ be a funnel where Reduction Rules 4.1.1 and 4.1.2 are no longer applicable. Then all incoming arcs of all sinks t_j are private.*

Proof. Consider two paths P and Q that start at some source where both contain the arc (v, t_j), with t_j being a sink. From Proposition 4.1.3 we know that all vertices with indegree greater than one are sinks. Thus, all vertices on either path except for the sink t_j have indegree zero or one. This means that there is only one path from a source to v, and so both paths P and Q are the same. Thus the arc (v, t_j) is private. □

With Corollary 4.1.4, we can construct a polynomial-time algorithm for DAG Partitioning by restricting the input digraph to a funnel. The idea is to start at the sinks and using dynamic programming compute a minimum-cost solution. Due to Corollary 4.1.4 we know that paths only meet at the sinks. This implies that the set of arcs belonging to the subgraph which starts at some vertex v is disjoint to the set of arcs which belong another vertex u if there is not path between both vertices. In other words, we know



T	v_4	v_5	v_6
v_1	4	2	4
v_2	1	1	∞
v_3	∞	1	1
v_4	0	∞	∞
v_5	∞	0	∞
v_6	∞	∞	0

Figure 4.2: An example of the polynomial-time algorithm for DAG Partitioning on a funnel. The dashed arcs represent an optimal solution of size two. All arcs here have cost 1. One can see that the cheapest solution costs $T(v_1, v_5) = 2$. This values comes from the sum $T(v_2, v_5) + T(v_3, v_5)$, that is, the cost of making both outneighbors of v_1 also reach v_5.

that an optimal solution for $\text{out}^*[v]$ can be combined with an optimal solution for $\text{out}^*[u]$ in order to produce an optimal solution for the union of both subgraphs. The algorithm is formally described below.

Let $(D = (V, A), \omega, k)$ be a DAG Partitioning instance, where D is a funnel with sources $\{s_i\}_{i=1}^n$ and sinks $\{t_i\}_{i=1}^m$ and the data reduction rules above are no longer applicable. We define a dynamic programming table $T : V \times V \to \mathbb{N} \cup \{\infty\}$ where $T(v, t_i)$ is the cost of making t_i the only sink reachable by v, or ∞ if that is not possible. Naturally, $T(t_i, t_j)$ equals zero if $i = j$ and ∞ otherwise. We first define an auxiliary function κ where $\kappa(v, t)$ represents the costs of making every outgoing neighbor u of v compatible with v when v can only reach t. That is, either u also can only reach t, or the arc (v, u) is removed.

In order to determine $T(v, t)$, we observe that for each $u \in \text{out}(v)$ there are two possibilities: Either u can also only reach t, or we have to remove the arc (v, u). To compute T efficiently, we first determine the minimum cost of making every outneighbor of v compatible with it. This, however, can lead to a solution where v is disconnected from all of its outneighbors, thus not reaching t. We fix this by choosing some $u \in \text{out}(v)$ with the smallest additional cost of forcing u to reach t while keeping the arc (v, u). This additional cost is given by the function $\delta(v, u, t)$. Clearly $\delta(v, u, t) = 0$ if u already reaches t in the cheapest solution found. We also define a table $M(v)$ which simply stores the

cheapest solution for v. We claim that those functions satisfy the following:

$$T(v,t) = \begin{cases} 0, & t = v \in \{t_i\}_{i=1}^m \\ \infty, & t \neq v \in \{t_i\}_{i=1}^m \\ \kappa(v,t) + \min\limits_{u \in \text{out}(v)} \delta(v,u,t), & \text{otherwise} \end{cases}$$

$$M(v) = \min_{1 \leq i \leq m} T(v,t_i)$$

$$\kappa(v,t) = \sum_{u \in \text{out}(v)} \min\{T(u,t), M(u) + \omega(v,u)\}$$

$$\delta(v,u,t) = \begin{cases} 0, & T(u,t) \leq M(u) + \omega(v,u) \\ T(u,t) - (M(u) + \omega(v,u)), & \text{otherwise.} \end{cases}$$

We then compute $T(a,t_i)$ for every $a \in V$ and every sink t_i by starting at the sinks and then going inwards towards the source. An example is given in Figure 4.2.

Lemma 4.1.5. *A* DAG Partitioning *instance* $(D = (V,A), \omega, k)$ *where* D *is a funnel with sources* $\{s_i\}_{i=1}^n$ *and sinks* $\{t_j\}_{j=1}^m$ *is a yes-instance if and only if*

$$\sum_{i=0}^{n} \min_{1 \leq j \leq m} T(s_i, t_j) \leq k.$$

Proof. We first show for every $v \in V$ and every sink t_j that if $T(v,t_j) \neq \infty$ then there is an arc-set S_j of cost at most $T(v,t_j)$ such that v reaches only t_j in $D \setminus S_j$. We do this by induction on the height of v.

Let $v \in V$ be a vertex of height zero, that is, a sink and t_j an arbitrary sink. Since v is a sink, it obviously can only reach itself, since it has no outneighbors. If $t_j = v$ we know that $T(v,v) = 0$. Thus, by choosing $S_j \coloneqq \emptyset$ we satisfy the statement. Otherwise $T(v,t_j) = \infty$ and the statement trivially holds.

Now let $v \in V$ be a vertex of height $h > 0$ and t_j a sink. Obviously the height of all outneighbors of v is smaller than the height of v itself, and so we can apply our induction hypothesis on them. We know that for every $u \in \text{out}(v)$ with $T(u,t_j) \leq M(u)$ there is some arc-set X_u with cost at most $T(u,t_j)$ such that u only reaches t_j in $D \setminus X_u$. For $u \in \text{out}(v)$ with $T(u,t_j) > M(u)$, we know there is some arc-set Y_u with cost at most $T(u,t_k) + \omega(v,u)$, with $k \neq j$, which contains the arc (v,u) and is such that u only reaches t_k in $D \setminus Y_u$. We also know from Corollary 4.1.4 that all arcs coming into a sink are private. This implies that every X_w and Y_u for $u, w \in \text{out}(v)$ are pairwise disjoint, since any arc that appears in a path from u to some sink t_k is not in any path from w to t_ℓ, for any $w \in \text{out}(v)$ different from u and any sink t_ℓ. Thus, the cost of the arc-set S_v obtained by the union of all X_u and Y_u equals the sum of the individual costs. Every X_u costs at most $T(u,t_j)$ and every Y_u costs at most $M(u) + \omega(v,u)$. In other words, the sum is at most $\kappa(v,t_j)$. However, if there are no X_u then all arcs between v and its outneighbors will be removed, meaning that v becomes a sink in $D \setminus S_v$ and, therefore, does not reach t_j as desired. In this case we replace some Y_u with an

arc-set A'_u that makes u reach t_j in $D \setminus A'_u$. We know that this arc-set exists, and its additional cost is at most $\delta(v, u, t) = T(u, t_j) - (M(u) + \omega(v, u))$. If we choose an A'_u with minimum additional cost, then removing the arc-set $(S_v \setminus Y_u) \cup A'_u$ costs at most $T(v, t_j) = \kappa(v, t_j) + \min_{u \in \text{out}(v)} \delta(v, u, t_j)$. By accumulating all solutions for all sources, this proves that, if $\sum_{i=0}^{n} \min_{1 \leq j \leq m} T(s_i, t_j) = k \neq \infty$, then there is a solution of cost at most k.

It remains to show that, if there is a vertex v and an arc-set S_j with cost k such that every vertex in $\text{out}^*[v]$ reaches only one sink in $\text{out}^*[v] \setminus S_j$ and v reaches the sink t_j, then $T(v, t_j) \leq k$. Again we prove this by induction on the height of v.

The statement clearly holds for vertices of height zero, since they are themselves sinks and no arcs need to be removed. As the cost of an arc-set is never negative, it is trivially greater than or equal to $T(v, v) = 0$.

Now let $v \in V$ be a vertex with height greater than 0 and S_j be some arc-set such that every vertex in $\text{out}^*[v]$ reaches only one sink in $\text{out}^*[v] \setminus S_j$ and v reaches the sink t_j. Let k be the cost of S_j. Consider an outneighbor $u \in \text{out}(v)$ of v and the arc-set $R_u = \mathcal{A}(\text{out}^*[u]) \cap S_j$. By the definition of S_j, the vertex u only reaches one sink t_u in R_u. Furthermore, by the induction hypothesis the cost of R_u is at least $T(u, t_u)$. Let X be the set of outneighbors of v such that the arc from v to any vertex in X is not present in S_j, and $w \in X$ be a vertex such that $R_w \leq R_x$ for all $x \in X$. Note that $t_u = t_j$ for any $u \in X$ and that X cannot be empty. Let Y be the set of the remaining outneighbors of v, that is, those whose incoming arc from v is in S_j.

The cost of S_j is then at least

$$
\begin{aligned}
&\sum_{u \in X} T(u, t_j) + \sum_{u \in Y} [T(u, t_u) + \omega(v, u)] \\
\geq &T(w, t_j) + \sum_{w \neq u \in \text{out}(v)} \min\{T(u, t_j), M(u) + \omega(v, u)\} \\
\geq &\delta(v, w, t_j) + \sum_{u \in \text{out}(v)} \min\{T(u, t_j), M(u) + \omega(v, u)\} \\
\geq &\kappa(v, t_j) + \min_{u \in \text{out}(v)} \delta(v, u, t_j) \\
= &T(v, t_j).
\end{aligned}
$$

Hence, the statement holds. □

With the previous lemma we can prove the following theorem, inferring that DAG Partitioning is polynomial-time solvable if the input is a funnel.

Theorem 4.1.6. DAG Partitioning *can be solved in* $\mathcal{O}(|V|^3)$ *time if the input DAG* $D = (V, A)$ *is a funnel.*

Proof. We first compute a topological ordering of D in linear time. By following this ordering in reverse (that is, starting from the sinks), we compute $T(v, t)$ for each vertex $v \in V$ and each sink t. With respect to complexity, the worst case occurs when we

need to calculate $\kappa(v,t) + \min_{u \in \text{out}(v)} \delta(v,u,t)$. We claim that this can be done in linear time.

First, note that $M(v)$ can be stored for a certain vertex v as soon as $T(v,t_i)$ has been computed for every sink t_i. This adds $\mathcal{O}(|V|)$ when computing a row of T, which implies plus $\mathcal{O}(|V|^2)$ to the overall complexity of the algorithm. Hence, after finishing the row of v we can access $M(v)$ in constant time.

Computing $\kappa(v,t)$ requires us to iterate through all outneighbors of v and compute the minimum of a set of size two. All values in this set are precomputed, and so determining the value of $\kappa(v,t)$ can be done in linear time.

Computing $\delta(v,u,t)$ is done in constant time since all necessary values are already precomputed. Thus, computing each entry in $T(v,t)$ takes $\mathcal{O}(|V|)$ time. Since there are a total of $\mathcal{O}(|V|^2)$ entries, computing the entire table takes $\mathcal{O}(|V|^3)$ steps.

By Lemma 4.1.5, we only need to compute $\sum_{i=0}^{n} \min_{1 \le j \le m} T(s_i,t_j) \le k$ in order to determine whether (D,k,ω) is an yes-instance or not. The value $\min_{1 \le j \le m} T(s_i,t_j)$ is already stored in $M(s_i)$, and so this steps takes linear time. In total, we need $\mathcal{O}(|V|^3)$ steps. □

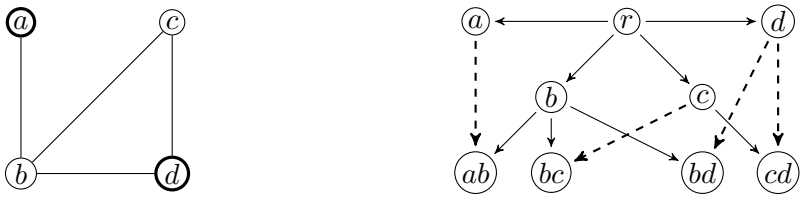
4.2 Maximum Leaf Out-Branching

In this section we consider the problem of finding a spanning tree of a digraph which contains the maximum number of leaves. Since the definition of a tree for digraphs can be ambiguous (for example, if we only require the absence of cycles like in undirected graphs, we actually obtain the definition of directed acyclic graphs), we first define what kind of tree are we interested in.

We say that an *out-tree* is a directed acyclic graph where one vertex has indegree zero (called root) and all others have indegree one. Vertices with outdegree zero are called leaves. An *out-branching* is then an out-tree which contains all vertices, that is, a spanning out-tree. Here we are interested in finding out-branchings with the maximum number of leaves. This is useful for sensor networks [BD07], where non-leaf vertices represent routers which forward information from other sensors to a central (root) vertex. Due to cost constraints, one wishes to minimize the number of routers, which is the same as maximizing the number of leaves. The problem is formally defined below.

Max-Leaf Out-Branching [BD11]
Input: A directed graph $D = (V,A)$ and an integer k.
Question: Does D contain an out-branching with at least k leaves, that is, a spanning subtree where one vertex has indegree zero, all others have indegree one and at least k vertices have outdegree zero?

We show that the problem is NP-hard on funnels by providing a reduction from Independent Set, defined below. An illustration of the reduction is provided in Figure 4.3.



(a) Independent Set instance. The bold vertices form a solution of size two.

(b) Reduced Max-Leaf Out-Branching instance. Removing the dashed arcs provides an out-branching with six leaves.

Figure 4.3: Example of the reduction from Independent Set to Max-Leaf Out-Branching.

Independent Set

Input: An undirected graph $G = (V, E)$ and a number k.

Question: Is there a vertex-set $V' \subseteq V$ of size at least k such that there are no edges in G between vertices of V'?

Theorem 4.2.1. Max-Leaf Out-Branching *remains NP-hard if the input digraph is a funnel.*

Proof. We provide a reduction from the NP-complete problem Independent Set. Let $G = (V, E)$ be a graph. We construct a funnel D as follows. For every vertex in G we add a vertex in D. For every edge $\{v, u\}$ in G we add a vertex in D together with the arcs $(v, \{v, u\})$ and $(u, \{v, u\})$. We add the arcs (r, v) to D for every vertex $v \in V$.

We now show that D has an independent set of size k if and only if it has an out-branching with $k + |E|$ leaves.

Let I be an independent set in G of size k. We construct an out-branching for D by first removing all arcs of every vertex $v \in I$. We know that for every edge $\{v, u\} \in E$ at most one of its vertices is in I. This means that we do not create new sources by removing the arcs of v. Since the indegree of the vertices of D which correspond to an edge in G is two, we also know that removing the arcs of v will decrease the indegree of its outneighbors to one. Thus we now have $k + |E|$ leaves in D. We then remove all but one of the arcs incoming into any vertex with indegree greater than one. This yields an out-branching for D with at least $k + |E|$ leaves.

Now assume there is an out-branching B of D with $k + |E|$ leaves. All vertices of D which correspond to edges in G are already leaves in D and must remain so in B. We then know there are k vertices of D which correspond to vertices of G which are leaves in B. Let I be the set of such vertices. Assume towards a contradiction that there is some edge $\{v, u\}$ in G such that $v, u \in I$. Since both v and u are leaves in B, it follows that the arcs $(v, \{v, u\}), (w, \{v, u\})$ are not present in B. Thus the vertex in B corresponding to the edge $\{v, u\}$ has indegree zero. Hence B is not an out-branching of D, which is a contradiction. Therefore, the vertices of I form an independent set of size k in G. □

4.3 k-Linkage

We consider in this section the problem of finding vertex-disjoint paths connecting certain pairs of vertices. This problem is also known as the k-linkage problem and we will use this name in order to avoid confusions to the similar k-connectivity problem, where the task is to find k vertex-disjoint paths connecting two sets of vertices (as opposed to specific pairs). The latter can be solvable in polynomial-time through a flow network, while the former is known to be NP-hard for undirected and directed graphs. The problem in question is defined below.

k-Linkage [BJG08, p. 373]
Input: A directed graph $D = (V, A)$, two disjoint sequences of terminal vertices $(s_1, s_2 \dots s_k) \in V^k$ and $(t_1, t_2 \dots t_k) \in V^k$.
Question: Are there k vertex-disjoint paths $P_1, P_2, \dots P_k$ such that P_i is an (s_i, t_i)-path?

The problem is NP-hard on general digraphs even if $k = 2$ [FHW80]. If we restrict the input to DAGs, the problem can be solved in $\mathcal{O}(k!n^k)$ time [BJG08] and is W[1]-hard with respect to k [Sli10]. Downey and Fellows [DF12] posed as an open question whether k-Linkage is FPT with respect to k on planar DAGs, and this question was answered positively by Cygan et al. [Cyg+13], who provided an $2^{2^{\mathcal{O}(k^2)}} \cdot n^{\mathcal{O}(1)}$ algorithm for the problem.

We first claim that, if two paths from different sources to different sinks share a vertex, then some forbidden subgraph is present and the input is not a funnel. Thus, by contraposition, we can prove the following statement.

Theorem 4.3.1. *Let $(D, (s_i)_{i=1}^k, (t_i)_{i=1}^k)$ be a k-Linkage instance. If D is a funnel and for every $1 \leq i \leq k$ there is an (s_i, t_i)-path in D, then $(D, (s_i)_{i=1}^k, (t_i)_{i=1}^k)$ is a yes-instance.*

Proof. We prove the statement by showing that two paths from different sources to different sinks which share a vertex produce some forbidden subgraph for funnels. Assume towards a contradiction that there is some (s_i, t_i)-path P and some (s_j, t_j)-path Q which share at least one vertex, for $i \neq j$. Let v be the earliest such vertex, and let u be the last vertex with $u \in P$ and $u \in Q$. Since s_i and s_j are sources, there is no (s_i, s_j)-path in D. Analogously, there is no path between t_i and t_j. Thus $\text{indeg}(v) \geq 2$ and $\text{outdeg}(u) \geq 2$. Since $u \in \text{out}^*(v)$, it follows that D is not a funnel, contradicting the definition of D. Hence, any (s_i, t_j)-path is vertex-disjoint with any (s_j, t_j)-path. □

We now consider an algorithm for the more general case where the input is only restricted to be a DAG. We construct the algorithm by exploiting the similarity of the input DAG to a funnel. From Theorem 4.3.1 above we can see that if a (s_1, t_1)-path shares a vertex with a (s_2, t_2)-path, then both of these paths together form a forbidden minor for funnels. Now assume we know an arc-set A' such that $D \setminus A'$ is a funnel. Clearly some arc on the (s_1, t_1)-path or on the (s_2, t_2)-path has to be in A', otherwise some forbidden minor would be present. We claim that if those paths do not share any endpoint of the arcs in A', then they are vertex disjoint (this is shown in Lemma 4.3.2

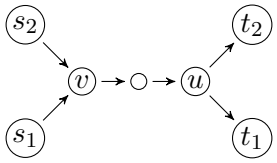


Figure 4.4: An (s_1, t_1)-path and another (s_2, t_2)-path form a forbidden subgraph for funnels when they share a vertex. If none of the arcs between v and u are removed, then it is necessary to remove some of the incoming arcs of v or some of the outgoing arcs of u in order to destroy the forbidden subgraph.

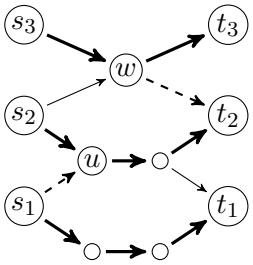
below). That is, we only need to consider which vertices of the endpoints in A' are used by each path. It is this property that we use in order to construct an algorithm for k-Linkage and we formally prove it now. Refer to Figure 4.4 when reading the proof of the following lemma.

Lemma 4.3.2. *Let $D = (V, A)$ be a DAG, $A' \subseteq A$ an arc-set such that $D \setminus A'$ is a funnel and let V' be the set of endpoints of the arcs in A'. Let P and Q be two paths from distinct sources to distinct sinks. If $P \cap Q \cap V' = \emptyset$, then $P \cap Q = \emptyset$.*

Proof. We prove the statement by contradiction. The idea is to argue that, if the paths meet at some vertex outside of V', then $D \setminus A'$ is not a funnel. Assume that $P \cap Q \cap V' = \emptyset$ but $P \cap Q \neq \emptyset$. Let $R = v \dots u$ be a maximal path with $R \subseteq P$ and $R \subseteq Q$. That is, all vertices of R are present in both paths, while none of the direct predecessors of v and none of direct successors of u are present in both paths. Obviously $\text{indeg}_D(v) > 1$ and $\text{outdeg}_D(u) > 1$. Since $P \cap Q \cap V' = \emptyset$ we know that none of vertices of R are in V'. This implies that none of the incoming or outgoing arcs of both v and u are in A'. Thus, $\text{indeg}_{D'}(v) > 1$ and $\text{outdeg}_{D'}(u) > 1$, where $D' = D \setminus A'$. Because none of the vertices in R lie in V', we also have $u \in \text{out}^*_{D'}(v)$. Hence, D' is not a funnel, which is a contradiction to the definition of A'. □

Using the result above, we develop an FPT algorithm with respect to $|A'|$ for k-Linkage. The idea is to construct a dynamic programming table where in column i we store the possibilities of finding valid paths for the first i source-target pairs. Each row then represents a subset of V' which is used by some path. By Lemma 4.3.2 we know we only need to consider vertices in V', and so the number of rows is upper bounded by a function of the size of A'. Since the paths need to be vertex disjoint, when computing the solution for the $(i+1)$-th pair it is irrelevant which path is already using a specific vertex. We only need to know which vertices are being used by some of the previous paths, and find a set of not yet used vertices which can appear in a path connecting the new pair.

Let $(D = (V, A), (s_i)_{i=1}^k, (t_i)_{i=1}^k)$ be a k-Linkage instance where D is a DAG. Let $A' \subseteq A$ be an arc-set such that $D' = D \setminus A'$ is a funnel and let V' be the set of the endpoints of the arcs in A'. We first define an auxiliary function $\text{Path} : \mathcal{P}(V) \times V \times V$ such that $\text{Path}(U, s, t) = \texttt{true}$ if and only if there is some (s, t)-path P such that $P \cap V' \subseteq U$.



	1	2	3
$\emptyset$	1	0	0
$\{u\}$	1	1	0
$\{w\}$	1	1	0
$\{u, w\}$	1	1	1

Figure 4.5: Example instance for k-Linkage Dashed arcs represent an arc-set whose removal produces a funnel. Thus V' here is $\{s_1, u, w, t_2\}$. Since the vertices s_1 and t_2 are exclusive to their respective pairs, they can be removed from V'. The solution where $T(\{u, w\}, 3) = \texttt{true}$ corresponds to the bold arcs.

We then define a table $T : \mathcal{P}(V') \times \mathbb{N} \to \mathbb{B}$ such that $T(U, i)$ is `true` if and only if there are i vertex-disjoint paths for the first i pairs in P such that none of these paths contains a vertex of $V' \setminus U$.

We compute the values on the table by distinguishing two cases. For $i = 1$ we only need to consider (s_1, t_1)-paths, and we can compute T as

$$T(U, 1) \coloneqq \text{Path}(U, s_1, t_1).$$

For $i > 1$, we need to consider all subsets $U' \subseteq U$ and ask if there is some (s_i, t_i)-path which uses no vertices in‘$V' \setminus U'$ and if there is a solution for the previous pairs which uses no vertex in $V' \setminus (U \setminus U')$. This can be done recursively as follows

$$T(U, i) \coloneqq \begin{cases} \texttt{true}, & \exists U' \subseteq U : T(U \setminus U', i-1) \wedge \text{Path}(U', s_i, t_i) \\ \texttt{false}, & \text{otherwise.} \end{cases}$$

We claim that $(D, (s_i)_{i=1}^k, (t_i)_{i=1}^k)$ is a yes-instance if and only if there is some $U \subseteq V'$ such that $T(U, |P|)$ is true. Since there are $2^{|V'|} \leq 2^{2 \cdot |A'|}$ subsets of V' and k pairs, this gives us a table of size $2^{|V'|} \cdot k$. Finding an (s_i, t_i)-path which does not visit any vertex in $V' \setminus U$ can be done with a standard linear time algorithm for path-finding in DAGs. Computing each entry on the table potentially requires going through every one of the $2^{|U|} \leq 2^{|V'|}$ subsets of U. This gives us an FPT algorithm with respect to the parameter arc-deletion distance to a funnel. We then use the previous lemma to prove the correctness of the algorithm.

Lemma 4.3.3. *Let $(D = (V, A), (s_i)_{i=1}^k, (t_i)_{i=1}^k)$ be a k-Linkage instance, $A' \subseteq A$ an arc-set such that $D \setminus A'$ is a funnel and let V' be the set of endpoints of the arcs in A'. Then $(D, (s_i)_{i=1}^k, (t_i)_{i=1}^k)$ is a yes-instance if and only if there is some $U \subseteq V'$ such that $T(U, k) =$ `true`.*

Proof. We first show by induction that if $\exists U \subseteq V' : T(U, i) = \texttt{true}$ then there are i vertex-disjoint paths which connecting $(s_j)_{j=1}^i$ to $(t_j)_{j=1}^k$ go through no vertex in $V' \setminus U$. By definition of T, the statement is true for $i = 1$.

We now show it is also true for $i \leq |P|$ assuming that it holds for $i-1$. If $T(U,i) = \texttt{true}$ for some $U \subseteq V'$, then, by definition of T, there is some $U' \subseteq U$ such that $T(U \setminus U') = \texttt{true}$ and $\text{Path}(U', s_i, t_i)$ is true. From the induction hypothesis we know there is a set X of $i-1$ vertex-disjoint paths for $(s_j)_{j=1}^{i-1}$ and $(t_j)_{j=1}^{i-1}$, and these paths do not use any vertex in $V' \setminus (U \setminus U')$. In particular, they do not use any vertex in U'. Let R_j denote an (s_j, t_j)-path in X and R_i be the (s_i, t_i)-path such that $R_i \cap V' \subseteq U'$. We then know that $R_i \cap R_j \cap V' = \emptyset$, for any $j \in \{1, \ldots, i-1\}$. From Lemma 4.3.2 we also know $R_i \cap R_j = \emptyset$. Hence there are i vertex-disjoint paths for $(s_j)_{j=1}^{i}$ and $(t_j)_{j=1}^{i}$, and those paths do not use any vertex in $V' \setminus U$. We can then conclude that if there is some $U \subseteq V'$ such that $T(U,k)$, then we can find a (s_i, t_i)-path for every $1 \leq i \leq k$ such that all such paths are vertex-disjoint. Thus $(D, (s_i)_{i=1}^{k}, (t_i)_{i=1}^{i})$ is a yes-instance.

Now assume $(D, (s_i)_{i=1}^{k}, (t_i)_{i=1}^{k})$ is a yes-instance. Consider some solution and let R_i be an (s_i, t_i)-path for each i. We show, again by induction, that $T(U,k) = \texttt{true}$ for $U = V' \cap \bigcup_{1 \leq i \leq k} R_i$. By definition of T, we know that $T(V' \cap R_1, 1) = \texttt{true}$. Let $i \leq k$ and assume the hypothesis holds for $i-1$. Let $U = V' \cap \bigcup_{1 \leq j \leq i} R_j$ and $U' = R_i \cap V'$. Since all paths R_j are vertex disjoint, it follows that $U \setminus U' = V' \cap \bigcup_{1 \leq j \leq i-1} R_j$. From the induction hypothesis we then know that $T(U \setminus U', i-1) = \texttt{true}$. Thus, $T(U,i) = \texttt{true}$ by definition of T. For $i = k$ and $U = V' \cap \bigcup_{1 \leq j \leq k} R_j$ we have that $T(U,k) = \texttt{true}$, and so the statement holds. □

We can now conclude the algorithm by combining Lemma 4.3.3 with an analysis of its running time. This is summarized in the theorem below.

Theorem 4.3.4. *k-Linkage can be solved in $\mathcal{O}(16^d \cdot |V| \cdot |A|)$ time, where d is the arc deletion distance to a funnel of $D = (V, A)$.*

Proof. The first step is to find an arc-deletion set A' such that $D \setminus A'$ is a funnel. As shown in Theorem 3.2.15, this can be done $\mathcal{O}(3^d \cdot (|V| + |A|))$ time.

Let V' be the set of the endpoints of the arcs in A'. Clearly $|V'| \leq 2d$. We then fill the table T described previously from the first to the last column. Computing a value $\text{Path}(U, s, t)$ can be done in linear time using standard path-finding algorithms for DAGs. Hence, computing all of the $2^{|V'|} \leq 4^d$ rows of the first column takes $\mathcal{O}(4^d \cdot |A|)$ time (without loss of generality $|V| \leq |A| + 1$ since weakly connected components can be considered individually).

For the subsequent columns, we need to find for each entry $T(U,i)$ some subset $U' \subseteq U$ such that $T(U \setminus U', i-1) \wedge \text{Path}(U', s_i, t_i)$. There is a total of $2^{|U|}$ subsets to test. Thus, each entry takes $\mathcal{O}(4^d \cdot |A|)$ time to be computed. Since the table has $\mathcal{O}(4^d \cdot |V|)$ entries, computing the entire table takes $\mathcal{O}(16^d \cdot |V| \cdot |A|)$ time.

From Lemma 4.3.3 we can decide the problem by searching for some $U \subseteq V'$ such that $T(U,k)$ is true, where k is the number of requested paths. This is doable in $\mathcal{O}(2^{|V'|})$ time. The bottleneck of the algorithm is computing the table itself, which is done in $\mathcal{O}(16^d \cdot |V| \cdot |A|)$ time, as shown before. □

From Figure 4.5 we can see that it is possible to simplify some instances. For example, if a non-terminal vertex v has outdegree equal to one, then any path which

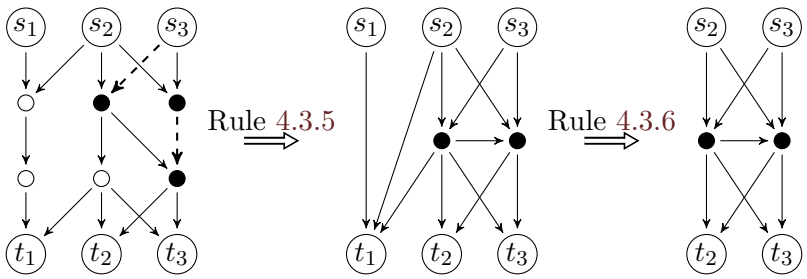


Figure 4.6: Example of the application of both data reduction rules. The dashed arcs correspond to a arc-set whose deletion turns the input DAG into a funnel. The black vertices are the endpoints of such arcs. We first contract all non-terminal vertices which have indegree or outdegree equal to one. After this step, we find a trivial path for the terminals s_1 and t_1 consisting only of those vertices. In the end, we obtain a DAG with $3d = 6$ vertices, where d is the arc-deletion distance to a funnel of the original DAG.

contains v must also contain its successor. Hence, we can skip such vertex by connecting its inneighbors with its outneighbor. This is yields the following reduction rule.

Reduction Rule 4.3.5. Let $v \in V$ be some non-terminal vertex. If $\text{indeg}(v) = 1$ and $(u, v) \in A$, then contract the arc (u, v). If $\text{outdeg}(v) = 1$ and $(v, u) \in A$, then contract the arc (v, u). If $\text{indeg}(v) = 0 \vee \text{outdeg}(v) = 0$, remove v.

Correctness of Reduction Rule 4.3.5. We first show that the reduction rule does not destroy any solution. Note that removing sources and sinks which are not terminals cannot change the solution since such vertices can never be used in any path connecting another source to another sink. Let D be the original DAG and D' the one obtained after applying Reduction Rule 4.3.5. Assume there is some solution for D. Let P be a path in this solution which goes through v. If no such path exists, the P is clearly a valid path in D', and so there is also a solution for D'. We first consider the case where $\text{indeg}(v) = 1$. The other case follows analogously. Since u is the only predecessor of v, it is clear the u must also be present in P. Let w be the successor of v which is also in P. Since reduction adds the arc (u, w) to D' after removing v, we obtain a path in D' by removing v from the sequence of P. Hence, there is also a solution for the k-Linkage instance in D'.

We now show that the reduction rule does not create new solutions. Again, we only consider the case where $\text{indeg}(v) = 1$ since the other one is analogous. Let P be some path containing u in some solution for D'. If no such path exists, then P is clearly a valid path in P since it also does not contain v. Otherwise, let w be the successor of u in P. If $w \in \text{out}_D(v)$, then adding v between u and w in P produces a valid path in D. Since P is the only path which contains u, there is other path in the solution which contains v. If $w \notin \text{out}_D(v)$, then the arc (u, w) is present in D, and so P is also a valid path for D. In both cases, there is also a solution for D. □

Another simple reduction rule which we can apply is to remove terminals which are

already connected by an arc. That is if there is an arc connecting some terminal s_i to its corresponding t_i, then we can remove both terminals from the input sequence since there is a trivial path connecting both of them. Even though this reduction rule is very simple, we need it in order to obtain an upper bound on the number of vertices which remain after exhaustively applying Reduction Rules 4.3.5 and 4.3.6. We formalize this reduction rule below.

Reduction Rule 4.3.6. Let s_i be a terminal source and t_i its corresponding sink. If an arc (s_i, t_i) exists, remove both terminals.

None of the reduction rules defined above use the arc- or vertex-deletion distance to a funnel directly. We can show, however, that if Reduction Rules 4.3.5 and 4.3.6 are not applicable, then the number of vertices on the DAG is linear on its arc-deletion distance to a funnel. The basic idea is to argue that all vertices which are neither sinks nor sources nor endpoint of any arc in an arc-deletion set are removed by Reduction Rule 4.3.5.

Lemma 4.3.7. *Let $(D = (V, A), \{s_i\}_{i=1}^k, \{t_i\}_{i=1}^k)$ be a k-*Linkage *instance. Then after exhaustively applying Reduction Rule 4.3.5 we have $|V| \leq 2(d + k)$ where d is the arc-deletion distance to a funnel of D.*

Proof. Let $A' \subseteq A$ be a vertex-set such that $D \setminus A'$ is a funnel. Let V' be the set of endpoints of the arcs in A', that is $V' = \{v \mid \exists u : (u, v) \in A' \vee (v, u) \in A'\}$. We argue that every non-terminal vertex in $V \setminus V'$ is removed by Reduction Rule 4.3.5. Since all non-terminal sinks and sources are removed by Reduction Rule 4.3.5, we can assume that no such vertex exists in D.

Now let v be a non-terminal vertex in $V \setminus V'$. Since no arc of v is in A', necessarily $\text{indeg}(v) = 1$ or $\text{outdeg}(v) = 1$. We only consider the case where $\text{indeg}(v) = 1$ since the other one follows analogously. We can then apply Reduction Rule 4.3.5 to v, contracting the arc (u, v) where u is the unique predecessor of v. If $u \notin V'$, then $\text{indeg}(u) \leq 1$, otherwise u and v would form a forbidden subgraph for funnels. Since contracting the arc (u, v) does not change the indegree of u, either u is a source or we can apply Reduction Rule 4.3.5 to u. Hence, every non-terminal vertex in $V \setminus V'$ is removed by the reduction rule.

Because $|V'| \leq 2d$, and no terminal vertex is removed, after exhaustively applying Reduction Rule 4.3.5 to D we obtain a DAG with at most $2(d + k)$ vertices. □

In order to efficiently apply both reduction rules exhaustively, we can use a strategy similar to breadth-first search, where each arc is visited a constant number of times. We also observe that, after applying Reduction Rule 4.3.6, we might be able to apply Reduction Rule 4.3.5 again. Hence, to maximize the usefulness of the data-reduction rules it is not enough to apply one rule and then the other, we may need to alternate between them multiple times. However, it is sufficient to apply only Reduction Rule 4.3.5 in order to obtain at most $2(d + k)$ vertices.

We provide the pseudo-code of how one can exhaustively apply Reduction Rules 4.3.5 and 4.3.6 to a k-Linkage instance in Algorithm Kernelize. We can now show that k-Linkage admits a polynomial problem kernel with respect to arc-deletion distance to a funnel. An example of the application of the problem kernel is provided in Figure 4.6.

Algorithm Kernelize Problem kernel for k-Linkage. The function `reducible` checks if Reduction Rule 4.3.5 is applicable to a vertex.

function `Kernelize`(DAG $D = (V, A)$, sequences $S = \{s_i\}_{i=1}^k$ and $T = \{t_i\}_{i=1}^k$)
  $Q \coloneqq \{v \in V \mid \texttt{reducible}(v)\}$
  changed $\coloneqq$ `true`
  while changed **do** ▷ We may need to alternate between reduction rules.
    while $Q \neq \emptyset$ **do**
      $v \coloneqq \texttt{next}(Q)$
      $Q \coloneqq Q \cup \{u \in \text{in}(v) \cup \text{out}(v) \mid \texttt{reducible}(u)\}$
      $A \coloneqq A \cup \{(w, u) \mid w \in \text{in}(v), u \in \text{out}(v)\}$
      $V \coloneqq V \setminus \{v\}$
    $S' \coloneqq \{s_i \mid (s_i, t_i) \in A\}$
    $T' \coloneqq \{t_i \mid (s_i, t_i) \in A\}$
    $S \coloneqq S \setminus S'$
    $T \coloneqq T \setminus T'$
    if $|S'| = 0$ **then** ▷ If S did not change, no reduction rule is applicable.
      changed $\coloneqq$ `false`
    else
      $Q \coloneqq \{u \in \text{out}(s) \cup \text{in}(t) \mid \texttt{reducible}(u), s, t \in S' \cup T'\}$
  return $((V, A), S, T)$

Theorem 4.3.8. *k-Linkage admits a problem kernel with $6d \in \mathcal{O}(d)$ vertices computable in $\mathcal{O}(k \cdot (|V| + |A|))$ time, where d is the arc-deletion distance to a funnel of the input DAG.*

Proof. We apply the problem kernel using the Kernelize algorithm. Each call to Kernelize visits each arc a linear number of times. Using an adjacency list, we can execute the innermost while loop in $\mathcal{O}(|V| + |A|)$. Whenever we remove some terminal, we need to check again if Reduction Rule 4.3.5 is applicable, and this can be done by adding the neighbors from the removed terminals back to the queue and running the loop again. We can find the sets S' and T' by iterating through all arcs once. The outer loop is executed at most k times, since it is only repeated if some pair of terminals is removed. If, at some point, the number of terminal pairs is larger than the number of non-terminal vertices, we can output a trivial no instance and stop the kernelization process. In total, we obtain a running time of $\mathcal{O}(k \cdot (|V| + |A|))$.

From Lemma 4.3.7 we know that the reduced instance has at most $2(d + k)$ vertices, where d is the arc-deletion distance to a funnel of the input DAG. If $k > 2d$ and Reduction Rule 4.3.6 is not applicable, then we can reject the input instance, since every

path connecting two terminals must use some non-terminal vertex. Hence, the reduced instance has at most $6d$ vertices. □

We can now join the problem kernel with the FPT algorithm provided earlier, obtaining a faster algorithm. We note that, while the FPT algorithm needed to compute the arc-deletion distance to a funnel of the input DAG, this is no longer required after we apply the problem kernel. That is, we can simply take every non-terminal vertex in our set $|V'|$ and proceed with the dynamic programming as before. Even though the reduced DAG might have $6d$ vertices, we know from Lemma 4.3.7 that there are at most $2d$ non-terminal vertices. Hence, we know that $|V'| \leq 2d$. We formalize this below.

Corollary 4.3.9. *k-Linkage can be solved in $\mathcal{O}(16^d d^3 + k \cdot (|V| + |A|))$ time, where d is the vertex-deletion distance to a funnel of the input DAG.*

Proof. We can apply the problem kernel as described in Theorem 4.3.8 in $\mathcal{O}(k \cdot (|V|+|A|))$ time, obtaining a DAG with at most $2(d + k) \leq 6d$ vertices and, consequently $\mathcal{O}(d^2)$ arcs, where d is the arc-deletion distance to a funnel of the input DAG and k is the number of terminal pairs. If $k > 2d$ we can reject the input instance.

We then apply the FPT algorithm described in Theorem 4.3.4, with some modifications. First, we no longer need to compute that arc-deletion distance to a funnel of the input DAG, since all non-terminal vertices will be taken on the set V'. Since there are at most $2d$ non-terminal vertices on the kernelized DAG, we also know that $|V'| \leq 2d$. Hence, the table only has 4^d rows and $2d$ columns, giving us a total of $\mathcal{O}(4^d d)$ entries. Computing each entry takes $\mathcal{O}(4^d \cdot |A|) \subseteq \mathcal{O}(4^d \cdot d^2)$. The total running time is then $\mathcal{O}(16^d d^3 + k \cdot (|V| + |A|))$. □

We note here that, although the algorithm provided in the end does not directly depend on the arc-deletion distance to a funnel in order to work, we needed the concept of funnel in order to give an upper bound for the running time. That is, even though it would be possible to design and implement this algorithm without any knowledge on funnels, using this graph class we manged to better understand when and why the algorithm works in practice. In fact, the reduction rules presented here are also described in Bang-Jensen and Gutin [BJG08], yet they do not provide any form of guarantee of their effectiveness.

4.4 Further Applications

We collect here some observations with respect to the computational complexity of certain problems when restricted to funnels. The corresponding results are either observations based on existing reductions or simple brute-force algorithms with polynomial running time due to some property of funnels.

Parallel-Time Scheduling

In this variant of scheduling, the execution of a parallel algorithm is represented as a DAG $D = (V, A)$ where each vertex is one job and there is an arc (v, u) whenever job u needs the output of v in order to be executed. Additionally to the computation time of each job we also consider the communication time. That is, if two jobs v, u are executed on different processors but u requires the output of v, then we need to transfer this information from one processor to the other, which takes a certain amount of time. If u and v were executed on the same processor, however, then the communication time is zero since this information is already available.

The computation time of a job is given by the function $f : V \to \mathbb{N}$ and the communication time between two jobs is given by function $\tau : A \to \mathbb{N}$. A *schedule* s for D is a surjective partial function $s : \mathbb{N} \times \mathbb{N} \to V$ such that $s(p, t)$ is the job that processor p executes at time t, or $\perp$ if undefined. The schedule must also be consistent, that is, all dependencies of a job must be executed before the job itself is executed, and a processor can only execute one job at a time. Formally, it must satisfy the following

$$s(p,t) = v \wedge (u,v) \in A \Rightarrow \exists t' \leq t - f(u) : s(p,t') = u \vee \exists p' : s(p', t' - \tau(u,v)) = u \text{ and}$$
$$s(p,t) = u \neq \perp \Rightarrow \forall t \leq t' < t + f(u) : s(p,t') = \perp.$$

The goal is to the finding a schedule with parallel time at most k. This is given by $\max_{p \in \mathbb{N}}\{t + f(s(p,t)) \mid t = \max\{t \mid s(p,t) \neq \perp\}\}$. The problem is formally defined below.

Parallel-Time Scheduling [PY90]

Input: A directed acyclic graph $D = (V, A)$, a function $f : V \to \mathbb{N}$, a function $\tau : A \to \mathbb{N}$ and a number $k \in \mathbb{N}$.

Question: Is there a schedule s for D such that the parallel computation time of s is at most k?

If we construct the DAG of the execution of some divide-and-conquer algorithm, then we obtain a funnel: All divide steps correspond to Fork vertices, and conquer steps correspond to Merge vertices. Since divide-and-conquer algorithms are useful in practice (e.g. Mergesort), it would be interesting if we could obtain positive results by limiting the input DAG to be a funnel. Unfortunately, Parallel-Time Scheduling remains NP-hard on funnels, as can be seen in the reduction presented by Papadimitriou and Yannakakis [PY90]. It could still be interesting to know the behavior of the polynomial-time approximation algorithm (described in the same paper) on funnels.

Paths with Forbidden Pairs

When generating automated tests for programs, we are often interested in covering all branches of the code. When doing so, one needs to consider that certain combinations of branches are not possible: For example, if some variable x is divisible by three in one point, then later it cannot be a power of two. Hence, when constructing test paths we are only interested in feasible paths.

We model the program as a flow graph, with each vertex corresponding to an instruction and the arcs being the branches. Even though a program often contains cycles, the flow graph is usually converted to a DAG before being processed. The combination of impossible branches is given as a set F of vertex pairs. We then search for an (s, t)-path P (where s is the only source and t the only sink in the DAG) such that for no pair $(u, v) \in F$ at least one of u, v is not in P. The problem is formally defined below.

Path with Forbidden Pairs [GMO76]

Input: A directed acyclic graph $D = (V, A)$, with a source s and a sink t, and a set $F \subseteq V \times V$.

Question: Is there an (s, t)-path P such that for any pair $(u, v) \in F$ at least one of u, v is not in P?

While it is unlikely for a flow graph to be a funnel (as soon as we have two if-statements after each other, we obtain a forbidden subgraph for funnels), one can hope that it is at least funnel-like. The existence of complicated branch patterns makes the code much harder to read and are undesirable in practice. A simple branch pattern leads to a funnel-like structure of the flow graph.

One can easily obtain a polynomial-time algorithm if the input DAG is restricted to be a funnel. Since a funnel has at most $|A|$ paths, one can test each path for the existence of two vertices of the same pairs. This can be done in $\mathcal{O}(|A| \cdot (|P| + |V|))$ time.

PERT Networks

PERT (Program Evaluation and Review Technique) networks are sometimes used for planning projects. The project is described as a DAG, with each task being an arc and each vertex being a state in which certain tasks have been completed. Sometimes, however, the tasks can be given as vertices instead, and we might want to convert them to arcs. That is, we receive as input a DAG $D = (V, A)$ and want to construct another DAG $P = (V', A')$ such that for each vertex in V (that is, for each task) there is a corresponding arc in A'. Furthermore, we need to keep the dependencies consistent, that is, there is a vertex-path from u to v in D if and only if there is an arc-path between the corresponding arcs in D'. Since it is not always possible to have $|A'| = |V|$, the goal is to minimize the number of "dummy" activities. The problem is formally define below.

PERT Network [KD79]

Input: A directed acyclic graph $D = (V, A)$ and an integer k.

Question: Is there a PERT-Network corresponding to D, that is a digraph $P = (V', A')$ such that for each vertex in V there is a corresponding arc in A' and for any two vertices $u, v \in V$ there is a (u, v)-vertex-path in D if and only there is a (u, v)-edge-path in P and $|A'| \leq |V| + k$?

If a project is divided into a planning phase and an execution phase, then the corresponding PERT network is a funnel: Vertices in the planning phase can have outdegree greater than one, and those in the execution phase can have indegree greater than one.

Krishnamoorthy and Deo [KD79] proved that PERT Network is NP-hard. Since the DAG in the reduced instance is a funnel, the problem remains NP-hard even if we restrict the input to be a funnel.

Chapter 5

Implementation and Experiments

In this chapter we empirically evaluate some of the algorithms described in previous sections. Since k-Linkage on DAGs is W[1]-hard with respect to the number of terminals, but in FPT with respect to the arc-deletion distance to a funnel, we consider the linear problem kernel provided in Section 4.3 to be of practical significance. In order to compare the practical effectiveness of the kernel with the theoretical guarantee, we need to be able to compute the distance to a funnel of the input DAG.

Since we have a upper bound for the number of vertices of a kernelized DAG which depends on its arc-deletion distance to a funnel, we need to know the size of this parameter in order to analyze the effectiveness of the problem kernel. In Section 3.2 we provided an approximation algorithm as well as two FPT algorithms with respect to solution size. We implemented all three algorithms for computing the arc-deletion distance to a funnel, comparing both exact ones with each other and using them to evaluate how good the approximation algorithm is in practice.

The experiments were divided into two categories, each in turn divided into groups. The first category consists of experiments related to the arc-deletion distance to a funnel (Section 5.2), while the second concentrates on the linear problem kernel for k-Linkage (Section 5.3).

Before proceeding to the experiments, we first describe in Section 5.1 how we obtained the data used in the experiments and also explain some of the basic data structures used both in the experiments about the arc-deletion distance to a funnel of DAGs (Section 5.2) and in the experiment on k-Linkage (Section 5.3).

5.1 Basic Technical Aspects

We describe here some general aspects which apply to all experiments. This includes how we obtained and preprocessed DAGs, the development environment for the implementation as well as the data structures we used. Further details specific to each experiment are given in Sections 5.2 and 5.3.

Multiple experiments were executed in order to empirically determine the behavior of the implemented algorithms in different settings. The first canonical experimental

data one can use is random data: It has the big advantage of being easily obtainable in large quantities, but also the disadvantage that it often does not represent real data very well. We also used DAGs from real-world scenarios, yet often needing to preprocess them in order to obtain an acyclic digraph. Even though this decreases the relevance of the results, the DAGs obtained still do not resemble random data and may still provide some information about how the algorithms behave in practice.

We downloaded certain digraphs from the Konect database [Kun13]. The dataset includes source code dependencies (linux, subelj-jdk and subelj-jung-j), food webs (foodweb-baydry, foodweb-baywet, maayan-foodweb), interactions between animals (moreno-cattle, moreno-mac, moreno-sheep), and citation networks (subelj-cora). We also obtained the dependency graph of packages from the Arch Linux distribution (arch-packages). In this digraph, each vertex corresponds to a package and there is an arc (v, u) when package v depends on package u.

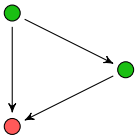
We first argue why some of the considered instances should have some resemblance to a funnel. In the case of dependencies graphs (linux, arch-packages, subelj-jdk), we can expect the DAG to be divided into roughly two types of vertices: Libraries and applications. That is, some packages or source files have few dependencies, but are used by many other packages or files. This is often the case with libraries which provide some core functionality. In contrast, applications (or front-end source files) often need many different libraries, yet are not required by many other packages since they interact directly with the user. Hence, libraries should correspond to MERGE vertices of a funnel, while applications are FORK vertices.

In food webs we can sometimes classify animals as being hunters or herbivores. We can expect few other animals to hunt other predators since these animals are often faster and harder to catch. Hence, vertices corresponding to predators should have a large outdegree but a small indegree. In contrast, herbivores should almost never hunt other animals, but they might be prey for the predators. Vertices corresponding to herbivores should therefore have a larger indegree but a small outdegree.

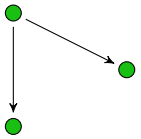
We preprocessed the data in the following way. First, we greedily identified cycles through breadth-first search. We then collapsed each cycle into a single vertex. That is, for a cycle $v_1 v_2 \dots v_k v_1$ we added the arcs $\{(v_1, u) \mid u \in \text{out}(v_i), 1 \leq i \leq k\}$ and $\{(u, v_1) \mid u \in \text{in}(v_i), 1 \leq i \leq k\}$. We then removed the vertices $v_2, v_3 \dots v_k$ and the arc (v_1, v_1). We repeat this procedure until the input becomes a DAG. We note here that, regardless of the order in which we collapse the cycles, the obtained DAG is the same. After applying this, we split the DAG into weakly-connected components, and then removed components with less than six vertices.

This preprocessing step can strongly change the input digraph, yet in some cases we can argue that it does not destroy its structure completely. For example, if the digraph describe some dependency relationship, then as soon as some vertex v depends on one vertex of some cycle C, then v depends on all vertices of C. Hence, we can treat a cycle as a single vertex since all members of the cycle have effectively the same neighborhood.

The algorithms used for generating random DAGs (including funnel-like DAGs) are described in Section 5.1.1. Reading this section is only necessary if the reader wishes to



(a) There are $2^3 = 8$ different funnels with this labeling.



(b) There are only 6 funnels with this labeling: Three with one arc, two with two arcs and one with no arcs.

Figure 5.1: Green vertices receive the label FORK, red ones receive MERGE. Different labelings may allow for a different number of funnels, yet labelings are drawn uniformly at random. In this example, we considered a fixed topological ordering.

know from which distributions the random DAGs were generated.

5.1.1 DAG Generation

Since not many publicly available datasets of DAGs were found, we decided to generate random DAGs according to certain parameters. In this section we describe how the DAGs were generated.

One relevant property that a random generator should have is that, for any two DAGs D_1, D_2 that satisfy the input parameters of the generator (e.g. number of vertices, average degree, etc.), the probability of generating D_1 should be the same as the probability of generating D_2. Here we consider only labeled DAGs, that is, two isomorphic DAGs are considered to be different if their vertex-sets or arcs-sets are different.

By fixing some topological ordering of the vertices, it is easy to generate uniformly at random DAGs with n vertices and m arcs. We first set $V \coloneqq \{0, 1, \ldots, n-1\}$. We can only add an arc (v, u) to the digraph if $v < u$, otherwise the topological ordering would not be respected. Hence, there are $n(n-1)/2$ different arcs. We then generate the arc-set A by taking m distinct arcs from the set of all possible arcs.

If we want to generate uniformly at random a DAG with average degree s, for example, we can simply set $m \coloneqq ns$ and then use the procedure above. This method also has the advantage of taking time proportional to the output size (that is, number of vertices and arcs) instead of depending on the number of potential arcs. This can make a big difference on running time when generating sparse DAGs with many vertices.

Generating funnels uniformly at random, however, is not as simple as generating DAGs. Any vertex can have either indegree greater than one or outdegree greater than one, but not both at the same time. If we just iteratively draw arcs, the random choices can no longer be independent from each other. This can easily lead to a biased distribution. What we do instead is to first assign each vertex a random label (FORK or MERGE) and then only add arcs which do not violate the label of a vertex. Unfortunately, we this method we lose control on the exact number of arcs that the output funnel has, since different labelings allow for different numbers of arcs to be added.

The generator receives as parameter the number of vertices and the density of the funnel (that is, the proportion of arcs that must be added given some labeling). This

allows us to generate funnels uniformly at random for a given labeling. The labeling, however, is drawn uniformly at random from all $2^{|V|}$ possible labelings, yet this method does not consider how many different funnels exist with a given labeling. Hence, funnels with fewer arcs have a larger chance of being generated than funnels with many arcs (when compared to the chances in a uniform distribution. Refer to Figure 5.1 for an example). We consider this bias to be a harmless for the experiments since the number of arcs is not decisive for the running time. Furthermore, most labelings permit a large number of arcs to be added, so the bias should not have a big impact on the size of the generated funnels. Unfortunately, counting how many funnels exist with a given labeling is not easy (except for trying all possibilities), and so obtaining an unbiased method for generating funnels is also not easy. For this reason we decided to generate funnels using the simple, yet not uniform, distribution described above.

Finally, in the experiments we are actually interested in funnel-like DAGs instead of funnels. We obtain a funnel-like DAG by first generating a funnel $D = (V, A)$, with $V = \{1, 2, \ldots, n\}$, and then adding d random arcs to it, with d being given as a parameter to the generator. These arcs are taken from the set $\{(u, v) \mid 1 \leq u \leq v \leq n\} \setminus A$, which contains all arcs which can be added to D without breaking the topological ordering of the vertices in V. While this does not guarantee that the arc-deletion distance to a funnel of the generated DAG is equal to d, we know it must be at most d.

5.1.2 Runtime Environment and Data Structures

The algorithms were implemented in Haskell 2010 [Mar]. Haskell is a purely functional programming language with a strong type system. We consider this paradigm to be adequate for the implementation because the algorithms in this work are described in a way which is much closer to a mathematical or functional description than to an imperative one. Hence, it is easier to verify that the algorithms were implemented correctly. Haskell also has a very strict type system, which statically detects many potential errors on the code, making debugging less time-consuming.

One characteristic of Haskell is that pure code has no side-effects (that is, functions only return some value and cannot change any variable). This has the advantage of avoiding many programming mistakes, but also the disadvantage of making certain optimization strategies more complicated. For example, it is not possible to use pointers inside pure code since changing some value is a side-effect. This is often not an issue, but in some cases the running time of an algorithm had to be increased by a logarithmic factor. Since we do not deal with huge datasets, this should not have a significant impact on the running time of the experiments.

For the implementation of the digraph data structure we used *sets* and *maps* from the `containers` package. Sets and maps allow for lookup, insertion and deletion in $\mathcal{O}(\log n)$ time. A digraph is then stored as a map, where each key is the ID of a vertex, and the value associated with a key contains a set of inneighbors and another of outneighbors. This allows us to remove arcs and to insert vertices and arcs in $\mathcal{O}(\log |V|)$ time. Removing a vertex v takes $\mathcal{O}((\text{indeg}(v) + \text{outdeg}(v)) \cdot \log |V|)$ time.

The experiments were run on an Intel® Xeon® E5-1620 3.6 GHz processor with

64 GB of RAM. The operating system was GNU/Linux, with kernel version 4.4.0-67. For compiling the code we used GHC version 7.10.3.

5.2 Arc-Deletion distance to a funnel

In Section 3.2 we described two exact algorithms with different branching strategies for computing the arc-deletion distance to a funnel of a DAG: One which branches on arcs and has a running time of $\mathcal{O}(5^d \cdot (|V|^2 + |V| \cdot |A|))$ (defined in Section 3.2.1), and another which branches on labels and has a running time of $\mathcal{O}(3^d \cdot (|V| + |A|))$ (defined in Section 3.2.3). For conciseness, we call the first algorithm the *arcs strategy*, and the second one the *labeling strategy*. The names come from the behavior of the algorithm: The arcs strategy searches for forbidden subgraphs and branches on all possibilities of destroying them, while the labeling strategy branches instead on the labels the vertices receive.

By the Big-O notation it is clear which one is faster in the worst case, yet in practice we are often interested on the average case instead of the worst case. For this reason, we compare in Section 5.2.2 both algorithms in order to determine which one is in practice faster. We expect the labeling strategy to be considerably faster than the arc strategy, specially because of the data-reduction rules it uses. However, the labeling strategy does not work on general digraphs, and so even if the other strategy is slower, it has the advantage of being more general.

In Section 5.2.3 we analyze the running time of the faster algorithm and the arc-deletion distance to a funnel on randomly generated DAGs. Since we do not expect the arc-deletion parameter to be small in such instances, we also generate in Section 5.2.4 random funnel-like DAGs and repeat a similar experiment on them. Finally, in Section 5.2.5 we analyze how large the parameter is in real-world data. We also analyze in all experiments how good the approximation algorithm from Section 3.2.2 is.

Before proceeding to the experiments, we describe in Section 5.2.1 some heuristics and pruning rules used in the implementation of the exact algorithm.

5.2.1 Heuristics and Pruning Rules

Both search-tree algorithms considered in this chapter solve a decision problem (that is, they only answer "`yes`" or "`no`"). For the experiments, we want to be able to compute the actual distance to a funnel of the input DAG. To obtain this, one can start with $d = 0$ and check if there is a solution of size at most d. If not, the budget k is increased by one. Since we know there is always a solution if k is the number of arcs, we also know the algorithm stops when $k = d$, where d is the arc-deletion distance to a funnel of the DAG. Furthermore, this increases the worst-case running time only by a factor of 2, which means the algorithm is still in FPT with respect to d. Nevertheless, properly "guessing" the correct value for k can have a big impact on the practical running time of the algorithm.

We describe here some of the heuristics used to speed-up the search-tree algorithm

for computing the arc-deletion distance to a funnel. A straightforward pruning rule is the lower bound provided in Section 3.2.4, which allows a branch of the search tree to be rejected before evaluating all of its leaves. Search trees are often slower when the input is a no-instance, since in order to be able to answer "`no`" one has to check all leaves. Hence, being able to reject some branch without computing the entire subtree can have a huge impact on performance.

We implemented a simpler version of the lower bound, considering only single-cores (i.e. vertices with in- and outdegree greater than one). The reason for this simplification is that it is easier to check the degree of a vertex than iterating through its successors. We do not expect this to have a great impact on the lower bound, since we only ignore subgraphs with arc-deletion distance to a funnel equal to one. Furthermore, the reduction rules for the labeling strategy perform well on such subgraphs, and we assume the gain obtained by implementing the lower bound as described in Section 3.2.4 would not be very high.

As an heuristic for the branching rule, we avoid branching on a vertex which is used by the lower bound, unless there is no other alternative. This is done in an attempt to only remove arcs which were not considered by the lower bound, thus not reducing it.

Finally, when deciding which branch to take first (that is, whether we first try to remove incoming or outgoing arcs), we adopt a greedy strategy. That is, when branching on a vertex v, we try to first remove incoming arcs if $\text{indeg}(v) \leq \text{outdeg}(v)$. Otherwise we branch first on removing outgoing arcs of v. In a sense, this mimics the behavior of the approximation algorithm, we the difference that we also need to branch on which arcs are removed.

With respect to heuristics, we believe there is plenty of room for improvements. For example, the labeling strategy could first branch on vertices where we only need to guess its label (since the arcs are removed by the reduction rule), thus potentially decreasing the number of children per node of the search tree for no-instances. Another possible optimization is to delete vertices from which we know no further arc will be taken into the solution. As soon as the DAG becomes disconnected, one can consider the connected components individually. The challenge of implementing this optimization is properly dividing the budget k for the individual connected components. By doing this correctly, it should be possible to avoid testing all solutions for one component with all solutions from the other component.

5.2.2 Branching Strategy Comparison

Our first experiment consisted of testing if the labeling strategy is in practice faster than the arc strategy. For $n \in \{50, 200\}$ and $s \in \{10, 50, 100\}$ we generated 30 funnels with n vertices and density 0.5, and then added s random arcs (respecting the topological ordering) to them. This gives us a total of 180 DAGs. The labeling strategy solved all of them within 10 minutes, while the arc strategy only solved 85 (47%) instances in the same time range.

When only 10 arcs were added, the arc strategy managed to solve all instances. However, when 100 arcs were added it did not solve any instance within 10 minutes.

Strategy	$n = 50$	$n = 200$
Arcs	51 (29.8 s)	34 (55.1 s)
Labeling	90 (0.1 s)	90 (0.2 s)

Table 5.1: Number of instances solved by each strategy within 10 minutes. The number in braces correspond to the average running time.

The number of vertices also made a big difference on the number of solved instances (Table 5.1). In contrast, the labeling strategy solved all instances within 7.1 s, with an average running time of 175 ms. From this we can already conclude that the labeling strategy is much faster on DAGs than the arc strategy, and so in the experiments that follow we only consider the faster strategy.

5.2.3 Randomly Generated DAGs

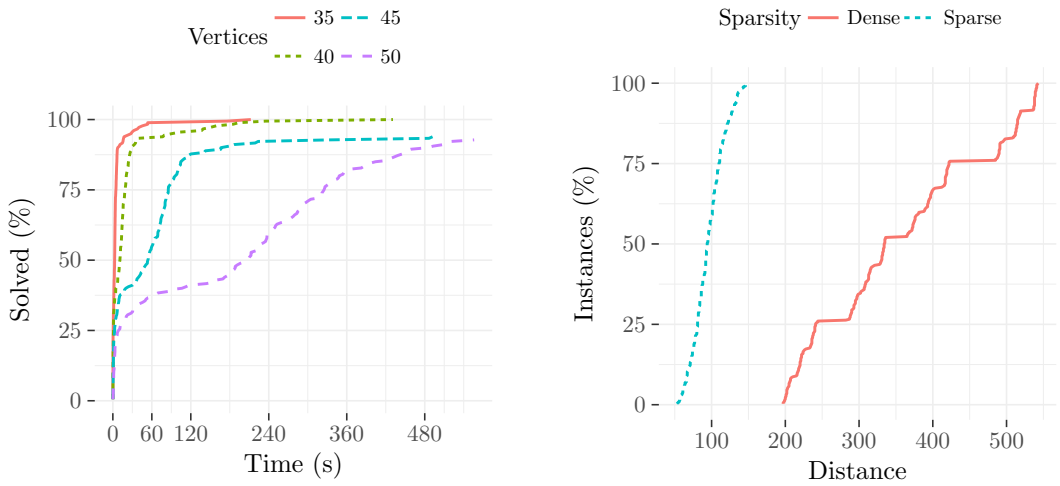
The first type of random data we considered was general random DAGs. We expect the arc-deletion distance to a funnel of such DAGs to be fairly large, since almost every vertex has, with high probability, in- and outdegree greater than one. Here we are also interested in verifying the effect that the sparsity of the DAG has on the running time. While on sparse DAGs the arc-deletion distance to a funnel should be smaller, on dense ones we need to remove multiple arcs per vertex, making the search tree also shorter.

We generated random DAGs with different numbers of vertices and different average degrees. For $n \in \{35, 40, 45, 50\}$ and $D \in \{1, 2, 3\}$ we generated 30 DAGs with n vertices and $n(D + 5)$ arcs (that is, average outdegree $D + 5$) and 30 also with n vertices but with $n(n/2 - D)$ arcs. This gives us a total of 720 DAGs, and for 696 (97%) we managed to compute the correct distance within 10 minutes. In this section, we consider a DAG to be *dense* if its density is greater than 0.5 and *sparse* otherwise (recall that the density of a DAG is the ratio between the number existing arcs and the maximum amount of arcs the DAG can have).

Running time

We first analyze how many instances can be solved within a given time range. To this end, we plot a cumulative curve of the percentage of instances solved in relation to the running time (Figure 5.2a). From this we can read the percentage of instances we can solve if we decrease the maximum allowed time for the algorithm. We plotted one curve for each value of n used.

The amount of vertices clearly had a big impact on the running time, with only 74 (22%) instances with 50 vertices being solved within two minutes, against 178 (49%) instances with 35 vertices. This can be easily explained by the fact that the average degree was fixed, and so a higher amount of vertices also implies a larger distance to a funnel.



(a) Amount of instances solved within a certain time-range. The algorithm solved 141 (20%) instances within one second, and 463 (64%) within thirty seconds.

(b) Amount of instances with a distance to a funnel below a certain value. On sparse DAGs the distance varied between 53 and 148, while on dense DAGs it was between 197 and 543 arcs.

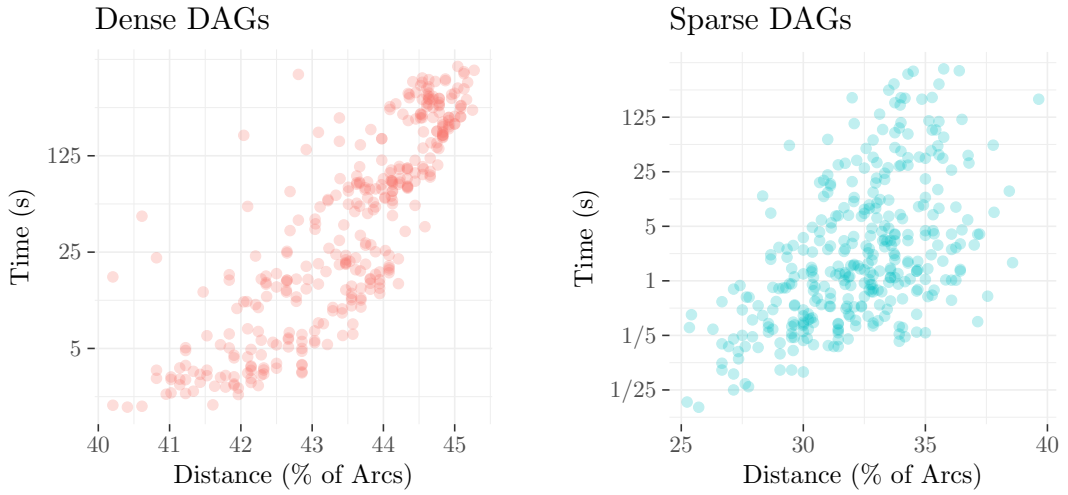
Figure 5.2: Cumulative computation time and arc-deletion distance to a funnel.

Arc-deletion distance

To analyze the arc-deletion distance to a funnel of the generated DAGs we plotted the cumulative curve of the percentage of instances (only of the solved ones) in relation to the distance (Figure 5.2b). For dense DAGs, the curve obtains the shape of a staircase. This happens because the distance increases considerably with the number of vertices, and so each large horizontal jump corresponds to an increase in the number of vertices. We can also see that the distance on this dataset was quite large, with all dense DAGs having a distance between 197 and 543. Considering that 463 (64%) instances were solved optimally within 30 seconds, this means that the practical running time was considerably below the theoretical worst-case time-bound of $\mathcal{O}(3^d \cdot (|V| + |A|)$. We can find two explanations for this. First, the branching algorithm could remove multiple arcs per branch, since the average degree was fairly large (at least 6). Hence, the depth of the search tree was definitely much smaller than the distance d. The second explanation is that each vertex is considered by the branching rule at most once. On this dataset, the number of vertices was much smaller than the distance to a funnel. Thus, the depth of the search tree was at most $3^{|V|} \leq 3^{50}$. Even though 3^{50} is still fairly large, pruning and data reduction rules can make the tree considerably smaller.

Relation between running time and distance

As the running time in practice is much shorter than the theoretical guarantee predicts, it is interesting to try to empirically understand what influences the speed of the algorithm. We compared the running time with the percentage of arcs that needed to



(a) The proportion of arcs that needed to be removed has a large impact on the running time on dense DAGs.

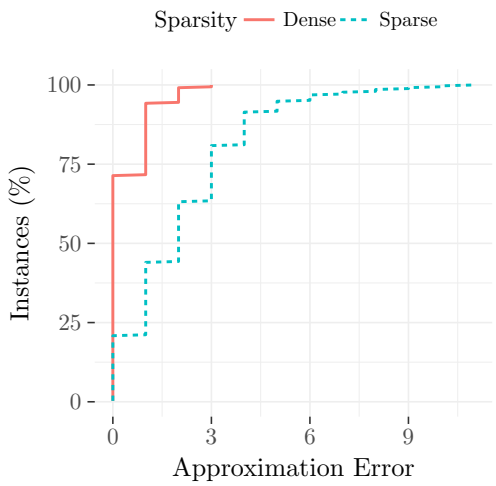
(b) On sparse DAGs, the proportion of removed arcs also had an impact on the running time, but other random factors also had a high influence.

Figure 5.3: Running time versus percentage of arcs that needed to be removed. A logarithmic scale was applied to the time axis.

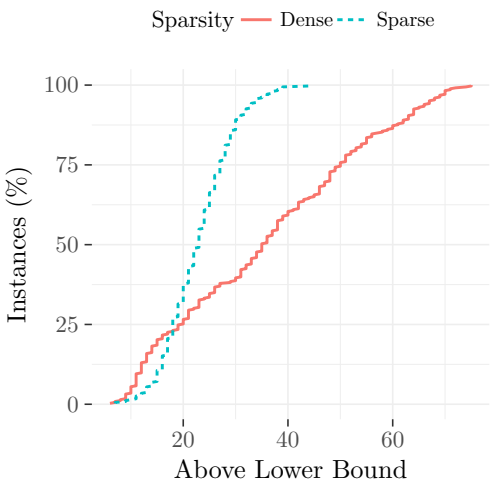
be removed in order to turn the input into a funnel (Figures 5.3a and 5.3b). Since the running time is exponential on the distance, we used a logarithmic scale for the time axis. Despite the large variance, one can see a certain correlation both in dense as in sparse DAGs, although this correlation is stronger on dense ones. Indeed, the Pearson correlation coefficient between the logarithm of the running time and the relative solution on dense DAGs is 0.83 and on sparse DAGs it is 0.55. We consider the correlation to be significant in both cases since many other factors should also influence the running time. The logical explanation for this behavior is that, as the relative solution approaches 50%, the effectiveness of heuristics and reduction rules decreases. In a sense, if half of the arcs need to be removed, the DAG bears no resemblance to a funnel and the algorithm cannot do anything better than guessing. We note here that the case where the relative solution equals 50% (that is, we have a complete DAG) can be quickly solved by the algorithm. Thus, at some point the problem becomes easier as the proportion of arcs that need to be removed increases. It is unclear whether the algorithm becomes slower when the relative solution is decreased below some threshold.

Approximation and lower bound

Even though the exact algorithm was much faster than expected, the instances considered here are relatively small. Trying to compute the arc-deletion distance to a funnel of a random DAG with 1000 vertices, for example, would take a very long time. In such cases, the approximation algorithm might be a better choice. For this reason, it is



(a) The approximation error was smaller on dense DAGs. On 337 (48%) instances the approximation found the optimal solution.



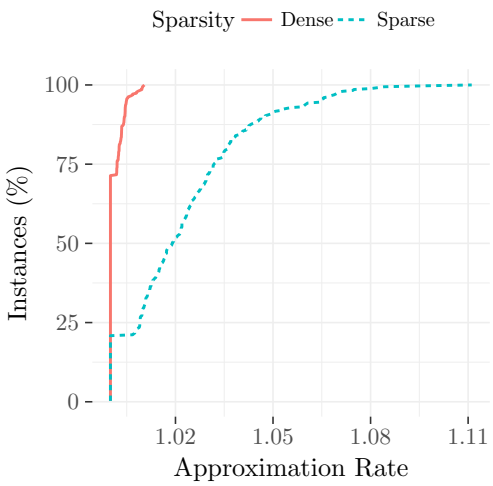
(b) Lower bound was closer to the correct value on sparse DAGs.

Figure 5.4: Accuracy of the approximation algorithm and the lower bound on DAGs of different density.

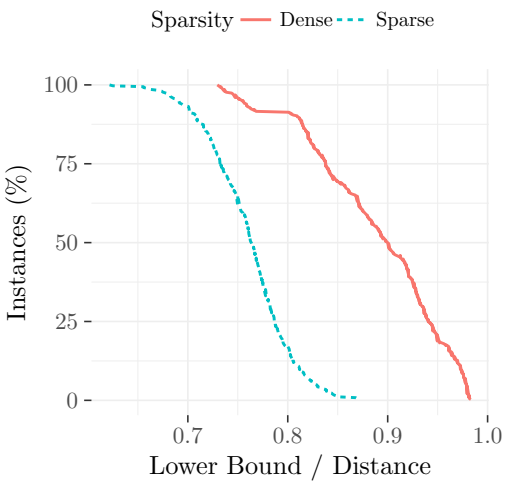
interesting to know how good the approximation factor is in practice.

We plotted the cumulative percentage of instances in relation to the approximation error (Figure 5.4a). The approximation error is given by the difference between the real distance and the distance computed by the approximation. The approximation error was very small, being considerably smaller than the factor-3 guarantee from Theorem 3.2.9. In particular, 247 (71%) of the dense DAGs were solved optimally by the approximation algorithm. Interestingly, this number drops to 73 (21%) on sparse DAGs. Since the distance is much larger on dense than on sparse DAGs, this suggests that the approximation performs much better on dense than on sparse DAGs. One possible explanation is that, if for a vertex x there is a large difference between $\text{indeg}(v)$ and $\text{outdeg}(v)$, then the label that v receives in an optimal solution depends mostly on the degree of v and not so much on its neighborhood. On dense DAGs, vertices which come earlier in the topological ordering have a larger outdegree, and those in the end have larger indegree. Since the approximation considers mainly the degree of a vertex when assigning a label, the decisions taken by this algorithm on dense DAGs are almost always optimal. This is, however, clearly not the only reason for the small approximation error, since even on sparse DAGs the approximation was very close to the real distance. We consider a deeper theoretical analysis to be of interest, specially finding sufficient conditions for the approximation to be optimal.

For the lower bound we made a similar plot, considering how many arcs were removed additionally to what the lower bound predicted (Figure 5.4b). The lower bound was not as close to the real distance as the approximation, but it still fairly close the real distance. In order to properly determine whether it performs better on sparse or dense DAGs, we



(a) The approximation rate was very close to one. In this case, the approximation error is more meaningful.



(b) The lower bound was not as close to the real distance as the approximation, but on 376 (52%) instances it was at least 80% of the real distance.

Figure 5.5: Quality of the approximation and the lower bound relative to the real distance. A value closer to one is better. Both are closer to the real distance on dense DAGs.

need to consider the relative error, that is, the lower bound divided by the real distance (Figure 5.5b). While not as close to the real distance as the approximate distance, the lower bound achieved fairly good ratio. On 316 (91%) dense DAGs the ratio was at least 80%, and on sparse DAGs such a ratio was achieved only on 60 (17%) instances. Nevertheless, it was always greater than 62%, likely having a great impact on the running time and being one of the reasons why the exact algorithm successfully solved (within reasonable time) instances with an arc-deletion distance to a funnel of around 500.

With a similar plot we can see that the approximation rate was very small (Figure 5.5a). On average an rate of 1.01 was achieved. For this reason, we consider the approximation error to be a more meaningful measure for the quality of the approximation.

Interestingly, both the approximation algorithm as well as the lower bound were closer to the real distance on dense DAGs than on sparse ones. This is surprising, since on dense DAGs one can expect the choices of which arcs to take in the arc-disjoint forbidden subgraphs to be more relevant. On sparse DAGs, the forbidden subgraphs should share less arcs, and so the arbitrary choices of the lower bound algorithm should have less impact. One explanation for this behavior is that the optimal solution for sparse DAGs might have a more complicated structure. That is, it might be better to disconnect the DAG or take a decision which is bad locally in order to decrease the number of arcs removed globally. This type of structure can easily lead both algorithms to a worse solution.

Estimated distance

To better understand the behavior of the approximation and the lower bound, we attempt to predict the arc-deletion distance to a funnel of a DAG given the parameters used to generate it. Considering Figure 5.2b, one can see that the distance is very regular with respect to the sparsity. On closer inspection it is also possible to see that the curve for dense DAGs has multiple jumps. These jumps actually happen when the number of vertices or the average degree change.

In order to estimate the distance of a DAG given the average degree and the number of vertices, we consider a simple greedy strategy for removing arcs: For the first $\lceil n/2 \rceil$ vertices (according to some topological ordering) we remove incoming arcs, and for the remaining we remove outgoing arcs. This clearly produces a funnel, since it is equivalent to assigning the label FORK to the first vertices and MERGE to the remaining. Because of the topological ordering, we do not need to remove arcs from MERGE to FORK vertices.

To determine the expected number of arcs that the algorithm above removes we first calculate the probability that some arcs starts or ends at some vertex v. Let $\{0, 1 \ldots, n-1\}$ be the set of vertices in their topological ordering and m the number of arcs of a DAG. Let $N = \binom{n}{2}$. For a vertex v we define $p_{\text{in}}(v)$ as the probability of drawing some arc (u, v) and $p_{\text{out}}(v)$ as the probability of drawing some arc (v, u). Because this is how we generated the random DAGs, these probabilities are correct for the instances used in this experiment. Formally, we have

$$p_{\text{in}}(v) = v/N \qquad\qquad p_{\text{out}}(v) = (n-1-v)/N.$$

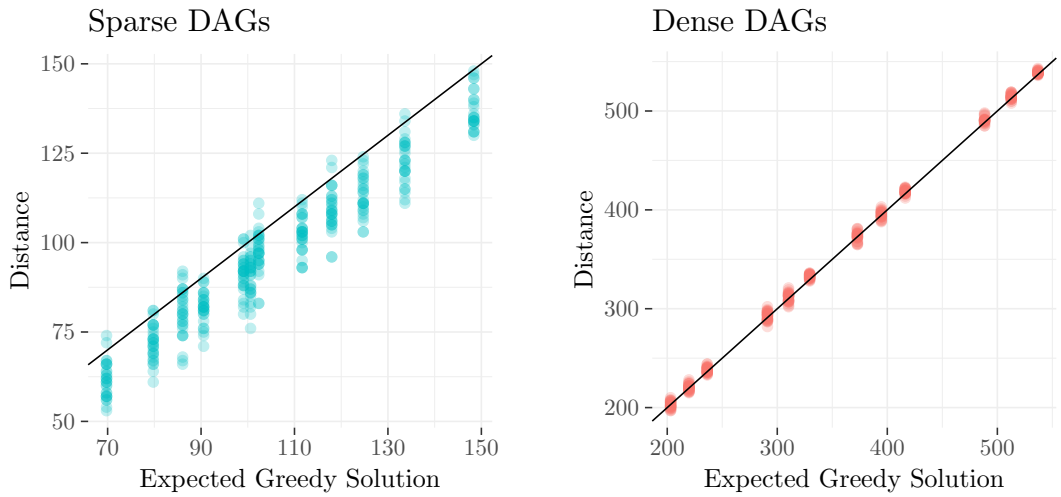
The expected number of arcs we need to remove from a vertex v is then given by $m \cdot \min(p_{\text{in}}(v), p_{\text{in}}(v)) - 1$ or zero if this number is negative. We now search for two values a, b such that a is the first vertex with $m \cdot p_{\text{in}}(a) > 1$ and b is the last vertex where $m \cdot p_{\text{out}}(b) > 1$. Formally, we obtain

$$\begin{aligned} m \cdot p_{\text{in}}(a) = ma/N > 1 &\Leftrightarrow a > N/m \\ m \cdot p_{\text{out}}(b) = m(n-1-b)/N > 1 &\Leftrightarrow b < n - 1 - N/m. \end{aligned}$$

Hence, $a = \lfloor N/m \rfloor + 1$ and $b = n - 2 - \lfloor N/m \rfloor = n - 1 - a$. We also define c as the last vertex where $m \cdot p_{\text{in}}(c) \leq m \cdot p_{\text{out}}(c)$. This is given by $c = \lfloor (n-1)/2 \rfloor$. With these definitions, the expected number of arcs removed by the greedy strategy described is

$$\begin{aligned} \hat{d} &= \sum_{v=a}^{c} (m \cdot p_{\text{in}}(v) - 1) + \sum_{v=c+1}^{b} (m \cdot p_{\text{out}}(v) - 1) \\ &= m/N \left(\sum_{v=a}^{c} v + \sum_{v=c+1}^{b} (n-1-v) \right) + a - b - 1. \end{aligned}$$

In Figures 5.6a and 5.6b we compare how accurate the estimator $\hat{d}$ was. On dense DAGs the expected value is very close to the real distance. On sparse DAGs, the error is still small, but other random factors influenced the distance, pulling the distance



(a) The real distance increases together with the expected distance given by the simple greedy algorithm.

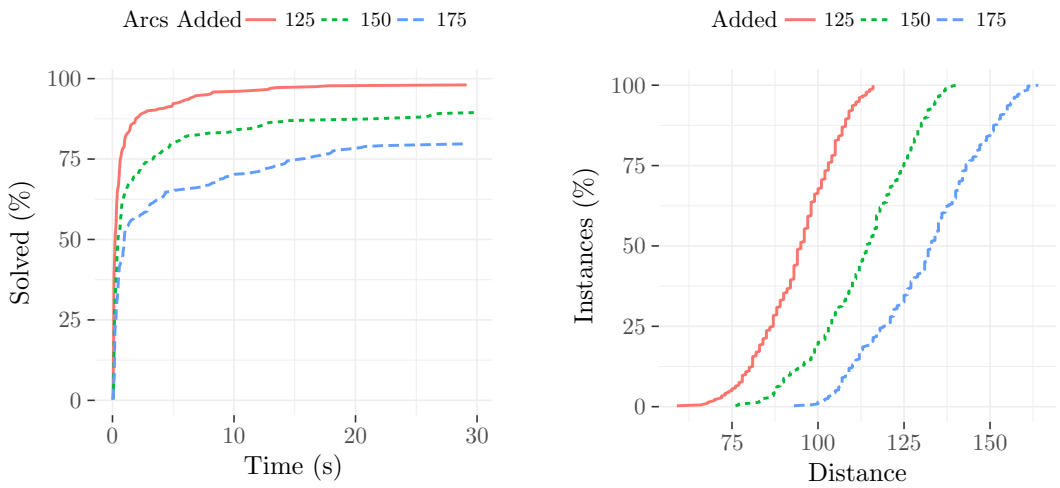
(b) Estimated distance was very close to the real distance on dense DAGs.

Figure 5.6: Estimated distance $\hat{d}$ compared to the real distance. The average error was 3.

below the estimated value. We attribute this to the fact that on sparse DAGs the solution is not very local: One can more easily create connected components, new sinks or new sources in order to obtain a funnel. When comparing Figures 5.6a and 5.6b to Figure 5.4a, we can see that the estimator loses accuracy precisely on the same cases where the approximation error is higher and the lower bound ratio is worse: On sparse DAGs. This supports our hypotheses about why the approximation algorithm and the lower bound perform worse on sparse DAGs. The estimator $\hat{d}$ corresponds to a very local solution. Since the actual distance of sparse DAGs was often smaller than the estimated distance, this implies that it was possible to decrease the number of arcs removed globally by making certain decisions which were not locally optimal. We did not, however, find any significant correspondence between the approximation error and the estimation error, and we also note that the approximation was considerably more accurate than the estimator on sparse DAGs (average error of 2.1 against 9 of the estimator).

Conclusion

From this first batch of experiments we can already see that the running time of the exact algorithm for computing the arc-deletion distance to a funnel of a DAG is much shorter than what the theoretical bound predicts. We also conclude that the error of the approximation algorithm is very small, making it suitable for practical applications where the input is too large for the exact algorithm.



(a) Percentage of instances which were solved within a certain time-range. Only 962 (89%) instances which were solved within thirty seconds were plotted. For 706 (65%) instances the distance was computed within one second.

(b) Percentage of the solved instances within a certain arc-deletion distance to a funnel. The median distance was 111.

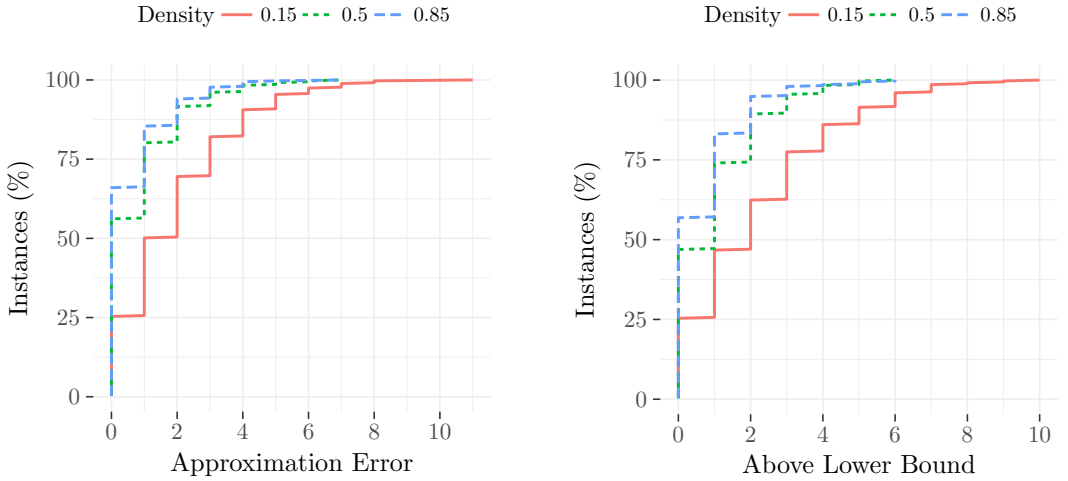
Figure 5.7: Running time and distance of the generated instances.

5.2.4 Funnel-like DAGs

Even though completely random DAGs already give us some insight about the behavior of the implemented algorithms, from a parameterized point of view they often represent cases where an FPT algorithm should perform poorly. That is because an FPT algorithm is constructed assuming that the input has a certain structure which makes some parameter small. By their random nature, random DAGs often have no special structure and, as seen in Section 5.2.3, their arc-deletion distance to a funnel is not very small. For this reason, we also considered DAGs with a relatively small arc-deletion distance to a funnel.

For $n \in \{250, 300, 500, 1000\}$, $D \in \{0.15, 0.5, 0.85\}$ and $s \in \{125, 150, 175\}$ we generated 30 funnels with n vertices and density D, and then added s random arcs which respected the topological ordering to each of them. This gives us a total of 1080 DAGs. From this amount, 1059 (98%) had their arc-deletion distance to a funnel computed within 10 minutes.

Running time We plotted a cumulative curve with the percentage of instances solved within a certain time range (Figure 5.7a). Most instance were solved fairly quickly: Within 15 seconds 932 (86%) instances were solved optimally. We can also observe that there were essentially two types of instances: Easy ones which were solved within few seconds, and harder ones which took much longer. That is, if we limit the running time to five seconds we can solve 855 (79%) instances, and if we increased it to sixty seconds



(a) Gap between the approximate distance and the actual distance. In 502 (48%) instances the approximation found the correct distance.

(b) Gap between the lower bound and the actual distance. In 449 (43%) instances the lower bound was equal to the actual distance.

Figure 5.8: Effectiveness of the lower bound and the approximation algorithm in determining the arc-deletion distance to a funnel.

we can solve only 142 additional instances.

In order to compare the measured running time to the theoretical worst-case upper bound of $\mathcal{O}(3^d \cdot (|V| + |A|))$, we first need to know how large the solution size was. We then plotted a cumulative curve with the percentage of solved instances with an arc-deletion distance to a funnel below a certain value (Figure 5.7b). Clearly, the number of added arcs influenced the distance directly, yet the variance was quite high: The shortest distance was 59 arcs, and the longest was 164. The median distance was 111. Considering that $3^{111} \approx 9 \cdot 10^{52}$, the running time given by the theoretical worst-case analysis would be of multiple millennia. Clearly, in practice the search-tree constructed by the algorithm does not need to be exhaustively evaluated. With certain heuristics, pruning and reduction rules it is possible to considerably increase performance.

Approximation and lower bound

In this context, it is interesting to analyze how both the lower and the upper (given by the approximation) bound behave in this dataset. We first consider the absolute error of both bounds (i.e., the difference between the bound and the actual arc-deletion distance to a funnel). In Section 5.2.3 we concluded that the sparsity of a DAG had a big impact on the quality of both the approximation algorithm and the lower bound. Hence, we compare the error of both bounds with the density of the generated funnel (Figure 5.8). We immediately see they were both very close to the correct value. In fact, considering that the real distance was often greater than 100, we can see that both measures are off by a very small amount. This comes in contrast with the proven

approximation factor of 3. Since the DAGs used here are already close to funnels, most decisions of the approximation algorithm end up being correct. Having correct local information helps the approximation making a globally optimal decision, and so it is no surprise that the approximation factor in funnel-like DAGs is much better than the theoretical bound. This explanation is supported also by the fact that both bounds performed worse on sparse funnels than on dense one. The approximation yielded an optimal solution on 89 (25%) instances with density 0.15, while for density 0.85 it found an optimal solution on 231 (66%) instances. This behavior is similar to what we observed on the experiments with random DAGs, and we attribute the reason again to be the more complicated structure of the solution on sparse DAGs.

We also call attention to the fact that the both plots from Figure 5.8 are very similar, even though the approximation algorithm and the computation of the lower bound are based on completely different strategies. The former is based on labeling vertices, while the latter uses forbidden subgraphs.

We compared the approximation error with the difference between the lower bound and the arc-deletion distance to a funnel of each instance. To this end, we made a density plot of the amount of instances in relation to the gap between the arc-deletion distance to a funnel and each bound (Figure 5.9). The accuracy of the bounds does not appear to be correlated. That is, a high approximation error does not imply a weaker lower bound. This could be related to the fact that on 776 (73%) instants both bounds had an error of at most two. From such a small range it is not possible to draw any conclusions for the correlation of both bounds. The lack of correlation is, in this case, actually positive, since a heuristic can then benefit from both bounds. We also call attention to the fact that in 310 (28%) instances both bounds found an optimal solution. This means that, for such instances, it is possible not only to find an optimal arc-deletion set in linear time, but also provide, again in linear time, a certificate that such arc-set is indeed optimal.

Relation between distance and added arcs With respect to what influences the relationship between the distance to a funnel and the number of added arcs, one can expect that the density of a funnel has a big impact on this relation. That is, random arcs added to a dense funnel have a higher chance of destroying the funnel property.

We investigated the distribution of the arc-deletion distance to a funnel of the generated instances in relation the their density and the number of added arcs. This information was then represented with a box plot (Figure 5.10a). In such plots, each box contains half of the samples and the horizontal bar in it stands for the median value. The whiskers can be interpreted as the certainty interval and have a length of at most 1.5 times the inter-quartile range (i.e., the difference between the third and first quartile). Instances outside this range were plotted as individual dots. For a fixed number of added arcs, the distance increases together with the density. We can also see that, for dense funnels, the arc-deletion distance was somewhat close to the number of arcs added. Yet, the density is not the only relevant factor.

When adding random arcs to a funnel, we draw arcs from all possible arcs a DAG can have. As the number of vertices increases, so does the number of possible arcs.

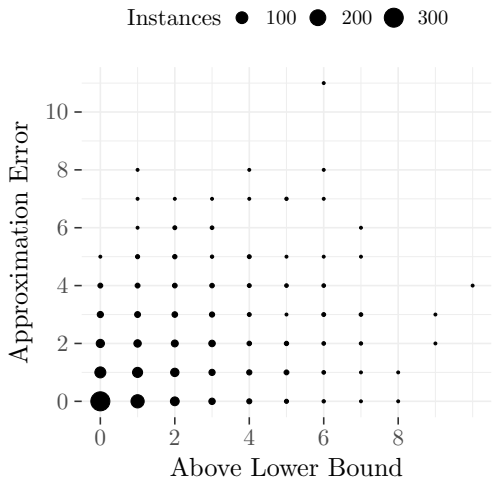


Figure 5.9: Relationship between approximation error and lower bound accuracy. The size of the dot indicates how many instances lie on that point. In 301 (28%) instances the lower bound met the approximate solution.

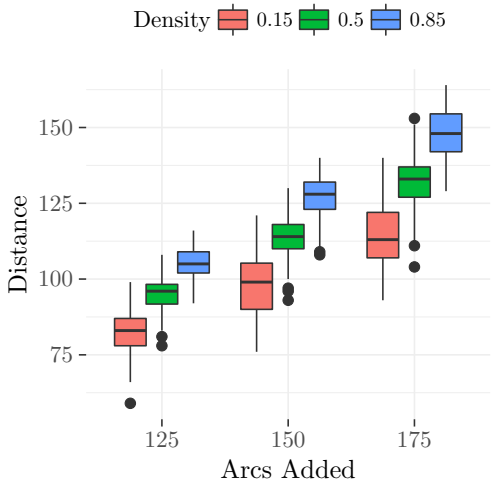
Since the density is fixed, the proportion of arcs which can be added (that is, which are not already there) and respect the labeling of the funnel decreases. Recall that a DAG with n vertices has at most $\binom{n}{2}$ arcs and a funnel with n_f FORK vertices and n_m MERGE vertices has at most $n_f n_m + n_f + n_m - 2$ arcs. Let $N = \binom{n}{2}$ and $M = n_f n_m + n_f + n_m - 2$. The number of arcs that can be added is $N - MD$, where D is the density of the funnel. The amount of arcs that violate the labeling is given by $N - MD - M(D-1) = N - M$. Hence, if we add s arcs to a funnel, we estimate the arc-deletion distance to be

$$\hat{d} = s \cdot \frac{N - M}{N - MD}.$$

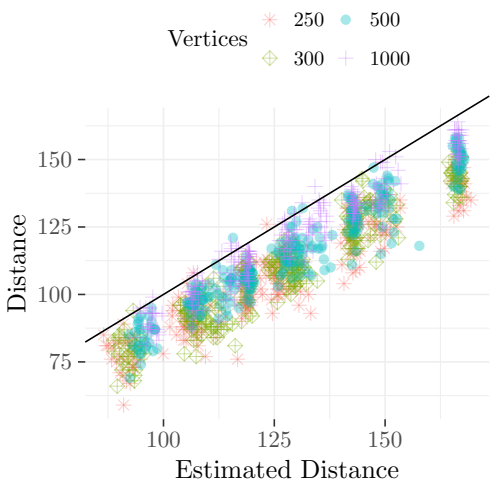
Intuitively, we expect to remove all added arcs except those which respect the labeling of the funnel. Clearly, this is not exact, since we are allowed to remove arcs which originally belonged to the funnel, possibly obtaining a different funnel. This means that $\hat{d}$ will generally overestimate the actual distance d. In Figure 5.10b we empirically verified how good the estimator is in practice. One can see that $\hat{d}$ grows linearly with d, even though $\hat{d}$ is almost always larger. Further, the average error of the estimator was 14. We also note that the error decreased when the number of vertices increases. This is likely due to the fact that s is fixed, and so the probability that the arcs added belong to a similar funnel should decrease as the number of arcs and vertices of the original funnel increases.

Conclusion

The findings from this experiment reinforces those from Section 5.2.3. Even on larger DAGs the exact algorithm was capable of quickly finding a solution. The approximation



(a) Impact of the density of a funnel on the arc-deletion distance of the DAG obtained after random arcs were added. Increasing density clearly increased the distance for a fixed number of arcs added. The large variance indicates that density is not the only important factor.



(b) How close is the estimator $\hat{d}$ to the real arc-deletion distance. The black line is the function $x = y$ and represents the cases where the distance was correctly estimated.

Figure 5.10: Relationship between number of random arcs added and the arc-deletion distance to a funnel.

error was again very small, and for funnel-like DAGs the lower bound was very close to the real distance.

5.2.5 Real-World DAGs

We now consider DAGs obtained from real-world scenarios. This data should represent much better than random DAGs the behavior of the algorithms in practice. At the same time, however, it might be biased towards specific areas and applications due to the fact that the sample size is very small.

For each DAG, we computed a lower bound and an approximation of its arc-deletion distance to a funnel. We also attempted to compute the real distance, stopping the algorithm if no solution was found within ten minutes.

The dataset was clearly divided into small DAGs (Table 5.2) and relatively large DAGs (Table 5.3). On large DAGs the exact algorithm did not find a solution within 10 minutes. For the small DAGs, however, we can verify that the approximation obtained an optimal solution in all cases. This reinforces the findings of the previous sections where the approximation rate was very close to one. Furthermore, we can use the lower bound in order to find the worst-case approximation rate for the instances to which we do not know an optimal solution. From this we see that even on large instances the approximation rate is fairly small. We note here that in the experiments in the previous

Instance	$\|V\|$	$\|A\|$	Lower Bound	Distance	Approximation
foodweb-baydry	26	82	5	6 (7%)	6
foodweb-baywet	26	82	5	6 (7%)	6
maayan-foodweb	156	1197	118	129 (10%)	129
moreno-cattle	7	13	2	3 (23%)	3
moreno-mac	24	163	52	60 (37%)	60
moreno-sheep	6	10	1	1 (10%)	1

Table 5.2: Instances which were solved optimally within ten minutes (also within one second). Numbers in braces represent the distance relative to the number of arcs.

Instance	$\|V\|$	$\|A\|$	L. Bound	Approximation	Ap. Rate
arch-packages	9710	26218	2529	2921 (11.1%)	≤ 1.16
linux	29810	103475	4845	5376 (5.2%)	≤ 1.11
subelj-cora	18061	44214	5054	5897 (13.3%)	≤ 1.17
subelj-jdk	6025	27868	2177	2326 (8.3%)	≤ 1.07
subelj-jung-j	5730	26477	2086	2236 (8.4%)	≤ 1.11

Table 5.3: Instances which were not solved by the exact algorithm within 10 minutes. Numbers in braces represent the distance to the number of arcs.

sections the approximation was much closer to the optimal solution than the lower bound, which means that the real distance is probably much closer to the approximate value than to the one given by the lower bound.

With respect to the size of the parameter in practice, it appears to be a small proportion of the total number of arcs of the DAG, even though in absolute value it is not very small. Furthermore, in most cases where we argued why the DAG should have a small distance to a funnel (except in arch-packaes) the proportion of arcs that needed to be removed in order to turn it into a funnel was at most 10%. This supports our argumentation about why these DAGs should have some similarity to a funnel.

Considering the absolute value of the distance to a funnel, the parameter might not be very useful in practice for an FPT algorithm. That is, an algorithm with running time $\mathcal{O}(2^d \cdot \text{poly}(n))$, where d is the arc-deletion distance to a funnel of the input, could take 2^{3000} steps to finish when run on the large DAGs analyzed here. This running time might not be very practical. If we consider, however, the distance relative to the number of arcs (that is, the arc-deletion distance divided by the total number of arcs), we can see that only a small portion of the arcs had to be removed. Hence, the arc-deletion distance parameter might be useful in practice when coupled together with a problem kernel. This motivates us to empirically determine the effectiveness of the linear-sized problem kernel from Section 4.3, which is done in the next section.

5.3 k-Linkage

For the k-Linkage problem we do not consider completely random DAGs. On such DAGs one can expect every vertex to have both in- and outdegree greater than one (if the average degree is large enough), and so the kernel described in Section 4.3 will not have any effect. Instead, we verify the effectiveness of the kernel in Section 5.3.2 for funnel-like DAGs. In Section 5.3.3 we apply the problem kernel to real-world DAGs. In both cases, however, the terminals in the k-Linkage instances were generated randomly. This was done because no publicly-available real-world instances were found. How the instances were generated is explained in Section 5.3.1.

5.3.1 Terminal Generation

For k-Linkage we need, additionally to the input DAG, also a sequence of terminals which must be pairwise connected. Simply picking p random pairs can often lead to trivial no-instances if, for example, there is no path connecting both vertices. There is also the problem that no vertex can appear twice in the sequence of terminals, since this also trivially leads to a no-instance.

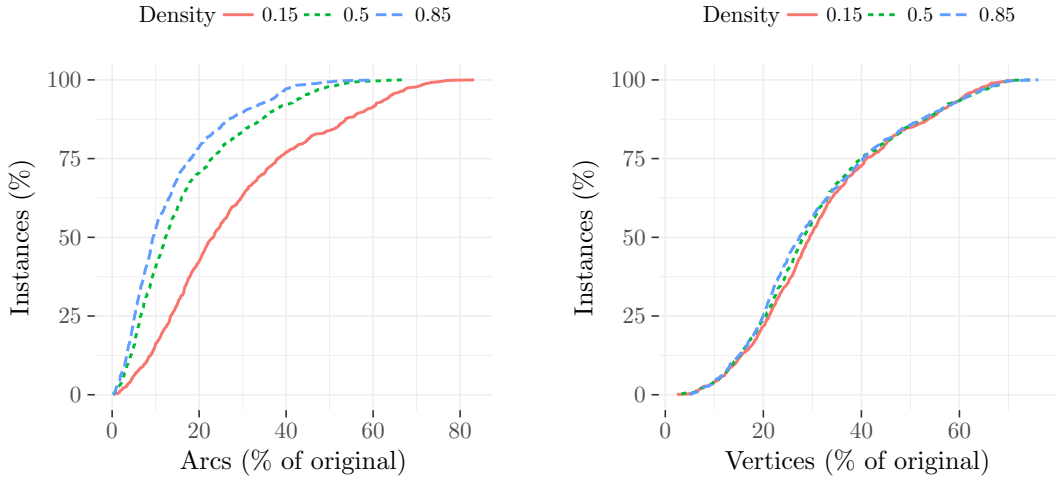
The method we used to generate k-Linkage instances works as follows. The generator receives a number p and a DAG D, and then chooses p sources from D at random, possibly choosing the same vertex twice. We exclude vertices with both in- and outdegree equal zero. Then, for each chosen source s, some sink t is chosen uniformly at random from the set of all sinks which are reachable by s. Vertices which are selected twice are copied together with their arcs. That is, if we choose a pair (s, t) then we add new vertices s', t' together with the arcs $\{(s', v) \mid v \in \text{out}(s)\}$ and $\{(v, t') \mid v \in \text{in}(t)\}$. We then add (s', t') to the set of pairs. Finally, we remove all vertices which do not appear in any (s_i, t_i)-path, where s_i and t_i are two terminals.

By taking only sources and sinks into the sequence of terminals we guarantee that the terminals themselves do not block any path between other terminals. This prevents the construction of trivial no-instances where a vertex v with outdegree one and its successor are chosen as terminals for different pairs, prohibiting the existence in the solution of any path starting at v.

We also decided to allow taking the same vertex twice as a terminal in order to make the random choices independent from each other. Duplicating the terminals is equivalent to allowing the endpoints of one path to be shared with other paths. This allows us to always produce the desired number of pairs, regardless of the structure of the DAG.

5.3.2 Funnel-like DAGs

We verify here the effectiveness of the problem kernel from Theorem 4.3.8 on randomly generated funnel-like DAGs. Even though on such DAGs we have certain control over their distances to funnel and, hence, over the effectiveness of the kernel, these DAGs allow us to measure the effect of other parameters (e.g. density) on the behavior of the kernel.



(a) Amount of instances where the relative number of arcs was reduced below a certain value after the kernel was applied.

(b) Amount of instances where the relative number of vertices was reduced below a certain value after the kernel was applied.

Figure 5.11: Effectiveness of the kernel with respect to the relative decrease of the input size.

For $n \in \{250, 500, 750, 1000\}$, $s \in \{150, 200, 250\}$, $D \in \{0.15, 0.5, 0.85\}$ and $p \in \{10, 20, 30\}$ we generated 30 funnels with n vertices and density D, added s random arcs to each of them and randomly selected p pairs of sources and sinks. This gives us a total of 3240 k-Linkage instances, each on a different DAG. From this amount 153 (4.8%) were solved by the kernel, whereas in 55 (1.8%) instances a solution was found and in 98 (3.1%) there was no solution. Such instances can be considered to have size zero after the kernelization process, and so we only consider them for the running time. The remaining statistics only refer to the unsolved instances. On average, applying the problem kernel took 48 ms, and each instance was kernelized in less than 480 ms. We do not further analyze the running time since the measurement errors can be quite high on such a short time-range.

Decrease on input size

We first measure the relative decrease of the input size after the instances were kernelized. We plot a cumulative curve of the percentage of instances against the proportion of vertices (and arcs) which remained after the kernelization process (Figures 5.11a and 5.11b). Instances were grouped by the density of the generated funnel.

The size of the instance, both in terms of arcs as in vertices, decrease considerably in most cases. On 2247 (73%) instances the kernelized DAG had at most 25% of the original arcs. On 2635 (85%) instances the number of vertices was decreased by at least half, and in 1227 (40%) instances it was 25% of the original value. Since the number of arcs is quadratic in the number of vertices, it is no surprise that the decrease on the

number of arcs was stronger than on the number of vertices.

The density had barely any effect on the proportion of vertices which were kept by the kernel. For the proportion of arcs, the kernel was somewhat less effective on DAGs where the density of the generated funnel was smaller. Here we recall that the generated k-Linkage instances modified the DAG by removing unnecessary vertices and also by duplicating the terminals. Thus, the funnel density used on the plots does not necessarily correspond to the funnel density of the DAG to which the kernel was applied. Nevertheless, the difference between instances with density 0.15 and those with density 0.85 is significant. Our explanation to this is that on dense funnels the kernelization process will often remove arcs when contracting an arc (v, u), since arcs from the inneighbors of v to u (or from the outneighbors of u to v) are more likely to exist than on sparse funnels.

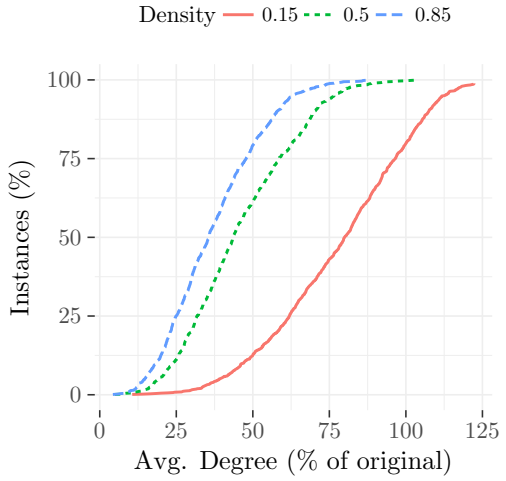
With respect to the meaningfulness of such values, consider the performance boost obtained if the problem kernel is used as a preprocessing step for some exponential-time algorithm. If the $\mathcal{O}(k!\,|V|^k)$ algorithm described by Bang-Jensen and Gutin [BJG08] is used to solve the k-Linkage instance after this problem kernel is applied, then halving $|V|$ possibly decreases the running time from $k!\,|V|^k$ to $k!(|V|/2)^k = 2^{-k}k!\,|V|^k$, making the algorithm 2^k times faster, which can be a big improvement in practice.

Impact of the distance to a funnel

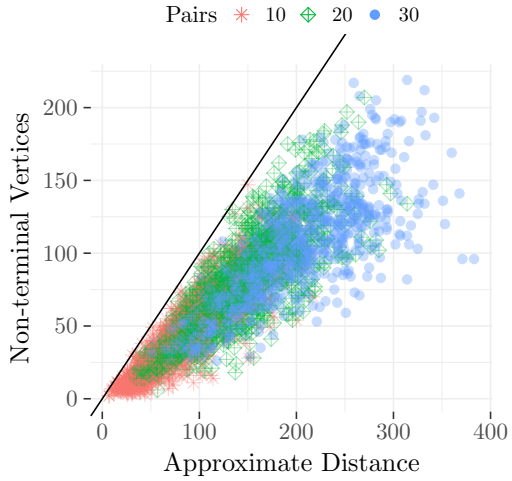
In order to determine the effectiveness of the kernel we also need to know how large the arc-deletion distance to a funnel of the instances was. Because the instances were changed after they were generated (a simple reduction rule was applied and vertices present in some of the pairs were duplicated), the actual distance might differ largely from the number of random arcs added.

Even though we know a kernelized instance has at most $\mathcal{O}(d)$ vertices, in practice it is relevant to know how big the constants are. Theoretically, the kernel outputs a DAG with at most $6d$ vertices (or $2(d + p)$ if $p \leq d$). However, in practice this number can be much smaller, specially considering that the kernelization process does not need to know an arc-deletion set in order to be applied.

We compared the number of non-terminal vertices of the kernelized instances with the arc-deletion distance to a funnel computed by the approximation algorithm from Section 3.2.2 (Figure 5.12b). From the experiments in Sections 5.2.3 and 5.2.4 we know the approximation rate is very close to one, and so the plot should barely change if we were to compute the real distance to a funnel. We often obtain much less than $2d$ non-terminal vertices after applying the kernel. On average, the kernelized instances had $0.7(d + p)$ vertices, with p being the number of terminal pairs of the kernelized instance. This ratio remained fairly stable, and in 374 (12%) instances we even obtained less than $0.5(d + p)$ vertices after applying the kernel. This means that, even if the distance to a funnel of a DAG is not very small, the problem kernel might still considerably reduce the input size.



(a) Amount of instances which had the average degree reduced by at most a certain percentage. Four instances where the average degree was more than 1.25 times larger were not plotted.



(b) Number of non-terminal vertices after applying the kernel compared to the approximate distance. On instances below the black line the number of non-terminal vertices was smaller than the arc-deletion distance to a funnel.

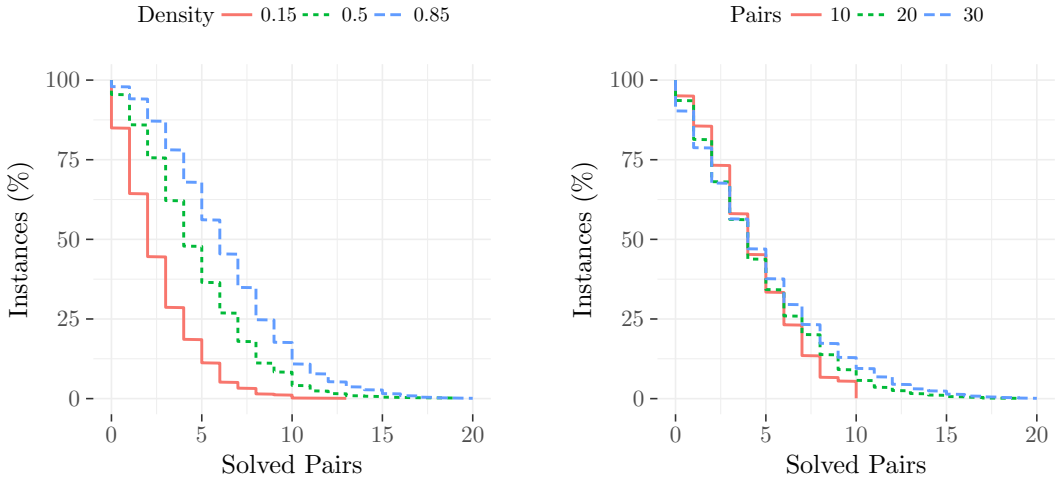
Figure 5.12: Effectiveness of the kernel with respect to average degree and to the size of kernelized instance.

Average degree

Interestingly, the average degree was also reduced in many cases (Figure 5.12a), with 1592 (52%) instances having their average degree decreased to at most 50% of the original value after the problem kernel was applied. The density of the funnel appears to have a large impact on the change of the average degree. Indeed, while 202 (2%) instances with density 0.15 had their average degree increased by the kernel (up to 1.4 times larger than the original value), this number drops to 3 (0.1%) in instances with density 0.5 or 0.85. A possible explanation for this behavior is that on DAGs with smaller (funnel) density the number of arcs which exist between vertices which are kept by the kernel (that is, which are an endpoint of some removed arc) is smaller than on dense DAGs. Since the vertices kept by the kernel includes those which are allowed to have both in- and outdegree greater than one, these vertices contribute to a larger average degree. Hence, the average degree might increase if the density is very low. On DAGs with a large (funnel) density, however, there are many arcs connecting vertices which are kept by the kernel to vertices which are removed. On such cases, the removed vertices possibly also decrease the degrees of the remaining ones.

Solved pairs

Because k-Linkage is in XP with respect to the number of terminal pairs, it is also interesting to know how many pairs were solved by the kernel. We plotted the cumulative



(a) On denser DAGs, a slightly larger number of terminal pairs were solved by the kernelization process.

(b) Initial number of pairs did not have much influence on the amount of solved ones.

Figure 5.13: Percentage of instances where the kernelization process solved at least a certain number of pairs.

percentage of instances in relation to the number of pairs solved by the problem kernel (Figures 5.13a and 5.13b). Interestingly, the number of solved pairs does not appear to be affected by the initial amount of pairs, but by the funnel density of the generated DAG. For denser DAGs, one can expect that the paths in a solution for the k-Linkage instance are shorter. In particular, the likelihood that there is an arc from a source to a sink after the kernel is applied is larger on dense DAGs. Hence, it is intuitive that a smaller number of pairs was solved on sparser DAGs.

It is surprising, however, that the number of pairs generated does not influence the number of pairs solved by the kernelization process. This could be related to the way that the terminal pairs are generated. Since we duplicate sources and sinks which are selected multiple times, increasing the number of pairs chosen also increases the chance that some source or sink is selected twice. When duplicating sources and sinks, we potentially increase the degrees of many vertices, which in turn can prevent such vertices from being removed by the data reduction rules.

Conclusion

The problem kernel for k-Linkage was very effective on funnel-like DAGs, with many instances having their sizes halved. We also verified that the number of non-terminal vertices on the reduced instance can be much smaller than the arc-deletion distance to a funnel of the DAG, making the problem kernel useful in practice even if the arc-deletion distance to a funnel of the input DAG is not very small.

DAG	`yes`	`no`	Unsolved
arch-packages	0	3	177
foodweb-baydry	0	120	60
foodweb-baywet	0	116	64
linux	0	178	2
maayan-foodweb	0	1	179
moreno-cattle	0	180	0
moreno-mac	0	120	60
moreno-sheep	180	0	0
subelj-cora	0	178	2
subelj-jdk	0	3	177
subelj-jung-j	0	1	179
Total	180	900	900

Table 5.4: Number of instances solved by kernelization process. The column `yes` stands for instances where a solution was found, and in the column `no` we have instances where no solution exist.

5.3.3 Real-World DAGs

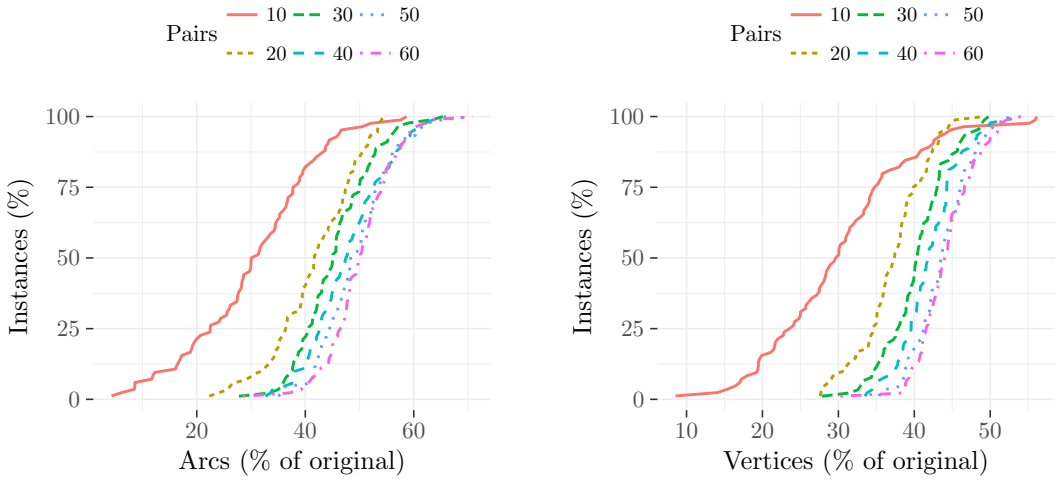
In order to better understand how the problem kernel for k-Linkage works in practice, we also used real-world DAGs for the instances. Unfortunately, we still needed to generate the terminal pairs randomly since no publicly available datasets were found. We can nevertheless compare the differences between using randomly generated DAGs and real-world ones, yet we only gain limited insight about the behavior of the problem kernel on practically relevant k-Linkage instances.

For $p \in \{10, 20, 30, 40, 50, 60\}$ and for each DAG D in the dataset we obtained we generated 30 k-Linkage instances on D with p terminal pairs. These instances were then preprocessed in order to remove irrelevant vertices. This gives us a total of 1980 k-Linkage instances which were kernelized. On average, applying the problem kernel took 2 ms.

Solved Instances

Differently from what we observed on random funnel-like DAGs, the kernel optimally solved 1080 (55%) instances (Table 5.4). Interestingly, it appears that the DAGs were divided into two groups: Some easily solvable by the kernel and those where the kernel solved very few instances.

In the case of moreno-sheep, the original DAG has only six vertices, and the only source is directly connected to the only sink, making all instances trivial yes-instances. The moreno-cattle DAG also has few vertices (only seven) yet two sinks aree not directly connected to the only source. When generating too many pairs, this easily leads to no-instances. Some other DAGs are also fairly small (foodweb-baydry, foodweb-baywet,



(a) Percentage of kernelized instances with at most a certain amount of arcs.

(b) Percentage of kernelized instances with at most a certain amount of vertices.

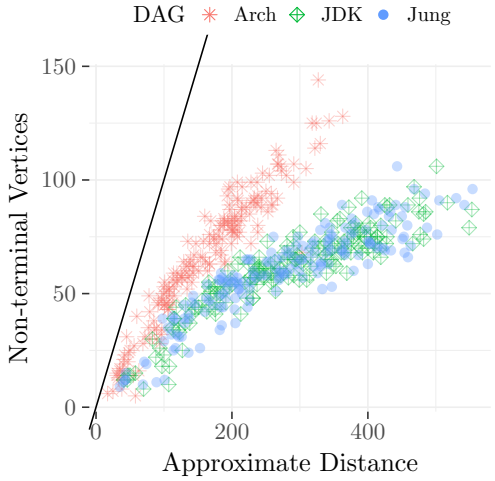
Figure 5.14: Reduction of the input size by the kernelization process.

moreno-mac), leading to many trivial no-instances. Finally, the maayan-foodweb has a special structure which made the kernel useless in most cases: Most vertices with indegree or outdegree equal to one are directly connected to the only source or the one of the two sinks in the DAG. Hence, when generating the terminal pairs we also increased the in- or outdegree of such vertices, making the reduction rules no longer applicable on them. The remaining DAGs are too large to be visually analyzed.

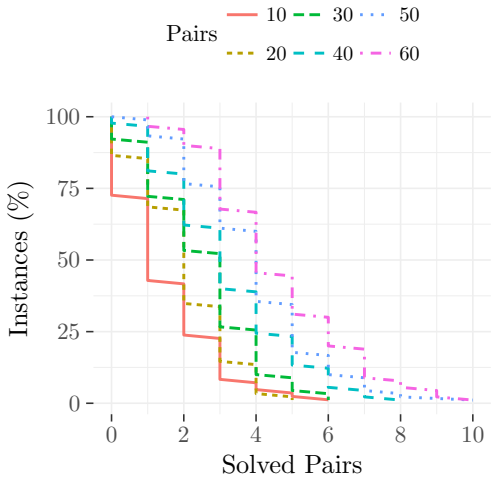
We call attention to the fact that linux and subelj-cora have more than 10000 vertices and 40000 arcs, yet almost all instances using such DAGs were solved by the kernel. This means that these DAGs must have some structure which makes most k-Linkage instances to be no-instances. This supports the findings from Section 5.3.2 where the number of terminal pairs did not affect how many pairs were solved by the kernelization process. Apparently the structure of the DAG plays a much more critical role on the existence of some solution than the terminal pairs chosen. As many DAGs in the dataset have an poor structure for k-Linkage, we only consider arch-packages, subelj-jdk and subelj-jung-j for subsequent analyses. This leaves us with 540 instances.

Decrease on input size

We plot the cumulative curve with the percentage of instances in relation to the relative size of the kernelized instance (Figures 5.14a and 5.14b). The size of the input DAG decreased considerably in many cases, with 516 (97%) instances having 50% of the original number of vertices or less. Unlike the experiments with random funnel-like DAGs, the relative decrease on the number of arcs was smaller than on the number of vertices, with only 372 (70%) instances having at most 50% of the original number of arcs after being kernelized. This is probably due to the fact that the DAGs used



(a) Relation between the number of non-terminal vertices on the reduced instance and its arc-deletion distance to a funnel. Names are abbreviated as follows: Arch (arch-packages), JDK (subelj-jdk) and Jung (subelj-jung-j).



(b) Percentage of kernelized instances where at least a certain number of pairs were solved.

Figure 5.15: Impact of the arc-deletion distance to a funnel on the size of the kernelized instance, and effectiveness of the kernel with respect to the number of solved pairs.

were rather sparse, and so contracting arcs often reduces the number of arcs by a small amount.

The number of pairs appears to have some impact on how much the instance sizes were decreased by the kernel. One can identify two groups: Instances with 20 terminal pairs or more, and those with only 10 pairs. The kernel was considerably more effective on instances with only 10 terminal pairs, yet between 20 and 60 pairs there is a comparatively small difference. This possibly happens because with few pairs we might rarely choose the same source or sink during the terminal generation process, and so the degrees of the remaining vertices is not increased. Yet as soon as we start choosing the same source or sink multiple times, it does not matter whether we choose it twice or three times: In both cases the kernelization process might not be able to remove the out- or inneighbors of the chosen terminals.

Impact of the distance to a funnel

From Section 5.2.5 we know the arc-deletion distance to a funnel of the DAGs used here is not very small, at least if we consider the absolute value. Since the instances were preprocessed and some unnecessary vertices were removed, it could be that the arc-deletion distance to a funnel decreased. Nevertheless, it is interesting to know how the distance to a funnel of the input DAGs influenced the effectiveness of the kernel. For each instance we compared the number of non-terminal vertices with its approximate

arc-deletion distance to a funnel (Figure 5.15a). Instances are grouped by the DAG used.

The number of non-terminal vertices in the kernelized instances was considerably smaller than the arc-deletion distance to a funnel. Theoretically, the problem kernel guarantees at most $2d$ non-terminal vertices, where d is the arc-deletion distance to a funnel of the input DAG. Yet, in practice, in 246 (46%) instances we obtained $0.25d$ non-terminal vertices. This is considerably less than what we observed on funnel-like DAGs. Even though the k-Linkage instances are not real-world instances, this result indicates that the problem kernel might be effective in practice even if the arc-deletion distance to a funnel of a DAG is not very small. We can also see that the number of non-terminal vertices grows linearly with the arc-deletion distance to a funnel, meaning that the distance is a decisive factor for the effectiveness of the kernel.

Solved Pairs

Finally, we measure the number of terminal pairs which were solved by the kernelization process. We consider the cumulative curve of the percentage of instances where at least a certain number of pairs was solved (Figure 5.15b). Similar to funnel-like DAGs, the initial number of terminal pairs had a relatively small influence on the number of pairs solved by the problem kernel. We believe the reason to be the same: By duplicating sources and sinks when generating terminals, we can prevent the neighbors of the terminals to be removed by the reduction rules.

Conclusion

The dataset used is mostly ill-suited for the k-Linkage problem, with a very large number of instances being trivially solvable due to structural properties of the DAGs. From the remaining instances, we can see that the problem kernel managed to considerably reduce the input size, being much more effective than the theoretical worst-case upper bound predicted.

Chapter 6

Conclusion and Outlook

We defined a new subclass of directed acyclic graphs (DAGs) which we called *funnels*, analyzing it from a graph-theoretical as well as from an algorithmic point of view. We provided different ways of characterizing this class, some of them being more suitable for algorithms and others more suitable for proofs of combinatorial properties. We observed that funnels have a very interesting structure and can be analyzed from considerably different perspectives. We did not consider how certain DAG parameters (e.g. directed tree-width) behave on funnels and believe this to be an interesting direction for future work. We did, however, define four new DAG parameters involving funnels: arc- and vertex-deletion distance to a funnel, and funnel depth and height.

From the four distance-to-funnel parameters defined, we studied in greater depth the arc-deletion distance to a funnel. We provided a fixed-parameter (FPT) algorithm with respect to solution size for computing the arc-deletion distance to a funnel of a digraph. We also described a factor-3 approximation algorithm which runs in linear time, but we do not know if the approximation bound is tight, that is, the algorithm might actually be a factor-2 approximation. We do have, however, a concrete example where a factor of 2 is achieved. With respect to the computational complexity of the arc-deletion problem, we do not know if it is NP-hard. We made some attempts in proving NP-hardness, yet none of them was very promising. At the same time, we obtained certain theoretical results which might indicate that the problem lies in coNP. Since we know it is in NP, proving that it is also in coNP would imply that it is unlikely to be NP-hard, unless NP = coNP, which is not expected. As the evidence for containment in coNP is not strong enough, we only conjecture that the arc-deletion problem is not NP-hard.

After studying some properties of funnels and of some measurements for the distance to a funnel of a digraph, we applied these concepts to NP-hard problems. In particular, we searched for problems which are known to be NP-hard for DAGs. It is natural to ask whether we can solve such problems more efficiently (e.g. in polynomial time) if we restrict the input to be a funnel.

We focused on three problems. The first one, DAG Partitioning, was used to study the behavior of the news cycle [LBK09]. If we restrict the input to be a funnel, we obtain an $\mathcal{O}(|V|^3)$ time algorithm. By analyzing an NP-hardness reduction due to

Alamdari and Mehrabian [AM12], we observed that the problem is NP-hard even if the vertex-deletion distance to a funnel as well as the funnel depth and height are constants. It remains open whether the problem is in FPT with respect to the arc-deletion distance to a funnel.

We then considered the Max-Leaf Out-Branching problem, which finds applications in the construction of sensor networks [BD07]. Unfortunately, the problem remains NP-hard even if we restrict the input to be a funnel.

Finally, we studied the k-Linkage problem. The input for this problem is a digraph and a set of vertex pairs, and the goal is to find vertex-disjoint paths connecting both vertices of each pair. We provided an $\mathcal{O}(16^d \cdot |V| \cdot |A|)$ time algorithm for this problem, where d is the arc-deletion distance to a funnel of the input DAG. We also analyzed an already known data reduction rule, showing that it yields a problem kernel with $\mathcal{O}(d)$ vertices computable in $\mathcal{O}(k \cdot (|V| + |A|))$ time. Even though from an algorithmic point of view this result is not new, the concept of funnels provided the theoretical tools in order to analyze the effectiveness of the data reduction rules. For this reason it would be interesting to know how other distance to a funnel parameters relate to the k-Linkage problem. In particular, we pose as an open question whether k-Linkage is in FPT with respect to the vertex-deletion distance to a funnel.

From our theoretical analysis we conclude that funnels are a useful concept both in designing new algorithms as well as in analyzing the behavior of algorithms which do not directly use the concept of funnels. The graph class is also not restricted to a single domain, having applications to considerably different problems. Also for parameterized complexity we consider the distance to a funnel parameters to be an interesting direction for further research, specially because very few subclasses of DAGs exist, yet many problems remain NP-hard on DAGs [Gan+09].

From a practical perspective, we conclude after some experiments that the arc-deletion distance to a funnel of a DAG can often be quickly computed and also approximated to a value which is very close to the real distance. Hence, algorithms which require an arc-deletion set in order to work might be useful in practice. We also verified, for some selected real-world DAGs, that the arc-deletion distance to a funnel was small relative to the total amount of arcs. Even though it was fairly large in absolute value, this means that the parameter might be useful for data reduction rules, since they might drastically decrease the input size. We also release the code used to compute the arc-deletion distance to a funnel as free software [Mil17], allowing the research community not only to verify our results, but also investigate the size of the parameter in different scenarios.

Literature

[AB09] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. New York, NY, USA: Cambridge University Press, 2009 (cit. on p. 12).

[Alo+07] N. Alon, F. V. Fomin, G. Gutin, M. Krivelevich, and S. Saurabh. "Parameterized Algorithms for Directed Maximum Leaf Problems". In: *Automata, Languages and Programming: 34th International Colloquium, ICALP 2007, Wrocław, Poland, July 9-13, 2007. Proceedings*. Ed. by L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 352–362 (cit. on p. 9).

[AM12] S. Alamdari and A. Mehrabian. "On a DAG Partitioning Problem". In: *Proceedings of the 9th International Conference on Algorithms and Models for the Web Graph*. WAW'12. Halifax, NS, Canada: Springer-Verlag, 2012, pp. 17–28 (cit. on pp. 9, 48, 95).

[BB72] U. Bertele and F. Brioschi. *Nonserial dynamic programming*. Academic Press, 1972 (cit. on p. 8).

[BD07] P. Bonsma and F. Dorn. "An FPT Algorithm for Directed Spanning k-Leaf". In: *Preprint 046-2007, Combinatorial Optimization & Graph Algorithms Group*. 2007 (cit. on pp. 9, 53, 95).

[BD11] P. Bonsma and F. Dorn. "Tight Bounds and a Fast FPT Algorithm for Directed Max-Leaf Spanning Tree". In: *ACM Transactions on Algorithms* 7.4 (Sept. 2011), 44:1–44:19 (cit. on pp. 9, 53).

[Bev+17] R. van Bevern, R. Bredereck, M. Chopin, S. Hartung, F. Hüffner, A. Nichterlein, and O. Suchý. "Fixed-parameter algorithms for {DAG} Partitioning". In: *Discrete Applied Mathematics* 220 (2017), pp. 134–160 (cit. on pp. 9, 47–49).

[BJG08] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Science & Business Media, 2008 (cit. on pp. 8, 9, 55, 62, 87).

[Cyg+13] M. Cygan, D. Marx, M. Pilipczuk, and M. Pilipczuk. "The Planar Directed k-Vertex-Disjoint Paths Problem Is Fixed-Parameter Tractable". In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science* 00 (2013), pp. 197–206 (cit. on pp. 9, 55).

[Cyg+15] M. Cygan, F. V. Fomin, Łukasz Kowalik, D. L. D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer International Publishing, 2015 (cit. on p. 12).

[DF12] R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012 (cit. on pp. 9, 12, 55).

[FHW80] S. Fortune, J. Hopcroft, and J. Wyllie. "The directed subgraph homeomorphism problem". In: *Theoretical Computer Science* 10.2 (1980), pp. 111–121 (cit. on pp. 9, 55).

[Gan+09] R. Ganian, P. Hlinenỳ, J. Kneis, A. Langer, J. Obdrzálek, and P. Rossmanith. "On Digraph Width Measures in Parameterized Algorithmics." In: *IWPEC*. Vol. 5917. Springer. 2009, pp. 185–197 (cit. on pp. 9, 95).

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990 (cit. on p. 12).

[GMO76] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil. "On Two Problems in the Generation of Program Test Paths". In: *IEEE Transactions on Software Engineering* 2.3 (May 1976), pp. 227–231 (cit. on p. 65).

[KD79] M. S. Krishnamoorthy and N. Deo. "Complexity of the Minimum-Dummy-Activities Problem in a PERT network". In: *Networks* 9.3 (1979), pp. 189–194 (cit. on p. 65).

[Kun13] J. Kunegis. "KONECT – The Koblenz Network Collection". In: *Proc. Int. Conf. on World Wide Web Companion*. 2013, pp. 1343–1350. URL: http://userpages.uni-koblenz.de/~kunegis/paper/kunegis-koblenz-network-collection.pdf (cit. on p. 67).

[LBK09] J. Leskovec, L. Backstrom, and J. Kleinberg. "Meme-tracking and the Dynamics of the News Cycle". In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '09. Paris, France: ACM, 2009, pp. 497–506 (cit. on pp. 9, 47, 94).

[LY80] J. M. Lewis and M. Yannakakis. "The node-deletion problem for hereditary properties is NP-complete". In: *Journal of Computer and System Sciences* 20.2 (1980), pp. 219–230 (cit. on p. 22).

[Mar] S. Marlow. *Haskell 2010 Language Report*. URL: https://www.haskell.org/onlinereport/haskell2010/ (cit. on p. 69).

[Mil17] M. G. Millani. *Parfunn – Parameters for Funnels*. 2017. URL: https://gitlab.tubit.tu-berlin.de/mgmillani1/parfunn (cit. on p. 95).

[Nie06] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006 (cit. on p. 12).

[Pap03] C. H. Papadimitriou. "Computational Complexity". In: *Encyclopedia of Computer Science*. Chichester, UK: John Wiley and Sons Ltd., 2003, pp. 260–265 (cit. on p. 12).

[PY90] C. H. Papadimitriou and M. Yannakakis. “Towards an Architecture-independent Analysis of Parallel Algorithms”. In: *SIAM Journal on Computing* 19.2 (Apr. 1990), pp. 322–328 (cit. on pp. 13, 63).

[Sch94] A. Schrijver. “Finding k disjoint paths in a directed planar graph”. In: *SIAM Journal on Computing* (1994), pp. 780–788 (cit. on p. 9).

[Sli10] A. Slivkins. “Parameterized tractability of edge-disjoint paths on directed acyclic graphs”. In: *SIAM Journal on Discrete Mathematics* 24.1 (2010), pp. 146–157 (cit. on pp. 9, 55).

[SP78] Y Shiloach and Y Perl. “Finding two disjoint paths between two pairs of vertices in a graph”. In: *Journal of the ACM (JACM)* 25.1 (1978), pp. 1–9 (cit. on p. 9).