Diss. ETH No. 25608

Digraph Reachability Algorithms

A thesis submitted to attain the degree of
Doctor of Sciences of ETH Zürich
(Dr. sc. ETH Zürich)

presented by

Daniel Wolleb-Graf
MSc ETH Computer Science
born on September 9, 1990
citizen of Rebstein SG and Vilters-Wangs, Vilters SG

accepted on the recommendation of
Prof. Dr. Peter Widmayer, examiner
Prof. Dr. Juraj Hromkovic, co-examiner
Prof. Dr. Giuseppe F. Italiano, co-examiner

2018

For Helene

"It is a common experience that a problem difficult at night is resolved in the morning after the committee of sleep has worked on it." – John Steinbeck, "Sweet Thursday"

Abstract

Take a blank page, draw some small circles, and connect them with a few arrows, and voilà you created a directed graph, a *digraph* for short. Now let us ask an algorithmic question: If I point out two of those circles to you, one as the *start* and one as the *target*, can you find a way to go from the start to the target by only using the arrows in their one-way direction? While this *reachability* question is well-studied, we found new algorithms for the following slight variations of the reachability problem:

- What if we ask this questions for many starts and many targets while we also add and remove arrows in between answering these questions?
- What if we want to find two ways from the start to the target that use completely different arrows? Or can we even find three disjoint paths? And how fast can we compute the answer for all combinations of starts and targets?

For the dynamic reachability problem, we review the link-cut tree data structure for dynamic rooted forests before we extend it to a new class of dynamic graphs: partial-function graphs. We provide algorithms for arbitrarily interleaved queries and updates to the graph in time $\mathcal{O}(\log n)$.

In the generalized reachability problem, we ask about the existence of multiple arc-disjoint paths between the query vertices. The all-pairs k-reachability problem asks for the number of arc-disjoint paths between all vertex pairs, where one can answer "at least k", whenever there are k or more arc-disjoint paths.

2-Reachability answers basic resilience of paths, i.e., is there an arc that can not be avoided on the path from u to v. As our main result, we present an algorithm running in time $\mathcal{O}(n^\omega \log n)$ to compute all-pairs 2-reachability of any digraph, where ω is the matrix multiplication exponent. This result comes in three steps: first, we develop a path algebra with binary encodings for acyclic graphs, second, we use dominator trees to build auxiliary graphs that represent the answer for strongly connected graphs, and third, we carefully combine the two on arbitrary digraphs.

Finally, we look at the general k-reachability problem restricted to acyclic graphs. We develop a framework dealing with the structure of extremal cuts and encoding and handling those cuts efficiently to come up with two algorithms for DAGs: one running in time $\mathcal{O}(mn^{1+o(1)})$ for $k = o(\sqrt{\log n})$ and one in time $\mathcal{O}(n^{\omega+o(1)})$ for $k = o(\log \log n)$. One nice side effect of these algorithms is that they do not only report the size of the minimum cut but also provide a cut as a witness.

Zusammenfassung

Nehmen Sie ein leeres Blatt, zeichnen Sie einige kleine Kreise und verbinden Sie diese mit einigen Pfeilen, und schon haben Sie einen gerichteten Graphen kreiert. Nun stellen wir uns eine algorithmische Frage: Wenn ich auf zwei der Kreise zeige, einen als *Start* und einen als *Ziel*, können Sie einen Weg finden, um vom Start zum Ziel zu gelangen und dabei nur Pfeile in entlang ihrer Richtung benutzen? Diese *Erreichbarkeitsfrage* ist altbekannt, aber wir haben neue Algorithmen für die folgenden Variationen des Erreichbarkeitsproblems gefunden:

- Was passiert, wenn wir diese Frage für viele Start- und Zielkreise stellen und dazwischen auch noch Pfeile hinzufügen und entfernen?
- Was, wenn wir zwei Wege vom Start ins Ziel finden möchten, die komplett unterschiedliche Pfeile benutzen? Oder können wir gar drei disjunkte Pfade finden? Und wie schnell können wir die Antwort für alle Kombinationen von Start und Ziel berechnen?

Für das dynamische Erreichbarkeitsproblem fassen wir zuerst die Link-Cut Tree Datenstruktur für dynamische gewurzelte Wälder zusammen, bevor wir sie auf einen neue Klasse von dynamischen Graphen ausbauen: Graphen von partiellen Funktionen. Wir stellen einen Algorithmus vor, der Anfragen und Aktualisierungen in beliebiger Reihenfolge in der Zeit $\mathcal{O}(\log n)$ verarbeitet.

Im verallgemeinerten Erreichbarkeitsproblem fragen wir nach der Existenz von mehreren kantendisjunkten Pfaden zwischen den Anfrageknoten.

Das Alle-Paare-k-Erreichbarkeitsproblem fragt nach der Anzahl kantendisjunkter Wege zwischen allen Knotenpaaren, wobei die Antwort "mindestens k" gegeben werden kann, wann immer es k oder mehr kantendisjunkte Wege gibt.

2-Erreichbarkeit beantwortet die Frage nach der Ausfallsicherheit von Wegen, d.h. ob es eine Kante gibt, die auf einem Weg von u nach v nicht umgangen werden kann. Wir stellen als unser Hauptresultat einen Algorithmus vor, der in Zeit $\mathcal{O}(n^\omega \log n)$ die 2-Erreichbarkeit zwischen allen Knoten eines beliebigen gerichteten Graphen berechnet, wobei ω der Matrixmultiplikationsexponent ist. Das Resultat besteht aus drei Schritten: zuerst entwickeln wir eine Pfadalgebra mit binären Kodierungen für azyklische Graphen, dann benutzen wir Dominatorbäume, um Hilfsgraphen zu bauen, welche die Antwort für stark zusammenhängende Graphen repräsentieren, und am Ende kombinieren wir die beiden auf beliebigen gerichteten Graphen.

Schliesslich schauen wir uns das generelle k-Erreichbarkeitsproblem auf azyklischen Graphen an. Wir entwickeln einen Rahmen, um mit der Struktur von extremalen Schnitten umzugehen und um diese Schnitte effizient zu codieren und zu manipulieren, und bauen damit zwei Algorithmen für DAGs: einer läuft in Zeit $\mathcal{O}(mn^{1+o(1)})$ für $k = o(\sqrt{\log n})$ und einer in Zeit $\mathcal{O}(n^{\omega+o(1)})$ für $k = o(\log \log n)$. Ein hübscher Nebeneffekt dieser Algorithmen ist, dass sie nicht nur die Grösse des minimalen Schnitts bestimmen, sondern auch gleich einen Schnitt als Zeuge liefern.

Preface

Motivation

Why did I write this PhD thesis? Why should you read it? What new results can you expect to see on the following pages and how did they come to be? Which prior results did we build upon and who helped me in doing so? How did I get interested in *digraph algorithms* in the first place and why might they matter to society? I hope that reading this manuscript will answer all of these questions for you, most likely in reverse order, if at all.

Studying *theoretical computer science* meant that during the past three years, I mostly worked with pencil and paper. Just take a blank page, draw some small circles, and connect them with a few arrows, and voilà you created a directed graph, a *digraph* for short. Now let us ask an algorithmic question: If I point out two of those circles to you, one as the *start* and one as the *target*, can you find a way to go from the start to the target by only using the arrows in their one-way direction? While this *reachability* question is well-studied, we found new algorithms for the following slight variations of the reachability problem:

- What if we ask this questions for many starts and many targets while we also add and remove arrows in between answering these questions?
- What if we want to find two ways from the start to the target that use completely different arrows? Or can we even find three disjoint ways?

And how fast can we compute the answer for all combinations of start and target circles?

Acknowledgements

I am thankful to so many people that have been involved in the making of this thesis either directly or semi-unknowingly: Most of all, I thank my examiners Peter, Juraj and Pino for their support throughout the years and for carefully reviewing the final thesis.

I am deeply grateful to all my kind colleagues in our group and our institute: Andreas, Thomas, Przemek, Katerina, Barbara, Paolo, Stefano, Tobias, Akaki, Dan, Tomas, Tomas, Chih-Hung, Chen, Sandro, Jerri, Ahad, Alex, Luis, Patrick, and Michael. Also thanks to Nikos and Loukas for very enjoyable collaborations and visits.

I equally enjoyed having the opportunity to work with some great students: Karim, Stefan, and Kieran, and with the amazing staff of our department: Blanca, Denise, and Marianna.

There would be no fun in computer science, at least not for me, without the Swiss Olympiad in Informatics, especially without Sandro, Timon, Samuel, Johannes, Benjamin, Daniel, Stefanie, Joël, Luca, Luc, Pascal, and Timon.

I am happy that I got to spend some great months during my PhD in Sunnyvale. Thank you Christian, Daniel, Kaspar, Ugur and Siddharth.

Without some great role models as professors and lecturers, I would not have embarked on the PhD journey in the first place. Besides the ones above, Prof. Steger, Prof. Welzl, Prof. Sudakov, Prof. Holenstein, Prof. Meier and Prof. Wolf all made a lasting impression on me. Any Hochschule needs great high schools, like the one in Sargans. Many thanks to Josef, Rolf, Ivo, and Marianna. And even further back, thanks to Peter, Alexandra, Thomas, Helmut, and Ursula.

Last, but definitely not least, I deeply thank Cobra, Turbo, Anja, Jan, Annemarie, Heinz, Julia, Maja, Ursula, Andreas, Claudia, and Helene.

Contents

1

Introduction

1.1 Graphs: Notation and Naming

Directed Graphs. Let $D = (V, A)$ be a *directed graph* (*digraph* for short) on $n := |V|$ *vertices* and $m := |A|$ *arcs*. Each arc $a = (u, v) \in V \times V$ is an ordered pair of vertices consisting of a *tail* u and a *head* v. We say that a *leaves* u and *enters* v. We usually consider *simple* digraphs, i.e., digraphs where A contains each arc at most once and no *loops* (arcs with (v, v) for any $v \in V$). Digraphs with repeated arcs are called *multigraphs*.

If $(u, v) \in A$, v is called an *out-neighbor* of u, and u is called an *in-neighbor* of v. With $\deg^+(u) := |\{(u, v) \in A\}|$, we denote the out-degree of vertex u and with $\deg^-(v) := |\{(u, v) \in A\}|$, we denote the in-degree of vertex v.

A sequence of vertices $W = (v_1, v_2, v_3, \dots)$, where any two subsequent vertices v_i and v_{i+1} are arcs (v_i, v_{i+1}) of the digraph, we call a *walk*. The sequence is called a *path* if none of the vertices are repeated. The *length* of a directed path is given by its number of arcs. As a special case, there is a path of length 0 from each vertex to itself.

If only the first and last vertex are the same and the length is greater than 0, the sequence forms a *cycle*. A *directed acyclic graph* (in short *DAG*) is a digraph with no cycles. A DAG has a *topological ordering*, i.e.,

a linear ordering of its vertices such that for every arcs (u, v), u comes before v in the ordering (denoted by $u < v$).

We say that vertex v is *reachable* from vertex u if there is a directed path from u to v in D. We write $u \rightsquigarrow v$ to denote that there is a path from u to v, and $u \not\rightsquigarrow v$ if there is no path from u to v. The *transitive closure* R_D of D is the full n-by-n pairwise boolean reachability matrix of D.

A digraph D is *strongly connected* if there is a directed path from each vertex to every other vertex. The *strongly connected components* of a digraph are its maximal strongly connected subgraphs.

Given a subset of vertices $V' \subset V$, we denote by $D \setminus V'$ the digraph obtained after deleting all the vertices in V', together with their incident arcs. Given a subset of arcs $A' \subset A$, we denote by $D \setminus A'$ the digraph obtained after deleting all the arcs in A'.

Digraphs with $m \in \mathcal{O}(n)$ we call *sparse*, digraphs with $m \in \Omega(n^2)$ we call *dense*. A rooted tree is an arc-minimal digraph with the property that a designated root vertex r can be reached from every vertex. A rooted forest is a collection of disjoint rooted trees.

By $D[S]$, we denote the subgraph of D *induced* by the set of vertices S, formally $D[S] = (S, A \cap (S \times S))$.

Undirected Graphs. If the arcs do not have directions, we call them *edges*. For clarity, we will talk about *nodes* not vertices in undirected graphs. So let $G = (V, E)$ denote an *undirected graph* on $n := |V|$ nodes and $m := |E|$ edges. Now $E \subseteq \binom{V}{2}$ and two vertices u, v are *adjacent*, if $\{u, v\} \in E$. The definitions of walks, paths, and cycles do not change, they just ignore the direction.

Also, we talk about *connectivity* not *reachability* in undirected graphs. An undirected graph is *connected* if there is an undirected path from each node to every other node. The *connected components* of a graph are its maximal connected subgraphs. For a digraph D, its *weakly-connected components* are the connected components of the undirected counterpart of D, i.e., the undirected graph with one edge for every arc in D, ignoring the direction of the arcs in D.

1.2 Summary of Contributions

In Chapter 2, we study the reachability problem: for two vertices u and v is there a directed path from u to v in digraph D? The static all-pairs

version of the problem is well-known to be equivalent to Boolean matrix multiplication. We consider the dynamic version where arcs are inserted and deleted in between queries. We review the link-cut tree data structure for dynamic rooted forests in Section 2.2.1 before we extend it to a new class of dynamic graphs: partial-function graphs. In Section 2.2.2, we present a fully dynamic algorithm for graphs of partial functions.

In Chapters 3 and 4, we look at the generalized reachability problem where we ask about the existence of multiple arc-disjoint paths between the query vertices. The all-pairs k-reachability problem asks for the number of arc-disjoint paths between all vertex pairs, where one can answer "at least k", whenever there are k or more arc-disjoint paths.

Going beyond 1-reachability, the distinction between "a path" or "no path" (also known as transitive closure), we look at 2-reachability in Chapter 3. 2-Reachability answers basic resilience of paths, i.e., is there an arc that can not be avoided on the path from u to v. As our main result, we present an algorithm running in time $\mathcal{O}(n^\omega \log n)$ to compute all-pairs 2-reachability of any digraph, where ω is the matrix multiplication exponent. This result comes in three steps: first, we develop a path algebra with binary encodings for acyclic graphs, second, we use dominator trees to build auxiliary graphs that represent the answer for strongly connected graphs, and finally, we carefully combine the two on arbitrary digraphs.

We look at the general k-reachability problem restricted to acyclic graphs in Chapter 4. We develop a framework dealing with the structure of extremal cuts and encoding and handling those cuts efficiently to come up with two algorithms for DAGs: one running in time $\mathcal{O}(mn^{1+o(1)})$ for $k = o(\sqrt{\log n})$ and one in time $\mathcal{O}(n^{\omega+o(1)})$ for $k = o(\log \log n)$. One nice side effect of these algorithms is that they do not only report the size of the minimum cut but also provide a cut as a witness.

1.3 Attribution of Collaborators

The result for dynamic graphs of partial functions in Section 2.2.2 was obtained in collaboration with Timon Gehr. The results on 2-reachability and k-reachability in Chapters 3 and 4 were obtained and written up jointly with Loukas Georgiadis, Giuseppe F. Italiano, Nikos Parotsidis and Przemysław Uznański.

2

Reachability

2.1 Static Reachability

Problem Statement. The *all-pairs reachability problem* consists of preprocessing a digraph $D = (V, A)$ so that we can answer any query that asks whether a vertex v is reachable from a vertex u instantaneously. The result of this preprocessing is the $n \times n$-sized *transitive closure matrix* R_D such that

$$R_D(u, v) = \begin{cases} 1, u \text{ can reach } v \text{ in } D, \\ 0, \text{otherwise.} \end{cases} \tag{1}$$

We call this setting *static* since the digraph never changes. There are only two phases in solving this problem: first the preprocessing, second the answering of queries. Later we will see how things get more complicated when the digraph can change over time.

Graph Traversal. One solution to the problem is to perform a graph traversal algorithm, like depth-first or breadth-first search, once per vertex as a start. This way, we learn all the targets that a certain start vertex can reach in $\mathcal{O}(m)$ and finish computing R_D in $\mathcal{O}(mn)$ overall.

Matrix Multiplication. For dense digraphs, there is a faster way to compute the transitive closure R_D using fast matrix multiplication in time $\mathcal{O}(n^\omega)$, where ω is the matrix multiplication exponent, $2 \leq \omega < 3$. Currently, the best upper bound for ω is 2.373 by Le Gall [30].

We now summarize a result by Munro [35] and Fischer and Meyer [13].

▶ **Theorem 2.1 ([35], [13]).** *The two problems of computing the transitive closure of a n-vertex digraph and multiplying two $n \times n$ boolean matrices are of equivalent asymptotic complexity $\mathcal{O}(n^\omega)$.*

Proof. To compute the transitive closure using matrix multiplication we proceed in two steps: First, note that strongly-connected components in the digraph D do not make the problem any harder. I.e., if we compute the reachability between the strongly-connected components, we know all we need to answer reachability queries for D. Therefore, we contract all cycles in the digraph D by contracting all the strongly connected components. We can do this in linear time, e.g., using Tarjan's algorithm [49]. What remains is an acyclic digraph which we can now attack using divide-and-conquer. We order the vertices in a topological order (also in linear time) and let A denote the upper-triangular adjacency matrix of the contracted graph with

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ 0 & A_{2,2} \end{bmatrix} \text{ and } A^* = \begin{bmatrix} A^*_{1,1} & A^*_{1,1} \cdot A_{1,2} \cdot A^*_{2,2} \\ 0 & A^*_{2,2} \end{bmatrix}, \tag{2}$$

where the asterisk in A^* denotes the transitive closure of the boolean matrix A. We exploit that any path from a vertex in the first half to a vertex in the second half uses exactly one arc in $A_{1,2}$, which allows us to compute A^* using only two recursive calls for $A^*_{1,1}$ and $A^*_{2,2}$ and two matrix multiplications for the upper right part. This gives raise to the recurrence relation (assuming $n = 2^i$)

$$T(2^i) \leq 2 \cdot T(2^{i-1}) + 2 \cdot M(2^{i-1}) + O(i), \tag{3}$$

where $T(n)$ denotes the runtime function for transitive closure and $M(n)$ the runtime of boolean matrix multiplication. As $M(n) = \mathcal{O}(n^\omega)$ and the recursive summation converges, we get $T(n) \in \mathcal{O}(n^\omega)$.

For the reverse direction, consider two $n \times n$-matrices A and B for which we would like to compute their product $A \cdot B$. We can compute this product through transitive closure by building a tri-partite graph with A

as the adjacency between the first and second part and B as the adjacency between the second and third part. We observe that

$$\begin{bmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{bmatrix}^* = \begin{bmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix} \tag{4}$$

so we can just read off the product AB in the upper right entry of the transitive closure. ◀

This result settles the asymptotic complexity of the all-pairs reachability problem for static, general digraphs. Reachability remains an interesting question if we make the graph dynamic (in the next section) or slightly generalize the question (in the next chapter).

2.2 Dynamic Reachability

Roadmap. In this section, we build reachability algorithms for digraphs that change over time. We first look at rooted forests. Sleator and Tarjan [46] came up with a beautiful data structure called *link-cut trees* for this setting and in Section 2.2.1 we will give a detailed introduction to its working.

We then look at a slightly more general class of dynamic digraphs, namely partial function graphs, in Section 2.2.2. We motivate this class by a practical example from compiler design. We then augment the link-cut trees to solve this new problem and present some additional operations which we would like to support for our application. A reader who is only interested in our digraph result may safely skip the motivational paragraphs.

Finally in Section 2.2.3, we give an overview of related work on general dynamic digraphs.

2.2.1 Link-Cut Trees

Our object of study for this section are forests of rooted trees, so a collection of undirected, cycle-free graphs each with a designated root node (giving each edge an implicit orientation towards the root). We want to ask connectivity questions of the form *Are u and v in the same tree?* or *Is v on u's path towards the root?* and support edge insertion and deletions

with the goal of achieving small (amortized) runtimes for all these (and more) operations. Formally, this is the interface that we want to support:

LINK(u, v) inserts an edge $\{u, v\}$ into the forest combining two separate trees. For this, u and v have to be in different trees and u has to be the root of its tree.

CUT(v) deletes the edge from v to its parent towards the root, splitting its tree into two, with v at the root of one of them. For this, v has to be any vertex but the root of its tree.

ROOT(v) returns the root vertex of the tree containing v.

LCA(u, v) returns the lowest common ancestor of nodes u and v, the vertex furthest from the root that lies on both the path from u to the root and the path from v to the root. For this, u and v have to be in the same tree.

We now loosely follow the excellent lecture by Demaine [11] to give a detailed introduction.

Semi-dynamic settings. As a quick warm-up, let us consider the two semi-dynamic settings, where only one type of update is admitted. In the *incremental* world, where edges are only inserted but never deleted, we can achieve $\mathcal{O}(\alpha(n))$ runtimes using a union-find data structure to keep track of the tree-merging process, cf. Tarjan [51]. If we consider the *decremental* setting, where we start with a single tree and chop it apart piece by piece until we end up with individual nodes, we can apply a result by Even and Shiloach [45]: We keep a label for each vertex to mark membership of each tree with a unique label. Upon deletion of an edge, we scan through both resulting trees in parallel using two synchronized BFS traversals. The scan through the smaller resulting tree will finish first. At that point we relabel all the nodes in this smaller tree and stop the traversal of the other one. As we only relabel the smaller tree, each node gets relabelled at most $\log n$ times throughout the entire time. Therefore, in an amortized sense, we get decremental updates in $\mathcal{O}(\log n)$ and constant time queries. With bit-tricks, Alstrup et al. [4] can even shave the log and get $\mathcal{O}(1)$ edge removals.

Core idea. Now talking about Sleator-Tarjan's [46] link-cut trees for fully dynamic updates, here is the key idea: To handle unbalanced trees,

we decompose each tree into a collection of disjoint paths and then store each path in a separate, balanced tree. This leaves two things to be defined: how do we decompose the trees and how do we store these pieces? We use the *preferred path decomposition*: each node v has a *preferred child* w, if the last access to v's subtree ended in w's subtree. If the last access ended at v, then v has no preferred child. So after accessing v, the tree containing v decomposes into the root-v-path being preferred plus some older preferred path pieces too. We represent each preferred path by a splay tree, keyed by depth, and call this an *auxiliary tree*. The root of each auxiliary tree keeps a pointer to the parent of the path. So each *represented tree* is stored in the form of a tree of auxiliary trees.

Core operation. The surprising feature of this representation as tree of auxiliary trees is that all the operations can now be quite easily built on top of the following operation that simply accesses a node v. Accessing a node v means cutting the top parts of potentially many different preferred paths and recombining them into a single new one. We do this in a bottom-up manner starting at node v, see Algorithm 1 for pseudocode. To cut off a previous preferred child of v, we splay v within its auxiliary tree so that we can clip the right subtree containing everything below v. We then walk up the tree of aux trees and in each step take the top-most part above v's path parent and cut and merge it with v's auxiliary tree. Figure 1 illustrates these steps.

Algorithm 1 Access(v) [11] [46] [52]

```
1: splay v (within its auxiliary tree)
2: v.right.pathparent = v                    ▷ cut the preferred path below v
3: v.right.parent = none
4: repeat  ▷ each iteration merges one more piece onto the root-v-path
5:     w = v.pathparent
6:     splay w
7:     switch w's preferred child to v
8:     splay v
9: until v.pathparent = none
```

Ensure: v is the root of the tree of auxiliary trees

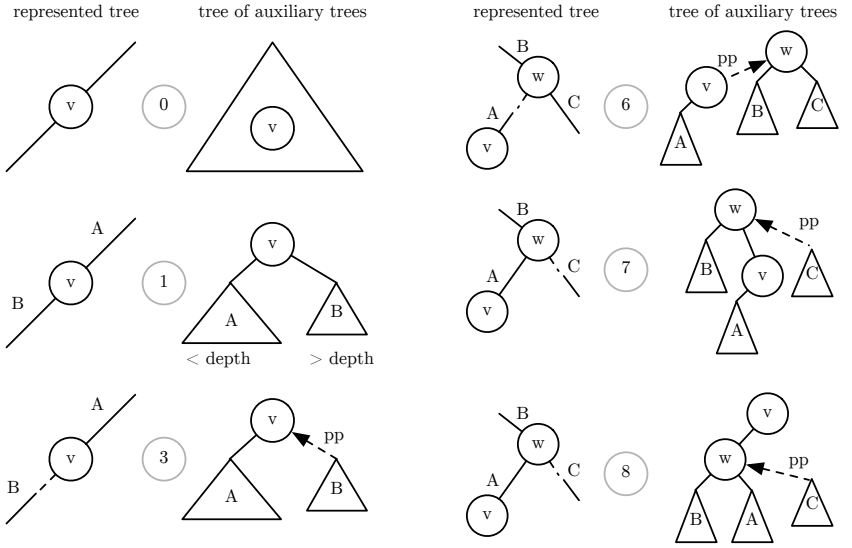


Figure 1 Steps of the access function corresponding to Algorithm 1. Preferred paths are drawn as solid lines, path parents as dashed lines.

Implementing the operations. Using ACCESS(v), we can now build the other operations quite easily. For ROOT(v), we simply access v such that the root becomes the smallest key in v's auxiliary tree. As shown in Algorithm 2 and Figure 2, we also call ACCESS(r) to avoid paying a lot when we would repeatedly search for this minimum key.

Algorithm 2 ROOT(v) [11]

```
1: ACCESS(v)                    ▷ make root-to-v path preferred
2: walk left to find r          ▷ find the minimum in v's auxiliary tree
3: ACCESS(r)                    ▷ so that it is fast next time
4: return r
```

Linking u below v is straightforward after accessing them: u is a singleton auxiliary tree and v is the root of its splay tree, so we make v the right child of u and thus make $\{u, v\}$ a preferred edge. See Algorithm 3 and Figure 3.

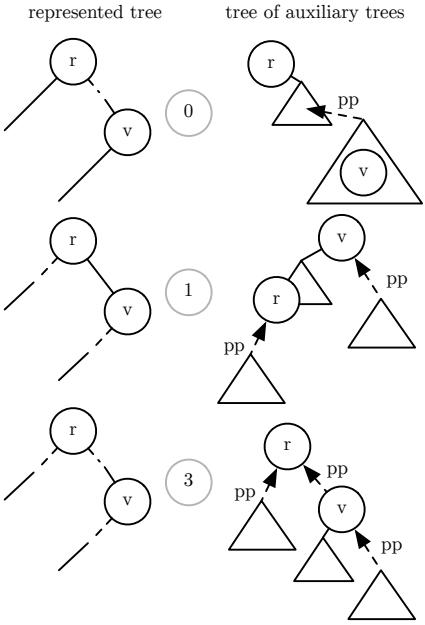


Figure 2 Steps of the root function corresponding to Algorithm 2.

Algorithm 3 LINK(u, v) [11]

1: ACCESS(u)
2: ACCESS(v)
3: u.left = v
4: v.parent = u

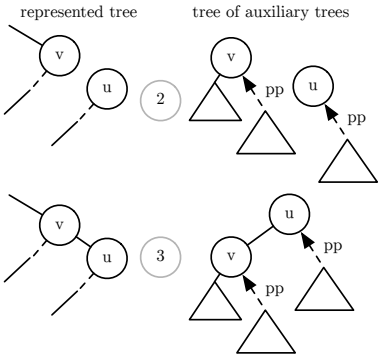


Figure 3 Steps of the LINK(u, v) function.

To CUT(v), we simply chop off v from its preferred path after accessing it. This makes v its own auxiliary tree disconnecting it from everything above it, see Algorithm 4 and Figure 4.

Algorithm 4 CUT(v) [11]

1: ACCESS(v)
2: v.left.parent = none
3: v.left = none

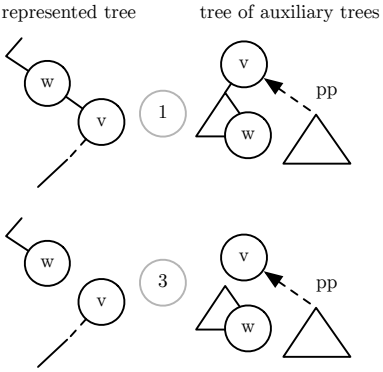


Figure 4 Steps of the CUT(v) function.

Finally, to compute the lowest common ancestor of u and v, we access them both and carefully watch where the common prefix of their preferred paths ends: After accessing u and while accessing v, the last path parent w that we modify is their lowest common ancestor. See Algorithm 5 and Figure 5.

Algorithm 5 LCA(u, v) [46]

Require: Root(u) = Root(v)
1: Access(u)
2: Access(v)
3: **return** last w in the Access(v)-loop

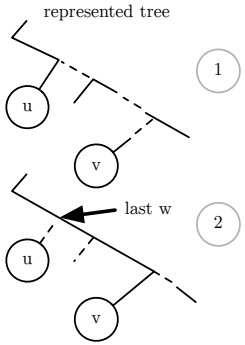


Figure 5 Steps of the LCA(u, v) function.

With these queries, link-cut trees can answer reachability queries on rooted trees: There is a directed path from u to v if and only if Root(u) = Root(v) and LCA$(u, v) = v$.

Without going into the potential argument for the amortized analysis, all of these operations are supported in amortized time $\mathcal{O}(\log n)$. Sleator and Tarjan's original paper [46] guarantees these time bounds in the worst-case per operation using biased search trees [6]. The version we presented here [52], building on top of self-adjusting binary trees [47] (splay trees), gives the same bound only in an amortized sense but is significantly simpler.

Hence link-cut trees can be used to maintain dynamic connectivity of undirected forests in $\mathcal{O}(\log n)$. By the $\Omega(\log n)$ lower bound of Demaine and Pătraşcu [39], which improved an earlier $\Omega(\log n / \log \log n)$ bound by Fredman and Saks [17], we know that this is optimal in the cell probe model.

2.2.2 Digraphs of Partial Functions

Problem Description. We say that a digraph $D = (V, A)$ is a *partial function graph*, if we have $\deg^+(v) \leq 1$ for all $v \in V$. The name of this class of graphs stems from the fact that the set A can be interpreted as representing a partial function $f : V \nrightarrow V$. Note that this implies $m \leq n$ for any partial function graph. Therefore, all such graphs are sparse.

In this section, we address the *fully dynamic reachability* problem on partial function graphs. Our goal is to maintain a representation of D

such that the following three operations can be performed efficiently in any order, starting initially with $A = \emptyset$:

INSERT(u, v) adds an arc (u, v) to D, assuming that $\deg^+(u) = 0$ before the insertion.

DELETE(u, v) removes the arc (u, v) from D.

QUERY(u, v) returns whether there is a directed path from u to v in D or not.

We consider the *fully* dynamic setting, so updates and queries can be interleaved in any order.

Our contribution. In this section, we show that graphs of partial functions allow updates and queries in $\mathcal{O}(\log n)$ worst-case time. To the best of our knowledge, no bound better than $\mathcal{O}(n)$ time per query was previously known for this setting, which is achieved by a simple BFS. As we will see, building on top of link-cut trees, we can achieve this result with only relatively little bookkeeping of the cyclic structure on top of it.

Motivation. Our interest in the dynamic reachability problem for partial function graphs arose from a performance challenge when implementing an efficient compiler. Our solution enables dynamic scheduling of a set of interdependent tasks for which coping strategies for cyclic dependencies exist. The execution of a task blocks as soon as it discovers that in order to continue, it depends on the result of another task. Therefore, at any point any task is waiting for at most one other task to complete.

Semantic Analysis. In a compiler for a modern programming language like *D*, a key component is the so-called *semantic analysis*. During this analysis, the compiler enforces, among other things, that all the variables used throughout the code are properly defined. This is a non-trivial task. Just reading through the code is not sufficient to figure out which variables are even defined at all. This is especially true for programming languages that support, at compile time, introspection, evaluation of code, and code generation (the `mixin` concept in *D*). See [15] for details. As the name suggests, compile time generation of code can cause dependencies within the code to only appear while the code is being analyzed.

Example: Here is a short, yet non-trivial code snippet as an example.

```
1 enum x = 3;
2 struct S {
3       enum y = x;
4       mixin(z);
5       mixin(foo(y));
6 }
```

What does `x` on line 3 refer to? Is it the `enum` from line 1 or is it something within the scope of `S` that appears only after evaluating `mixin(z)` on line 4? But what is `z`? It may be some global string but it may also be something that is defined locally, maybe by the second `mixin` on line 5. As declarations of variables within the inner nearest scope have precedence, we can not continue deciding about line 3 before evaluating lines 4 and 5.

Throughout the analysis, the compiler thus has to maintain a set of *nodes* of the abstract syntax tree, short AST. These nodes represent variable names like `x`, `y`, and `z`, but also other expressions like `foo(y)` or declarations like `enum x = 3;`. The compiler will try to look up and evaluate these nodes and in this process detect some *dependencies* between them, e.g., of the form "I need to know `x` in order to evaluate `y`". This is a dynamic process, so the set of AST nodes and dependencies will vary over time, during the execution of the compiler.

In this process, cyclic dependencies might occur. In the example above, we need `x` to define `y` on line 3, but as `x` could be defined in the output of `mixin(foo(y))` on line 5, we also need to know `y` to look up `x`. In such a case, the compiler has to address this cyclic dependency and resolve it in a consistent fashion. For instance, by assuming that `z` will not be defined within `mixin(foo(y))`, the compilation process can continue. (If it later turns out that in fact, this assumption must be violated, the program is rejected as invalid. Designing a sound and efficient yet intuitive set of rules to determine whether a program is well-defined is an interesting problem that we do not further elaborate on here.) Hence, an important subtask of the compiler is to maintain and resolve such dependencies and their potentially cyclic structure efficiently. In particular, it may sometimes break cycles.

Modeling as a Dynamic Graph Problem. We now model this task of managing evaluation dependencies as a dynamic graph problem on a directed graph $D = (V, A)$. Each *vertex* $v \in V$ represents an *AST node* v

and each *arc* $(u, v) \in A$ represents a dependency of the form "to evaluate u, it is necesssary to first evaluate v".

The abstract approach of the compiler is the following: Take any node that is not waiting for some other node to be resolved and process it. When processing a node, the following things might happen:

- Some new nodes appear (for example, we discover new variable names that need definition).
- A node without an outgoing arc gets a new out-arc (for example, we discovered an AST node that could potentially define some variable).
- A node without an outgoing arc is deleted (we finished some evaluation), and also all its incoming arcs are deleted (all dependencies on this AST node are now resolved).

If at some point no node is left that does not depend on something else, the compiler is momentarily stuck. At this point, all cyclic dependencies[1] have to get resolved[2], which will free some nodes of their dependencies and will allow further progress.

Rephrasing this exclusively as graph operations, these are the operations that our data structure should be able to perform:

- Insert a new vertex v.
- Insert arc (u, v), where $\deg^+(u) = 0$.
- Delete vertex v, where $\deg^+(v) = 0$, and all arcs of the form (u, v) for any u.
- Report a vertex v such that $\deg^+(v) = 0$ (or report that there is none).
- Delete all arcs in A that are part of any cycle and report all the incident vertices.

Clearly, the graph maintained here is a partial function graph. We will now go back to describing our reachability data structure for partial function graphs. As we will see below, all the extra operations we would like to have for our application can easily be supported on top of our data structure.

[1] There always is at least one cyclic dependency at this point as $n = m$. One slow way of finding a cycle at this point would be to start at any vertex and then just repeatedly follow the out-arc of the current vertex until we end up in a cycle.

[2] We assume that we always resolve *all* cyclic dependencies, even if there are multiple ones at the same time. As we will see, our data structure could also just as well only delete one single cycle at a time.

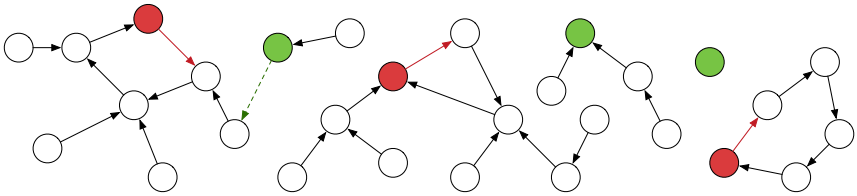


Figure 6 A graph of a partial function just before the dashed arc is inserted. The vertices in V_0 are colored in green, the arcs in A_C are colored in red. All the roots in the rooted forest D', the graph obtained by deleting the red arcs, are shown in red or green color.

Our Algorithm.

Data Structures. All our data structures do not rely on random access and thus work in the pointer machine model [53]. Throughout the process, we keep track of V and A explicitly. To represent A, we store both the in- and out-adjacencies of each vertex. While each vertex has at most one out-neighbor at any point in time, some vertices might have large in-degree. We do not assume that the vertices are represented as integers, just that they are comparable under a total order. This is sufficient for us to store V and the set of in-adjacency lists in balanced binary search trees (BBST), e.g., using [21], and then easily support insertion and deletion of arcs and (isolated) vertices in $\mathcal{O}(\log n)$.

In addition to maintaining V and A, we separately track the set of vertices with out-degree zero, which we denote by $V_0 := \{v \in V \mid \deg^+(v) = 0\}$ and also store it as a BBST.

The key component is the following set of arcs A_C. For each cycle C in D, we keep track of the last arc that was added to D, i.e., the one that closed C. We collectively denote this set as $A_C \subset A$ and we also maintain A_C as a BBST, indexed by the tails of its arcs.

We now define the subgraph $D' = (V, A \setminus A_C)$ (see Figure 6 for an example). Note that D' is acyclic but has the same weakly connected components as D, because D' is missing exactly one arc from each cycle in D. More precisely, D' is a forest of rooted trees. We can thus maintain D' as a link-cut tree. The link-cut tree representation of D' allows us to tell quickly, namely in $\mathcal{O}(\log n)$ time, whether a new arc (u, v) forms a cycle or not, namely whenever u and v are already in the same tree.

Updates. We specify our dynamic updates in Algorithms 6 and 7.

For the insertion of an arc (u, v), we check if the arc closes a cycle, i.e.,

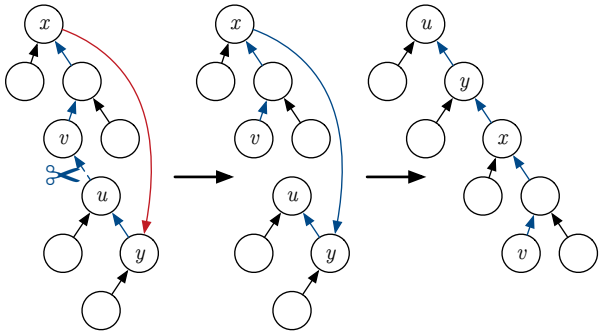


Figure 7 Deletion of an arc (u, v) which was part of a cycle in G.

if in D' we have $\textsc{Root}(v) = u$. If this is the case, (u, v) is added to A_C and hence not to D'. Otherwise, (u, v) connects two different trees and we $\textsc{Link}(u, v)$ in D'. Finally, we add (u, v) to A and remove u from V_0.

To support the deletion of an arc (u, v), we first check whether its component is a rooted tree or not. If it is, we just $\textsc{Cut}(u)$ to split up the tree. Otherwise, there is a cycle C in the component. If (u, v) is the cycle arc in A_C, we delete it and D' stays unchanged. If not, it remains to distinguish whether (u, v) is part of the cycle C or not. Let $(x, y) \in A_C$ be the cycle arc of the component. All vertices on C lie on the y-x-path in G'. So (u, v) lies on C whenever $\mathrm{LCA}(u, y) = u$. In that case, the deletion of (u, v) keeps the component weakly connected and we replace (x, y) by (u, v) in D'. This corresponds to moving the subtree of D' below u to the root and reattaching the remaining subtree below x (see Figure 7 for an illustration)[3]. In any case, u will be a root after the deletion, so we insert it into V_0.

[3] Note that this rerooting is different from the $\textsc{Evert}(u)$ operation supported by the link-cut trees.

Algorithm 6 Insert(u, v)

Require: V_0.Find(u) = true
1: **if** D'.Root$(v) = u$ **then**
2: A_C.Insert$((u, v))$
3: **else**
4: D'.Link(u, v)
5: **end if**
6: A.Insert$((u, v))$
7: V_0.Delete(u)

Algorithm 7 Delete(u, v)

Require: A.Find$((u, v))$ = true
1: $x \leftarrow D'$.Root(u)
2: **if** A_C.Find(x) = false **then**
3: D'.Cut(u)
4: **else**
5: $(x, y) \leftarrow A_C$.Find(x)
6: **if** $(x, y) = (u, v)$ **then**
7: A_C.Delete$((x, y))$
8: **else**
9: **if** D'.LCA$(u, y) \neq u$ **then**
10: D'.Cut(u)
11: **else**
12: A_C.Delete$((x, y))$
13: D'.Cut(u)
14: D'.Link(x, y)
15: **end if**
16: **end if**
17: **end if**
18: A.Delete$((u, v))$
19: V_0.Insert(u)

Queries. To decide for two given vertices u and v whether there exists a directed path from u to v, being able to query D' does go a long way, as shown in Algorithm 8. If u and v are in separate trees of D', there will not be a path in D. If v is the lowest common ancestor of u and v in D', v lies on the path towards the root from u and hence there is path in D' and also in D. It remains to take care of the potential cycles in D. Let x be the root of u's tree in D'. If there is an arc (x, y) in A_C, there might be a u-v-path through (x, y). Let C be the unique cycle containing x and y in D. All the vertices of C lie on the y-x-path in D'. Hence if $\mathrm{LCA}(v, y) = v$ in D', v is part of the cycle C that can be reached by all the vertices in this component, in particular by u. If $\mathrm{LCA}(v, y) \neq v$, there is no u-v-path in D as the arc (x, y) can only help reaching vertices in C.

Algorithm 8 QUERY(u, v)

```
1: if D'.ROOT(u) ≠ D'.ROOT(v) then
2:     return false
3: end if
4: if D'.LCA(u, v) = v then
5:     return true
6: end if
7: x ← D'.ROOT(u)
8: if A_C.FIND(x) = false then
9:     return false
10: end if
11: (x, y) ← A_C.FIND(x)
12: if D'.LCA(v, y) = v then
13:     return true
14: else
15:     return false
16: end if
```

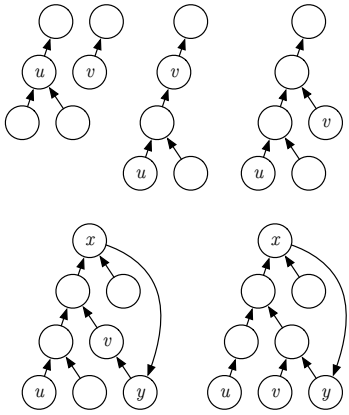


Figure 8 The cases considered for reachability queries in Algorithm 8.

▶ **Theorem 2.2.** *Our algorithm for dynamic graphs of partial functions handles each arc insertion and arc deletion in $\mathcal{O}(\log n)$ worst-case time and answers reachability queries in $\mathcal{O}(\log n)$ worst-case time.*

Proof. As each operation uses only a constant number of operations on the link-cut tree D' and the balanced binary search trees representing V, A, V_0 and A_C, the runtime bounds immediately follow. ◀

Extensions.

Cycle Queries. From our application in semantic analysis described earlier, we recall the four operations that we would like to support in addition to dynamic reachability: insert a new vertex, delete a vertex without out-arcs, report any vertex without out-arcs, and report and delete all cycles.

▶ **Corollary 2.3.** *We can support all these operations in $\mathcal{O}(\log n)$ time, where the bounds for vertex deletions and cycle deletions hold in an amortized sense.*

Proof. We provide implementations of the additional operations in Algorithms 9, 10, 11, and 12. For vertices of large in-degree or for multiple/long

cycles, the deletion of a single vertex or of all the cycles might have worst-case cost $\mathcal{O}(n \log n)$. However, these costs can be amortized by charging the insertions of the deleted arcs.

To retrieve all the arcs of a cycle in Algorithm 12, we follow the v-u-path in D' for any cycle arc $(u, v) \in A_C$, collect the vertices in a set C and cut the arcs along the way. As each vertex on the cycle will become the root of a newly formed rooted tree, we also add them all to V_0. ◀

Algorithm 9 Insert(v)

```
1: V.Insert(v)
2: V_0.Insert(v)
```

Algorithm 10 Delete(v)

```
Require: V_0.Find(v) = true
1: for (u, v) ∈ A do
2:     D'.Cut(u)
3:     A.Delete((u, v))
4:     V_0.Insert(u)
5: end for
6: V_0.Delete(v)
7: V.Delete(v)
```

Algorithm 11 Report()

```
1: if |V_0| ≠ 0 then
2:     return any v ∈ V_0
3: else
4:     return "V_0 is empty"
5: end if
```

Algorithm 12 DeleteCycles()

```
1: C ← ∅
2: for (u, v) ∈ A_C do
3:     x ← v
4:     repeat
5:         C ← C ∪ {x}
6:         y ← D'.Parent(x)
7:         D'.Cut(x)
8:         A.Delete((x, y))
9:         V_0.Insert(x)
10:        x ← y
11:    until x = u
12:    C ← C ∪ {u}
13:    A_C.Delete((u, v))
14:    A.Delete((u, v))
15:    V_0.Insert(u)
16: end for
17: return C
```

Heavy Arcs. In the dependency graph during semantic analysis, some arcs represent *strict* dependencies, that cannot be resolved in an alternative fashion. Even when deleted in a DeleteCycles operation, these arcs (x, y) will immediately be inserted again once the compiler reevaluates their tail node x, meaning that we really have to evaluate y before being able to proceed with x.

Currently, our data structure is agnostic to this distinction between

those strict and other weak dependencies. This can cause significant inefficiencies if a long cycle mainly consists of strict dependencies and we repeatedly delete and reinsert all of them. So can we handle the setting with such arcs more efficiently? I.e., can we not delete those *heavy* arcs of the cycles but only the other *light* ones? For simplicity, we assume that there is always at least one light arc in every cycle so that eventual progress is ensured.

▶ **Corollary 2.4.** *We can support the deletion of all light arcs on any cycle in $\mathcal{O}(\log n)$ amortized time.*

Proof. We can incorporate the distinction between heavy and light arcs using edge weights in the link-cut tree D'. As described in [46], link-cut trees can be augmented so that every $\textsc{Link}(u, v, w)$ operation takes a weight w for the edge $\{u, v\}$ as a third argument and it supports an additional $\mathcal{O}(\log n)$ time query $\textsc{MinCost}(v)$ that returns a vertex u such that the edge $(u, \textsc{Parent}(u))$ is of minimum weight among all edges on the v-root-path. We let $\textsc{Cost}((u, v))$ denote the weight of any arc (u, v) in D.

Our adapted, weighted implementation of DeleteCycles is described in Algorithm 13. When processing a cycle arc (u, v), we query for the lightest edge on the v-u-path. If the returned arc (x, y) is light, we cut (x, y) and recurse on the two subpaths v-x and y-u. This recursion stops whenever the subpath only consists of heavy arcs. Hence, we perform at most thrice as many MinCost queries as we delete light arcs. If the cycle arc (u, v) is heavy, we link it in D', being sure not to create a cycle as at least one other edge on the v-u path was cut. In the end, the set C contains all the vertices whose outgoing arc was deleted. ◀

Note that none of our algorithms makes use of the $\textsc{Evert}(v)$ operation of the link-cut tree, which significantly simplifies its implementation.

Algorithm 13 DeleteCycles()

```
1: C ← ∅
2: for (u, v) ∈ A_C do
3:     C ← C ∪ DeleteLightArcs(v)
4:     if Cost((u, v)) = light then
5:         A_C.Delete((u, v))
6:         V_0.Insert(u)
7:         C ← C ∪ {u}
8:     else                                   ▷ (u, v) is heavy
9:         D'.Link(u, v, heavy)
10:    end if
11: end for
12: return C
```

Algorithm 14 DeleteLightArcs(v)

```
1: C ← ∅
2: x ← D'.MinCost(v)
3: y ← D'.Parent(x)
4: if Cost((x,y)) = light then
5:     D'.Cut(x)
6:     V_0.Insert(x)
7:     C ← C ∪ {x}
8:     C ← C ∪ DeleteLightArcs(v)
9:     C ← C ∪ DeleteLightArcs(y)
10: end if
11: return C
```

2.2.3 Related Work

Dynamic reachability is a well studied problem that has many practical applications in the study of large time-varying graphs like road networks or social networks.

For undirected dynamic connectivity, all operations can be done in amortized polylogarithmic time as shown by Holm, de Lichtenberg and Thorup [23]. Their result builds on Euler tour trees by Henzinger and King [22], a data structure to maintain connectivity of forests in time $\mathcal{O}(\log n)$ per operation.

The best known algorithms for the more general problem on arbitrary directed graphs explore a trade-off between update and query time. For instance, Roditty and Zwick [41, 42] present an algorithm with amortized update time $\mathcal{O}(m+n \log n)$ and worst-case query time $\mathcal{O}(n)$. Two different trade-off points, namely $\mathcal{O}(n^{1.575})$ time per update and $\mathcal{O}(n^{0.575})$ time per query or $\mathcal{O}(n^{1.495})$ time for both, were given by Sankowski [43]. We refer to the survey by Demetrescu, Eppstein, Galil and Italiano [12] for more. Currently, all dynamic reachability algorithms for general graphs are fairly complicated and require at least linear update or query time. Conditional lower bounds by Pătraşcu [38] and recently by Abboud and Vassilevska Williams [1] indicate that polylogarithmic running times might be impossible.

For sparse digraphs, such as partial function graphs, recomputing reachability for each query from scratch, for example using breadth-first search (BFS), in $\mathcal{O}(n)$ time is astonishingly still the best known (in terms of maximum cost per operation).

However, this *linear time barrier* for directed reachability was broken for some special classes of directed graphs. Subramanian [48] showed that $\mathcal{O}(n^{2/3} \log n)$ time per operation can be achieved for plane graphs. Husfeldt [24] gave an $\mathcal{O}(\log n)$ time algorithm for plane acyclic graphs with a single source and sink. Italiano, Marchetti Spaccamela and Nanni [27] gave $\mathcal{O}(\log n)$ time algorithms for various series parallel digraphs. In particular, they studied looped two terminal series parallel graphs, a family of graphs closed under serial, parallel, and parallel reverse composition. Note that these looped series parallel graphs do not include partial function graphs since they do allow for cycles but not for arbitrary rooted trees attached to them. To the best of our knowledge, this last result is the only previously known dynamic reachability algorithm with polylogarithmic time bounds for any cyclic graph class.

3

2-Reachability

In this chapter, we study a natural generalization of the all-pairs reachability problem, that we refer to as *all-pairs 2-reachability*, where we wish to preprocess the digraph D so that we can answer the following type of queries fast: For a given vertex pair $u, v \in V$ are there two directed arc-disjoint (resp. internally vertex-disjoint) paths from u to v? Equivalently, by Menger's theorem [33], we ask if there is an arc $a \in A$ (resp., a vertex $w \in V$) such that there is no path from u to v in $D \setminus a$ (resp., $D \setminus w$). We call such an arc (resp., vertex) *separating* for the pair u, v.

One solution to the all-pairs 2-reachability problem is to compute all the dominator trees of D, with each vertex as source. The dominator tree of D with start vertex s is a tree rooted at s, such that a vertex v is an ancestor of a vertex w if and only if all paths from s to w include v [31]. All the separating arcs and vertices for a pair s, v, appear on the path from s to v in the dominator tree rooted at s, in the same order as they appear in any path from s to v in G. Given all the dominator trees, we can process them to compute the 2-reachability information for all pairs of vertices (see Section 3.8). Since a dominator tree can be computed in $\mathcal{O}(m)$ time [3, 8], the overall running time of this algorithm is $\mathcal{O}(mn)$.

3.1 Overview

In this chapter, we show how to beat the $\mathcal{O}(nm)$ bound for dense graphs. Specifically, we present an algorithm that computes 2-reachability information for all pairs of vertices in $\mathcal{O}(n^\omega)$ time in a strongly connected digraph, and in $\mathcal{O}(n^\omega \log n)$ time in a general digraph. Hence, we show that the running time of all-pairs 2-reachability is only within a log factor of transitive closure. This result is tight up to a log factor, since it can be shown that all-pairs 2-reachability is at least as hard as computing the transitive closure (see Section 3.6), which is asymptotically equivalent to Boolean matrix multiplication (see Theorem 2.1).

Moreover, our algorithm produces a witness (separating arc or separating vertex) whenever 2-reachability does not hold. By processing these witnesses, we can find all the dominator trees of D in $\mathcal{O}(n^2)$ additional time. Thus, we also show how to compute all the dominator trees of a digraph in $\mathcal{O}(n^\omega \log n)$ time (in $\mathcal{O}(n^\omega)$ time if the graph is strongly connected), which improves the previously known $\mathcal{O}(nm)$ bound for dense graphs. This in turn enables us to answer various connectivity queries in $\mathcal{O}(1)$ time. For instance, we can test in $\mathcal{O}(1)$ time if there is a path from u to v avoiding an arc a, for any pair of query vertices u and v, and any query arc a, or if there is a path from u to v avoiding a vertex w, for any query vertices u, v, and w. We can also report all the arcs or vertices that appear in all paths from u to v, for any query vertices u and v.

Related Work. To the best of our knowledge, ours is the first work that considers the all-pairs 2-reachability problem and gives a fast algorithm for it. In recent work Georgiadis et al. [19] investigate the effect of an arc or a vertex failure in a digraph D with respect to strong connectivity. Specifically, they show how to preprocess D in $\mathcal{O}(m+n)$ time in order to answer various sensitivity queries regarding strong connectivity in D under an arbitrary arc or vertex failure. For instance, they can compute in $\mathcal{O}(n)$ time the strongly connected components (SCCs) that remain in D after the deletion of an arc or a vertex, or report various statistics such as the number of SCCs in constant time per query (failed) arc or vertex. This result, however, cannot be applied for the solution of the 2-reachability problem. The reason is that if the deletion of an arc a leaves two vertices u and v in different SCCs in $D \setminus a$, the algorithm of [19] is not able to distinguish if there is still a path or no path from u to v in $D \setminus a$.

Previously, King and Sagert [29] gave an algorithm that can quickly answer sensitivity queries for reachability in a directed acyclic graph (DAG). Specifically, they show how to process a DAG D so that, for any pair of query vertices x and y, and a query arc a, one can test in constant time if there is a path from x to y in $D \setminus a$. Note that the result of King and Sagert does not yield an efficient solution to the all-pairs 2-reachability problem, since we need $\mathcal{O}(m)$ queries just to find if there is a separating arc for a single pair of vertices. Moreover, their preprocessing time is $\mathcal{O}(n^3)$.

Another interesting fact that arises from our work is that, somewhat surprisingly, computing all dominator trees in dense graphs is currently faster than computing a spanning arborescence from each vertex. The best algorithm for this problem is given by Alon et al. [2], who studied the problem of constructing a BFS tree from every vertex, and gave an algorithm that runs in $\mathcal{O}(n^{(3+\omega)/2})$ time.

Our algorithm uses fast matrix multiplication. Several other important graph-theoretic and network optimization problems can be solved by reductions to fast matrix multiplication. These include finding maximum weight matchings [44], computing shortest paths [57], and finding least common ancestors in DAGs [10] and junctions in DAGs [56]. Our algorithms can be used for constant-time queries on whether there exists a path from vertex u to vertex v avoiding an arc a (called *avoiding path*). This notion is closely related to a *replacement path* [20, 54, 55] (for which we additionally require to be shortest in $G \setminus a$).

Our Techniques. Our result is based on two novel approaches, one for DAGs and one for strongly connected digraphs. For DAGs we develop an algebra that operates on paths. We then use some version of 1-superimposed coding to apply our path algebra in a divide and conquer approach. This allows us to use Boolean matrix multiplication, in a similar vein to the computation of transitive closure. Unfortunately, our algebraic approach does not work for strongly connected digraphs. In this case, we exploit dominator trees in order to transform a strongly connected digraph D into two auxiliary graphs, so as to reduce 2-reachability queries in D to 1-reachability queries in those auxiliary graphs. This reduction works only for strongly connected digraphs and does not carry over to general digraphs. Our algorithm for general digraphs is obtained via a careful combination of those two approaches.

Roadmap. The remainder of this section is organized as follows. After introducing some definitions and notation in Section 3.2, we present our algorithm in three steps. In Section 3.3 we describe our approach for acyclic graphs, Section 3.4 covers strongly connected graphs and Section 3.5 describes their combination for arbitrary digraphs. We provide a matching lower bound and extend our approach to vertex-disjointness in Sections 3.6 and 3.7, respectively. Finally, Section 3.8 lists several applications of our algorithm.

3.2 Preliminaries

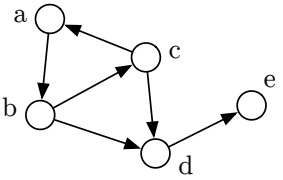
2-Reachability and 2-Reachability closure. We write $u \rightsquigarrow_{2a} v$ (resp., $u \rightsquigarrow_{2v} v$) to denote that there are two *arc-disjoint* (resp., internally *vertex-disjoint*) paths from u to v, and $u \not\rightsquigarrow_{2a} v$ (resp., $u \not\rightsquigarrow_{2v} v$) otherwise. As a special case, we assume that $v \rightsquigarrow_{2a} v$ (resp., $v \rightsquigarrow_{2v} v$) for each vertex v in D.

We define an abstract set $A^+ = A \cup \{\top, \bot\}$. The semantic of this set is as follows: $a \in A$ corresponds to an arc a separating two vertices, $\top$ corresponds to $\rightsquigarrow_{2a}$ (no single arc separates) and $\bot$ corresponds to $\not\rightsquigarrow$ (no arc is necessary for separation, vertices are already separated). Given a digraph D, we define the 2-*reachability closure* of D, denoted by $D^{\rightsquigarrow_{2a}}$, to be a matrix such that:

$$D^{\rightsquigarrow_{2a}}[u,v] \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } u \rightsquigarrow_{2a} v \\ \bot & \text{if } u \not\rightsquigarrow v \\ a & \text{where } a \text{ is any separating arc for } u \text{ and } v. \end{cases}$$

Since $v \rightsquigarrow_{2a} v$ for each $v \in V$, $D^{\rightsquigarrow_{2a}}[v,v] = \top$. An example of a graph with a 2-reachability closure matrix is given in Figure 9. Note that a 2-reachability closure matrix is not necessarily unique, as there might be multiple separating arcs for a given vertex pair. We define the 2-*reachability left closure* $D_L^{\rightsquigarrow_{2a}}$ by replacing *any separating arc* with *first separating arc* and the 2-*reachability right closure* $D_R^{\rightsquigarrow_{2a}}$ by replacing it with *last separating arc*.

Note that if there is only one edge separating u and v, then $D^{\rightsquigarrow_{2a}}[u,v] = D_L^{\rightsquigarrow_{2a}}[u,v] = D_R^{\rightsquigarrow_{2a}}[u,v]$. Given any 2-reachability closure matrix, one can compute efficiently the 2-reachability left and right closure matrices. We sketch below the basic idea for the left closure (the right closure being completely symmetric). Let u and v be any two vertices. If $D^{\rightsquigarrow_{2a}}[u,v]$ is either



$$\begin{bmatrix} \top & (a,b) & (a,b) & (a,b) & (a,b) \\ (b,c) & \top & (b,c) & \top & (d,e) \\ (c,a) & (c,a) & \top & \top & (d,e) \\ \bot & \bot & \bot & \top & (d,e) \\ \bot & \bot & \bot & \bot & \top \end{bmatrix}$$

Figure 9 A graph and its (non-unique) 2-reachability closure matrix.

$\top$ or $\bot$, then $D_L^{\rightsquigarrow_{2a}}[u,v] = D^{\rightsquigarrow_{2a}}[u,v]$. Otherwise, let $D^{\rightsquigarrow_{2a}}[u,v] = (x,y)$: if $u \rightsquigarrow_{2a} x$ (i.e., if $D^{\rightsquigarrow_{2a}}[u,x] = \top$) then (x,y) is the first separating edge for u and v and $D_L^{\rightsquigarrow_{2a}}[u,v] = (x,y)$; otherwise, $u \not\rightsquigarrow_{2a} x$ (i.e., $D^{\rightsquigarrow_{2a}}[u,x] \neq \top$) and $D_L^{\rightsquigarrow_{2a}}[u,v] = D_L^{\rightsquigarrow_{2a}}[u,x]$. Algorithm 15 gives the pseudocode for computing the 2-reachability left and right closures $D_L^{\rightsquigarrow_{2a}}$ and $D_R^{\rightsquigarrow_{2a}}$ in a total of $\mathcal{O}(n^2)$ worst-case time.

3.3 All-pairs 2-Reachability in DAGs

In this section, we present our $\mathcal{O}(n^\omega \log n)$ time algorithm for all-pairs 2-reachability in DAGs. The high-level idea is to mimic the way Boolean matrix multiplication can be used to compute the transitive closure of a graph: recursively along a topological order, combine the transitive closure of the first and the second half of the vertices in a single matrix multiplication. However, while in transitive closure for each pair (i,j) we have to store only information on whether there is a path from i to j, for all-pairs 2-reachability this is not enough. First, we describe a path algebra, used by our algorithm to operate on paths between pairs of vertices in a concise manner. We then continue with the description of a matrix product-like operation, which is the backbone of our recursive algorithm. Finally, we show how to implement those operations efficiently using some binary encoding and decoding at every step of the recursion.

Before introducing our new algorithm, we need some terminology.

Definition 3.1 (Arc split). *Let $D = (V, A)$ be a DAG, and let A_1, A_2 be a partition of its arc set A, $A = A_1 \cup A_2$. We say that a partition is an* arc split *if there is no triplet of vertices x, y, z in D such that $(x,y) \in A_2$ and $(y,z) \in A_1$ simultaneously.*

Informally speaking, under such an arc split, any path in D from a

Algorithm 15 Closure recovery

Require: $N \times N$ matrix IN of 2-reachability closure, type $\in \{\mathsf{left}, \mathsf{right}\}$
Ensure: Matrix OUT of 2-reachability left or right closure.

```
1: procedure RECOVERY(IN, type)
2:     OUT ← N × N matrix of undefined
3:     for i ← 1 to N do
4:         for j ← 1 to N do
5:             AUX(i, j, IN, OUT, type)
6:         end for
7:     end for
8: end procedure
9: procedure AUX(u, v, IN, OUT, type)
10:    if OUT[u][v] is undefined then
11:        if IN[u][v] = ⊤ or IN[u][v] = ⊥ then
12:            OUT[u][v] ← IN[u][v]
13:        else
14:            (x, y) ← IN[u][v]
15:            if type = left then
16:                if IN[u][x] = ⊤ then
17:                    OUT[u][v] ← (x, y)
18:                else
19:                    OUT[u][v] ← AUX(u, x, IN, OUT, type)
20:                end if
21:            else                                   ▷ type = right
22:                if IN[y][v] = ⊤ then
23:                    OUT[u][v] ← (x, y)
24:                else
25:                    OUT[u][v] ← AUX(y, v, IN, OUT, type)
26:                end if
27:            end if
28:        end if
29:    end if
30:    return OUT[u][v]
31: end procedure
```

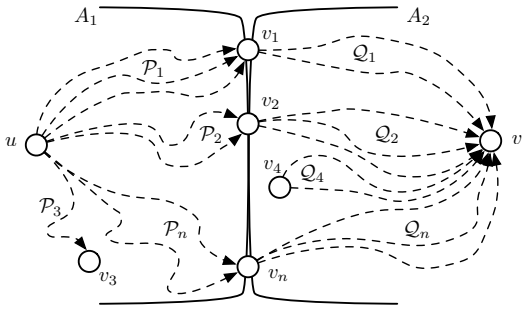


Figure 10 An arc split of a DAG $D = (V, A_1, A_2)$.

vertex u to a vertex v consists of a sequence of arcs from A_1 followed by a sequence of arcs from A_2 (as a special case, any of those sequences can be empty). We denote the arc split by $D = (V, A_1, A_2)$ (see Figure 10). We say that vertex x in $D = (V, A_1, A_2)$ is on the *left* (resp., *right*) *side* of the partition if x is adjacent only to arcs in A_1 (resp., A_2). We assume without loss of generality that the vertices of D are given in a topological ordering $v_1, v_2, \ldots, v_n$.

3.3.1 Algebraic approach

Consider a family of paths $\mathcal{P} = \{P_1, P_2, \ldots, P_\ell\}$, all sharing the same starting and ending vertices u and v. We would like to distinguish between the following three possibilities: (i) $\mathcal{P}$ is empty; (ii) at least one arc a belongs to every path $P_i \in \mathcal{P}$; or (iii) there is no arc that belongs to all paths in (nonempty) $\mathcal{P}$. To do that, we define the *representation* $\mathrm{repr}(\mathcal{P})$:

$$\mathrm{repr}(\mathcal{P}) \stackrel{\text{def}}{=} \bigcap_{i=1}^{\ell} P_i = \begin{cases} \mathbb{U} & \text{if } \mathcal{P} = \emptyset \\ \emptyset & \text{if no arc belongs to all } P_i \\ \{a \in A : a \in P_i,\ 1 \le i \le \ell\} & \text{otherwise.} \end{cases}$$

where $\mathbb{U}$ denotes the top symbol in the Boolean algebra of sets (i.e., the complement of $\emptyset$).

We also define a *left representation* $\mathrm{repr}_L(\mathcal{P}) \in A^+$, where $A^+ = A \cup$

$\{\top, \bot\}$, as follows:

$$\mathrm{repr}_L(\mathcal{P}) \stackrel{\text{def}}{=} \begin{cases} \bot & \text{if } \mathcal{P} = \emptyset \\ \top & \text{if no arc belongs to all } P_i \\ a & \text{such that } a \in P_i, 1 \le i \le \ell, \text{ and } \mathrm{tail}(a) \\ & \text{is } \textit{minimum} \text{ in the topological order} \end{cases}$$

A *right representation* $\mathrm{repr}_R(\mathcal{P}) \in A^+$ is defined symmetrically to $\mathrm{repr}_L(\mathcal{P})$, by replacing *minimum* with *maximum*. If $\mathrm{repr}_L(\mathcal{P}) \in A$ (resp., $\mathrm{repr}_R(\mathcal{P}) \in A$), we say that $\mathrm{repr}_L(\mathcal{P})$ (resp., $\mathrm{repr}_R(\mathcal{P})$) is the *first* (resp., *last*) *common arc* in $\mathcal{P}$. Note that if $\mathcal{P}$ is the set of *all the paths* from u to v, then $\mathrm{repr}(\mathcal{P})$ contains all the information about $D^{\rightsquigarrow 2a}[u, v]$. Additionally, $D_L^{\rightsquigarrow 2a}[u, v] = \mathrm{repr}_L(\mathcal{P})$ and $D_R^{\rightsquigarrow 2a}[u, v] = \mathrm{repr}_R(\mathcal{P})$. With a slight abuse of notation we also say that $D^{\rightsquigarrow 2a}[u, v] \in \mathrm{repr}(\mathcal{P})$.

Observation 3.2. *Let $D = (V, A_1, A_2)$ be an arc split of a DAG, and let u and v be two arbitrary vertices in D. For $1 \le i \le n$, let $\mathcal{P}_i = \{P \subseteq A_1 : P$ is a path from u to $v_i\}$, and $\mathcal{Q}_i = \{Q \subseteq A_2 : Q$ is a path from v_i to $v\}$ (see Figure 10) and let $\mathcal{S}$ be the family of all paths from u to v. Then:* $\mathrm{repr}(\mathcal{S}) = \bigcap_{i=1}^{n} \Big(\mathrm{repr}(\mathcal{P}_i) \cup \mathrm{repr}(\mathcal{Q}_i)\Big)$.

A straightforward application of Observation 3.2 yields immediately a polynomial time algorithm for computing $D^{\rightsquigarrow 2a}$. However, this algorithm is not very efficient, since the size of $\mathrm{repr}(\mathcal{P})$ can be as large as $(n-1)$. In the following, we will show how to obtain a faster algorithm, by replacing $\mathrm{repr}(\mathcal{P})$ with a suitable combination of $\mathrm{repr}_L(\mathcal{P})$ and $\mathrm{repr}_R(\mathcal{P})$.

▶ **Lemma 3.3.** *Let $D, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 3.2. If v_i is such that both $u \rightsquigarrow v_i$ in A_1 and $v_i \rightsquigarrow v$ in A_2 (both $\mathcal{P}_i \neq \emptyset$ and $\mathcal{Q}_i \neq \emptyset$), then*

(a) if $\mathrm{repr}_L(\mathcal{S}) = a \in A_1$ and $u \neq v_i$, then $\mathrm{repr}_L(\mathcal{P}_i) = a$;

(b) if $\mathrm{repr}_R(\mathcal{S}) = a \in A_2$ and $v_i \neq v$, then $\mathrm{repr}_R(\mathcal{Q}_i) = a$.

Proof. We only prove (a), since (b) is completely analogous. Assume by contradiction that $\mathrm{repr}_L(\mathcal{S}) = a \in A_1$, $u \neq v_i$ and $v_i \rightsquigarrow v$ in A_2, but $\mathrm{repr}_L(\mathcal{P}_i) = a' \neq a$. Since $\mathrm{repr}_L(\mathcal{S}) = a \in A_1$, it must be $a \in \mathrm{repr}(\mathcal{P}_i)$, as otherwise we would have a path $u \rightsquigarrow v_i \rightsquigarrow v$ avoiding a. Since $a \in \mathrm{repr}(\mathcal{P}_i)$ and $\mathrm{repr}_L(\mathcal{P}_i) = a' \neq a$, all paths in $\mathcal{P}_i$ must go first from u to arc a', then to arc a and finally to v_i. However, since $\mathrm{repr}_L(\mathcal{S}) = a \in A_1$, then a is reachable from u by a path avoiding a'. By definition of the arc split, this

path must be fully contained in A_1, which contradicts the fact that arc a' precedes a in all paths in $\mathcal{P}_i$. ◀

It is important to note that Lemma 3.3 holds regardless of whether u and v are on the same side of the partition or not.

Next, we define two operations, denoted as *serial* and *parallel*. Although those operations are formally defined on $A^+ = A \cup \{\top, \bot\}$, they have a more intuitive interpretation as operations on path families. We start with the serial operation $\otimes$. For $a, b \in A^+$, we define:

$$a \otimes b \stackrel{\text{def}}{=} \begin{cases} (\bot, \bot) & \text{if } a = \bot \text{ or } b = \bot \\ (a, b) & \text{otherwise.} \end{cases}$$

We define $\oplus$ as the parallel operator. Namely, for arbitrary $a \in A^+$: $a \oplus \bot \stackrel{\text{def}}{=} a$, $\bot \oplus a \stackrel{\text{def}}{=} a$, $a \oplus \top \stackrel{\text{def}}{=} \top$, $\top \oplus a \stackrel{\text{def}}{=} \top$, and otherwise, for $e, e' \in A$:

$$e \oplus e' \stackrel{\text{def}}{=} \begin{cases} \top & \text{if } e \neq e' \\ e & \text{if } e = e' \end{cases}$$

We extend the definition of $\oplus$ to operate on elements of $A^+ \times A^+$, as follows: $(a_1, b_1) \oplus (a_2, b_2) \stackrel{\text{def}}{=} (a_1 \oplus a_2, b_1 \oplus b_2)$. Ideally, we want the operator $\oplus$ either to preserve consistently the first common arc or to preserve consistently the last common arc, under the union of path families. If for instance we preserve the first common arc, that means that if $\mathcal{P}$ and $\mathcal{P}'$ are two path families sharing the same endpoints then we want $\text{repr}_L(\mathcal{P} \cup \mathcal{P}') = \text{repr}_L(\mathcal{P}) \oplus \text{repr}_L(\mathcal{P}')$ to hold. However, this is not necessarily the case, as for example both $\mathcal{P}$ and $\mathcal{P}'$ could consist of a single path, with both paths sharing an intermediate arc e', but both with two different initial arcs, respectively e_1 and e_2. Thus $\text{repr}_L(\mathcal{P}) \oplus \text{repr}_L(\mathcal{P}') = e_1 \oplus e_2 = \top$ while $\text{repr}_L(\mathcal{P} \cup \mathcal{P}') = e'$. As shown in the following lemma, this is not an issue if the path families considered are exhaustive in taking every possible path between a pair of vertices.

▶ **Lemma 3.4.** *Let $D, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 3.2. Then:*

(a) $\bigoplus_{i=1}^{n}(\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\bot, \bot)$ *iff* $\text{repr}(\mathcal{S}) = \mathbb{U}$;

(b) if $\bigoplus_{i=1}^{n}(\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (a_1, \top)$ *then* $\text{repr}(\mathcal{S}) \ni a_1$;

(c) if $\bigoplus_{i=1}^{n}(\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\top, a_2)$ *then* $\text{repr}(\mathcal{S}) \ni a_2$;

(d) if $\bigoplus_{i=1}^{n}(\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (a_1, a_2)$ *then* $\text{repr}(\mathcal{S}) \ni a_1, a_2$;

(e) $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\top, \top)$ *iff* $\mathrm{repr}(\mathcal{S}) = \emptyset$.

Proof. We proceed with a case analysis:

(a) $\mathrm{repr}(\mathcal{S}) = \mathbb{U}$ iff $\mathcal{S} = \emptyset$ iff $\forall_i$ ($\mathcal{P}_i = \emptyset$ or $\mathcal{Q}_i = \emptyset$) iff $\forall_i$ $\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i) = (\bot, \bot)$ iff $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\bot, \bot)$.

(b) Let $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (e_1, \top)$. By definition of $\oplus$ and $\otimes$, we must have that $\forall_i(\mathrm{repr}_L(\mathcal{P}_i) \in \{e_1, \bot\} \vee \mathrm{repr}_R(\mathcal{Q}_i) = \bot)$ and there must be at least one j such that $\mathrm{repr}_L(\mathcal{P}_j) = e_1$ and $\mathrm{repr}_R(\mathcal{Q}_j) \neq \bot$. Hence, any path in $\mathcal{S}$ must contain e_1.

(c) The proof is similar to (b).

(d) The proof is again similar to (b).

(e) If $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) \neq (\top, \top)$, then by cases (b), (c) or (d) it follows that $\mathrm{repr}(\mathcal{S}) \neq \emptyset$, clearly a contradiction.

To prove the other direction, assume that $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\top, \top)$ and $\mathrm{repr}(\mathcal{S}) \neq \emptyset$. From case (a) we know that $\mathrm{repr}(\mathcal{S}) \neq \mathbb{U}$, and thus there exists an arc $a \in A$ such that $a \in \mathrm{repr}(\mathcal{S})$. Without loss of generality, assume that $a \in A_1$. Then, it must be $\mathrm{repr}_L(\mathcal{S}) = a'$ for some arc $a' \in A_1$. Without loss of generality, assume that $v_1, v_2, \ldots, v_j$, are all the vertices such that simultaneously $u \rightsquigarrow v_i$ in A_1, $v_i \rightsquigarrow v$ in A_2 and $u \neq v_i$ (there is at least one such vertex, since $\mathcal{S} \neq \emptyset$ and $\mathrm{repr}(\mathcal{S}) \cap A_1 \neq \emptyset$). By Lemma 3.3(a), $a' = \mathrm{repr}_L(\mathcal{P}_i)$, $1 \leq i \leq j$. Thus:

$$
\begin{aligned}
&\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) \\
=&\bigoplus_{i=1}^{j}(a', \mathrm{repr}_R(\mathcal{Q}_i)) \oplus \bigoplus_{i=j+1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) \\
=&\bigoplus_{i=1}^{j}(a', \mathrm{repr}_R(\mathcal{Q}_i)) \oplus \bigoplus_{i=j+1}^{n}(\bot, \bot) \\
=&(a', \bigoplus_{i=1}^{j}\mathrm{repr}_R(\mathcal{Q}_i)) \neq (\top, \top),
\end{aligned}
$$

where we have used that (i) if $v_i = u$, then $\mathcal{Q}_i = \emptyset$, as otherwise $u \rightsquigarrow v$ in A_2, with a path avoiding $a \in A_1$, and (ii) by the choice of j, for $i > j$, $\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i) = (\bot, \bot)$. Thus we have a contradiction. ◀

We now consider the special case where one side of the partition defined in Observation 3.2 contains only paths of length one. In particular, we say that the arc set $A' \subseteq A$ is *thin*, if there exists no triplet of vertices x, y, z such that $(x, y) \in A'$ and $(y, z) \in A'$.

▶ **Lemma 3.5.** *Let $D, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 3.2. Additionally, let A_1 be thin. Then*

(a) $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\bot, \bot)$ *iff* $\mathrm{repr}_R(\mathcal{S}) = \bot$*;*

(b) if $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (a_1, \top)$ *then* $\mathrm{repr}_R(\mathcal{S}) = a_1$*;*

(c) if $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\top, a_2)$ *then* $\mathrm{repr}_R(\mathcal{S}) = a_2$*;*

(d) if $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (a_1, a_2)$ *then* $\mathrm{repr}_R(\mathcal{S}) = a_2$*;*

(e) $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\top, \top)$ *iff* $\mathrm{repr}_R(\mathcal{S}) = \top$.

Proof. Since A_1 is thin, we have that for each i: (i) $\mathrm{repr}_L(\mathcal{P}_i) = (u, v_i)$ iff $(u, v_i) \in A_1$, (ii) $\mathrm{repr}_L(\mathcal{P}_i) = \top$ iff $u = v_i$ and (iii) $\mathrm{repr}_L(\mathcal{P}_i) = \bot$, otherwise. We proceed with a case analysis as in Lemma 3.4.

(a) Since $\mathrm{repr}_R(\mathcal{S}) = \bot$ iff $\mathrm{repr}(\mathcal{S}) = \mathbb{U}$, this case follows immediately from Lemma 3.4(a).

(b) The condition implies that there must be v_j such that $a_1 = (u, v_j)$ and $v_j \rightsquigarrow_{2a} v$ in A_2. Additionally, for all $i \neq j$ such that $v_i \neq u$, either $(u, v_i) \notin A_1$ or $v_i \not\rightsquigarrow v$ in A_2, and for $v_i = u$ there is $v_i \not\rightsquigarrow v$ in A_2. It follows that every path in G from u to v must go through vertex v_j, and since A_1 is thin, this makes a_1 the separating arc. Since $v_j \rightsquigarrow_{2a} v$, arc a_1 is the only possible separating arc for u and v. Hence, $\mathrm{repr}_R(\mathcal{S}) = a_1$.

(c) The condition implies that for any $1 \leq j \leq n$, exactly one of the constraints is satisfied: (i) $u \rightsquigarrow v_j$ in A_1 (equivalently $v_j = u$ or $(u, v_j) \in A_1$) and $\mathrm{repr}_R(\mathcal{Q}_j) = a_2$, (ii) $u \not\rightsquigarrow v_j$ in A_1 (that is, $v_j \neq u$ and $(u, v_j) \notin A_1$) or (iii) $v_j \not\rightsquigarrow v$ in A_2. Additionally, unless there exists a j such that $v_j = u$ (which would mean $u \rightsquigarrow_{2a} v_j$), the first constraint is satisfied for at least two distinct values of j since the conditions (ii) and (iii) are not sufficient to satisfy $\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i)) = (\top, a_2)$. It follows that $\mathrm{repr}_R(\mathcal{S}) = a_2$.

(d) The condition implies that for exactly one j, there exists an arc $a_1 = (u, v_j)$ in A_1 and $v_j \rightsquigarrow v$ in A_2, and $\mathrm{repr}_R(\mathcal{Q}_j) = a_2$. Additionally, for every $i \neq j$, either $u \not\rightsquigarrow v_i$ in A_1 (that is, $v_i \neq u$ and $(u, v_i) \notin A_1$) or $v_i \not\rightsquigarrow v$ in A_2 (since otherwise there would be a path avoiding a_1). Similarly to case (c), it follows that $\mathrm{repr}_R(\mathcal{S}) = a_2$.

(e) Since $\mathrm{repr}_R(\mathcal{S}) = \top$ iff $\mathrm{repr}(\mathcal{S}) = \emptyset$, this case follows immediately from Lemma 3.4(e). ◀

One could prove a symmetric version of Lemma 3.5, with A_2 being thin. However, in the remainder, we stick with Lemma 3.5: namely, we choose a partition with a thin left side and thus break case (d) of Lemma 3.5 in favor of the rightmost arc (instead of the leftmost arc, as it would be in the symmetric version). Consistently, we define the following projection operator π: $\pi(\bot, \bot) \stackrel{\text{def}}{=} \bot$, $\pi(\top, \top) \stackrel{\text{def}}{=} \top$, $\pi(a', a) = \pi(\top, a) = \pi(a, \top) \stackrel{\text{def}}{=} a$. With this new terminology, Lemma 3.4 and Lemma 3.5 can be simply restated as follows:

▶ **Corollary 3.6.** *Let $D, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 3.2. Then*

(i) $\pi(\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i))) = \top$ *iff* $\mathrm{repr}(\mathcal{S}) = \emptyset$,

(ii) $\pi(\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i))) = \bot$ *iff* $\mathrm{repr}(\mathcal{S}) = \mathbb{U}$, *and*

(iii) $\pi(\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i))) \in \mathrm{repr}(\mathcal{S})$ *otherwise.*

▶ **Corollary 3.7.** *Let $G, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 3.2, and let A_1 be thin. Then* $\pi(\bigoplus_{i=1}^{n}(\mathrm{repr}_L(\mathcal{P}_i) \otimes \mathrm{repr}_R(\mathcal{Q}_i))) = \mathrm{repr}_R(\mathcal{S})$.

3.3.2 Matrix product

Now we define a *path-based matrix product* based on the previously defined operators: $(A \circ B)[i, j] \stackrel{\text{def}}{=} \pi\big(\bigoplus_k A[i, k] \otimes B[k, j]\big)$. Throughout, we assume that the vertices of D are sorted according to a topological ordering. In the following lemma, $\mathbf{F}$ represents a thin set of arcs (i.e., the set of arcs from a subset of vertices to another disjoint subset of vertices).

▶ **Lemma 3.8.** *Let* $\begin{bmatrix} E & F \\ 0 & G \end{bmatrix}$ *be the adjacency matrix of a DAG $D = (V, A)$, where E, F and G are respectively $k \times k$, $k \times (n-k)$ and $(n-k) \times (n-k)$ submatrices. If $\mathbf{F}$ is the matrix containing $\bot$ for every 0 in F and the appropriate $a \in A$ for every 1 in F, then:*

$$\begin{bmatrix} E_L^{\rightsquigarrow 2a} & E_L^{\rightsquigarrow 2a} \circ (\mathbf{F} \circ G_R^{\rightsquigarrow 2a}) \\ \bot & G_R^{\rightsquigarrow 2a} \end{bmatrix}$$

is a 2-reachability closure of D (not necessarily unique).

Proof. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the vertex set in order of rows and columns of the input matrix, and let $V_1 = \{v_1, v_2, \ldots, v_k\}$ and $V_2 = \{v_{k+1}, \ldots, v_n\}$. Matrices E, F and G correspond respectively to all arcs from V_1 to V_1, to all arcs from V_1 to V_2 and to all arcs from V_2 to V_2. We refer to the arc sets represented by those matrices as A_E, A_F and A_G. As a consequence of the fact that there are no arcs from V_2 to V_1, any path from V_1 to V_1 can use only arcs from A_E, and any path from V_2 to V_2 can use only arcs from A_G. Thus:

$$(V, A_E)_L^{\rightsquigarrow_{2a}} = \begin{bmatrix} E & 0 \\ 0 & 0 \end{bmatrix}_L^{\rightsquigarrow_{2a}} = \begin{bmatrix} E_L^{\rightsquigarrow_{2a}} & \bot \\ \bot & I \end{bmatrix},$$

$$(V, A_F)_L^{\rightsquigarrow_{2a}} = \begin{bmatrix} 0 & F \\ 0 & 0 \end{bmatrix}_L^{\rightsquigarrow_{2a}} = \begin{bmatrix} I & \mathbf{F} \\ \bot & I \end{bmatrix},$$

$$(V, A_G)_R^{\rightsquigarrow_{2a}} = \begin{bmatrix} 0 & 0 \\ 0 & G \end{bmatrix}_R^{\rightsquigarrow_{2a}} = \begin{bmatrix} I & \bot \\ \bot & G_R^{\rightsquigarrow_{2a}} \end{bmatrix},$$

where

$$I \stackrel{\text{def}}{=} \begin{bmatrix} \top & \bot & \cdots & \bot \\ \bot & \top & \cdots & \bot \\ \vdots & \vdots & \ddots & \vdots \\ \bot & \bot & \cdots & \top \end{bmatrix}$$

By Corollary 3.7 (since A_F is thin) and by definition of the path-based matrix product:

$$\begin{aligned}(V, A_F \cup A_G)_R^{\rightsquigarrow_{2a}} &= (V, A_F)_L^{\rightsquigarrow_{2a}} \circ (V, A_G)_R^{\rightsquigarrow_{2a}} \\ &= \begin{bmatrix} I & \mathbf{F} \\ \bot & I \end{bmatrix} \circ \begin{bmatrix} I & \bot \\ \bot & G_R^{\rightsquigarrow_{2a}} \end{bmatrix} = \begin{bmatrix} I & \mathbf{F} \circ G_R^{\rightsquigarrow_{2a}} \\ \bot & G_R^{\rightsquigarrow_{2a}} \end{bmatrix}.\end{aligned}$$

Finally, by Corollary 3.6:

$$\begin{aligned}(V, A)^{\rightsquigarrow_{2a}} &= (V, A_E)_L^{\rightsquigarrow_{2a}} \circ (V, A_F \cup A_G)_R^{\rightsquigarrow_{2a}} \\ &= \begin{bmatrix} E_L^{\rightsquigarrow_{2a}} & \bot \\ \bot & I \end{bmatrix} \circ \begin{bmatrix} I & \mathbf{F} \circ G_R^{\rightsquigarrow_{2a}} \\ \bot & G_R^{\rightsquigarrow_{2a}} \end{bmatrix} \\ &= \begin{bmatrix} E_L^{\rightsquigarrow_{2a}} & E_L^{\rightsquigarrow_{2a}} \circ (\mathbf{F} \circ G_R^{\rightsquigarrow_{2a}}) \\ \bot & G_R^{\rightsquigarrow_{2a}} \end{bmatrix}\end{aligned}$$

is a 2-reachability closure of D. ◀

By Lemma 3.8, the 2-reachability closure can be computed by performing path-based matrix products on the left and right 2-reachability closures of smaller matrices. This gives immediately a recursive algorithm for computing the 2-reachability closure: indeed, as already shown in Section 3.2, one can compute the left and right 2-reachability closures in $\mathcal{O}(n^2)$ time from any 2-reachability closure. In the next section we show how to implement this recursion efficiently by describing how to compute path-based matrix products efficiently.

3.3.3 Encoding and decoding for Boolean matrix product

We start this section by showing how to efficiently compute path-based matrix products using Boolean matrix multiplications. The first step is to encode each entry of the matrix as a bitword of length $8k$ where $k = \lceil \log_2(n+1) \rceil$. We use Boolean matrix multiplication of matrices of bitwords, with bitwise AND/OR operations, denoted respectively with symbols $\wedge$ and $\vee$. Our bitword length is $\mathcal{O}(\log n)$, so matrix multiplication takes $\mathcal{O}(n^\omega \log n)$ time by performing Boolean matrix multiplication for each coordinate separately.

We make use of the fact that after each multiplication we can afford a post-processing phase, where we perform actions which guarantee that the resulting bitwords represent a valid 2-reachability closure.

First, we note that when encoding a specific matrix, we know whether it is used as a left-side or as a right-side component of multiplication. The main idea is to encode left-side and right-side $\bot$ as $\{0\}^{8k}$, left-side and right-side $\top$ as $\{1\}^{8k}$. For any other value, append $\{1\}^{4k}$ as a prefix or suffix (depending on whether it is used as a left-side or right-side component), to the encoding of an arc. The encoding of an arc is a simple 1-superimposed code: the concatenation of the arc ID and the complement of the arc ID. To be more precise, whenever a bitword represents an arc a in a left-closure, then it is of the form $\mathrm{ID}_a\overline{\mathrm{ID}_a}\{1\}^{4k}$; whenever a bitword represents an arc a in a right-closure, then it is of the form $\{1\}^{4k}\mathrm{ID}_a\overline{\mathrm{ID}_a}$, where $\overline{w}$ denotes the complement of bitword w. The implementation of this encoding is given in Algorithm 16.

The serial operator $\otimes$ is implemented by coordinate-wise AND over two bitwords. Recall that the operator $\otimes$ always has as its first (left) operand an element from a left-closure matrix and as its second (right) operand an element from a right-closure. It is easy to verify that the result of

Algorithm 16 Left- and right-side encoding

Require: Matrix IN of dimension $N \times M$ with elements in A^+, type $\in$ {left, right}
Ensure: Matrix OUT of dimension $N \times M$ consisting of bitwords.

procedure ENCODE(IN, *type*)
  for all $IN[i][j]$ **do**
    if $IN[i][j] = \bot$ **then**
      $OUT[i][j] \leftarrow \{0\}^{8k}$
    else
      if $IN[i][j] = \top$ **then**
        $OUT[i][j] \leftarrow \{1\}^{8k}$
      else ▷ $IN[i][j] \in A$
        $(x, y) \leftarrow IN[i][j]$
        $b_1 b_2 \dots b_k \leftarrow$ binary encoding of x
        $c_1 c_2 \dots c_k \leftarrow$ binary encoding of y
        if *type* = left **then**
          $OUT[i][j] \leftarrow b_1 b_2 \dots b_k \, c_1 c_2 \dots c_k \overline{b_1 b_2 \dots b_k} \overline{c_1 c_2 \dots c_k} \{1\}^{4k}$
        else
          $OUT[i][j] \leftarrow \{1\}^{4k} b_1 b_2 \dots b_k \, c_1 c_2 \dots c_k \overline{b_1 b_2 \dots b_k} \overline{c_1 c_2 \dots c_k}$
        end if
      end if
    end if
  end for
end procedure

AND is a concatenation of two bitwords of length $4k$ encoding either $\bot, \top$ or $a \in A$. We observe that $\otimes$ is calculated properly in all cases: (let $a, a_1, a_2 \in A, a_1 \neq a_2$)

1. $a \otimes \top = (a, \top)$ since $\mathrm{ID}_a\overline{\mathrm{ID}_a}\{1\}^{4k} \wedge \{1\}^{8k} = \mathrm{ID}_a\overline{\mathrm{ID}_a}\{1\}^{4k}$
2. $\top \otimes a = (\top, a)$ since $\{1\}^{8k} \wedge \{1\}^{4k}\mathrm{ID}_a\overline{\mathrm{ID}_a} = \{1\}^{4k}\mathrm{ID}_a\overline{\mathrm{ID}_a}$
3. $a_1 \otimes a_2 = (a_1, a_2)$ since $\mathrm{ID}_{a_1}\overline{\mathrm{ID}_{a_1}}\{1\}^{4k} \wedge \{1\}^{4k}\mathrm{ID}_{a_2}\overline{\mathrm{ID}_{a_2}} = \mathrm{ID}_{a_1}\overline{\mathrm{ID}_{a_1}}\mathrm{ID}_{a_2}\overline{\mathrm{ID}_{a_2}}$
4. $a \otimes \bot = \top \otimes \bot = \bot \otimes \bot = \bot \otimes a = \bot \otimes \top = (\bot, \bot)$ since $\{0,1\}^{8k} \wedge \{0\}^{8k} = \{0\}^{8k} \wedge \{0,1\}^{8k} = \{0\}^{8k}$
5. $\top \otimes \top = (\top, \top)$ since $\{1\}^{8k} \wedge \{1\}^{8k} = \{1\}^{8k}$

The parallel operator $\oplus$ is implemented as coordinate-wise OR over bitwords of length $8k$. Note that all bitwords can be binary representations of pairs of elements in A^+ of the form $(a_1, a_2), (a_1, \top), (\top, a_2), (\bot, \bot), (\top, \top)$, since only those forms appear as a result of an $\otimes$ operation. Recall that $\oplus$ satisfies $(a_1, b_1) \oplus (a_2, b_2) = (a_1 \oplus a_2, b_1 \oplus b_2)$, thus w.l.o.g. it is enough to verify the correctness of the implementation over the first $4k$ bits of the encoding. Observe that all cases, except when both bitwords include encoded arcs, are managed correctly by the execution of coordinate-wise OR: (let $a \in A$)

1. $\bot \oplus \bot = \bot$ since $\{0\}^{4k} \vee \{0\}^{4k} = \{0\}^{4k}$
2. $\bot \oplus a = a \oplus \bot = a$ since $\mathrm{ID}_a\overline{\mathrm{ID}_a} \vee \{0\}^{4k} = \mathrm{ID}_a\overline{\mathrm{ID}_a}$
3. $\bot \oplus \top = \top \oplus \bot = \top$ since $\{1\}^{4k} \vee \{0\}^{4k} = \{1\}^{4k}$
4. $a \oplus \top = \top \oplus a = \top$ since $\mathrm{ID}_a\overline{\mathrm{ID}_a} \vee \{1\}^{4k} = \{1\}^{4k}$

We are only left to take care of operations of the form $a_1 \oplus a_2$ for $a_1, a_2 \in A$. According to the definition of the parallel operator $\oplus$, we would like $a_1 \oplus a_2 = a \in A$ iff $a_1 = a_2 = a$ and otherwise $a_1 \oplus a_2 = \top$. This special case is handled by the fact that we encode arcs using 1-superimposed codes. That is, the binary representation X of ID_a has the property that $X[1 \,..\, 2k] = \overline{X[2k+1 \,..\, 4k]}$. Moreover, the coordinate-wise OR of two encodings of arcs, that is $\mathrm{X} = \mathrm{ID}_{a_1}\overline{\mathrm{ID}_{a_1}} \vee \mathrm{ID}_{a_2}\overline{\mathrm{ID}_{a_2}}$, has this property iff $a_1 = a_2$. Or in other words: only when we compute the OR of two different arcs, the result will have more than half of its bits set.

Thus in order to successfully decode the result of chained $\oplus$ from coordinate-wise OR, we need to distinguish the following cases (our result is encoded as $\mathrm{X} = \mathrm{X}[1 \,..\, 2k]\mathrm{X}[2k+1 \,..\, 4k]$):

1. $\mathrm{X} = \{0\}^{4k}$, then the result is $\bot$,
2. $\mathrm{X}[1 \,..\, 2k] = \overline{\mathrm{X}[2k+1 \,..\, 4k]}$, then X is the encoding of the resulting arc,
3. otherwise the result is $\top$.

With all the tools and notation from above, the path-based matrix product over bitwords can be equivalently stated as $(A_L^{\rightsquigarrow 2a} \circ B_R^{\rightsquigarrow 2a})[i,j] \stackrel{\text{def}}{=} \textsc{Decode}\left(\bigvee_k \left(\textsc{Encode}(A_L^{\rightsquigarrow 2a}[i,k], \mathsf{left}) \wedge \textsc{Encode}(\mathsf{B}_\mathsf{R}^{\rightsquigarrow 2a}[\mathsf{k},\mathsf{j}], \mathsf{right})\right)\right)$, with the pseudocode for Decode being provided in Algorithm 17.

Algorithm 17 Decoding

Require: Matrix OUT of dimension $N \times M$ consisting of bitwords.
Ensure: Matrix IN of dimension $N \times M$ with elements in A^+.

```
1:  procedure DECODE(IN)
2:      for all IN[i][j] do
3:          b_1 b_2 ... b_8k ← IN[i][j]
4:          if b_1 b_2 ... b_8k = {0}^8k then
5:              OUT[i][j] ← ⊥
6:          else
7:              if b_4k+1 b_4k+2 ... b_6k = ¬b_6k+1 b_6k+2 ... b_8k then
8:                  x ← binary decoding of b_4k+1 b_4k+2 ... b_5k
9:                  y ← binary decoding of b_5k+1 b_5k+2 ... b_6k
10:                 OUT[i][j] ← (x, y)
11:             else
12:                 if b_1 b_2 ... b_2k = ¬b_2k+1 b_2k+2 ... b_4k then
13:                     x ← binary decoding of b_1 b_2 ... b_k
14:                     y ← binary decoding of b_k+1 b_k+2 ... b_2k
15:                     OUT[i][j] ← (x, y)
16:                 else
17:                     OUT[i][j] ← ⊤
18:                 end if
19:             end if
20:         end if
21:     end for
22: end procedure
```

To compute the entries of the final path-based matrix product (before the execution of Decode) it suffices to compute the bitwise Boolean matrix product of the appropriate bitword matrices. That is, we apply

Encode to $A_L^{\rightsquigarrow_{2a}}$ and $B_R^{\rightsquigarrow_{2a}}$, then we execute the Boolean matrix product for each coordinate separately, concatenate the coordinates of the resulting Boolean matrices into a matrix of bitwords, and finally execute the Decode operation from Algorithm 17. This is illustrated in Algorithm 18.

All the tools developed in this section allow us to compute the 2-reachability closure for DAGs. Our recursive algorithm follows closely Lemma 3.8, and its implementation in pseudocode is given as Algorithm 19. Since we implemented the right-side version of the projection, we only have to be careful to perform first the right multiplication before the left multiplication.

Algorithm 18 Path-based matrix product

Require: Matrices A and B of compatible dimension.
Ensure: Matrix being a path-based product of inputs is returned.

```
1: procedure MUL(A, B)
2:     return DECODE(ENCODE(A, left) · ENCODE(B, right))   ▷ here · denotes the coordinate-wise Boolean matrix multiplication
3: end procedure
```

▶ **Lemma 3.9.** *Given a DAG with n vertices, Algorithm 19 computes its 2-reachability closure in time $\mathcal{O}(n^\omega \log n)$.*

Proof. Algorithm 18 computes the path-based matrix product of matrices with every dimension bounded by n, if the initial graph size was n_0, in time $\mathcal{O}(n^\omega \log n_0)$, as it needs to compute $\mathcal{O}(\log n_0)$ Boolean matrix products, one for each coordinate of the stored bitwords. Closures are computed in time $\mathcal{O}(n^2)$. The recursion that captures the runtime of Algorithm 19 is thus given by the formula $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \mathcal{O}(n^\omega \log n_0)$ which is satisfied by setting $T(n) = \mathcal{O}(n^\omega \log n_0)$. The bound follows. ◀

3.4 All-pairs 2-Reachability in Strongly Connected Graphs

In this section, we focus on strongly connected graphs. In this case reachability is simple: for any pair of vertices $(u, v) \in V \times V$ we have $u \rightsquigarrow v$ in D. But in case that $u \not\rightsquigarrow_{2a} v$ in D, finding a separating arc that appears in

Algorithm 19 2-reachability closure for DAGs

Require: Matrix D of dimension $N \times N$, D is a DAG with topological order $1, 2, \ldots, n$.

Ensure: 2-reachability closure of G is returned.

procedure CLOSUREDAG(D)
  if $N = 1$ **then**
    return $\begin{bmatrix} \top \end{bmatrix}$
  end if
  $N' \leftarrow \lfloor N/2 \rfloor$
  $E \leftarrow D[1 \,..\, N'][1 \,..\, N']$
  $F \leftarrow D[1 \,..\, N'][(N'+1) \,..\, N]$, 0's replaced with $\perp$ and 1's with arc labels
  $G \leftarrow D[(N'+1) \,..\, N][(N'+1) \,..\, N]$
  $E' \leftarrow$ RECOVERY(CLOSUREDAG(E), left)
  $G' \leftarrow$ RECOVERY(CLOSUREDAG(G), right)
  return $\begin{bmatrix} E' & \text{MUL}(E', \text{MUL}(F, G')) \\ 0 & G' \end{bmatrix}$
end procedure

all paths from u to v in D can still be a challenge. We show that we can report such an arc in constant time after $\mathcal{O}(n^\omega)$ preprocessing. The main result of this section is the following theorem.

▶ **Theorem 3.10.** *The 2-reachability closure of a strongly connected graph can be computed in time $\mathcal{O}(n^\omega)$.*

Our construction is based on the notion of auxiliary graphs and it will be given in Section 3.4.3. A detailed implementation will be provided in Algorithms 20 and 21. Its running time will be analyzed in Lemma 3.16 and its correctness hinges on Lemma 3.18.

3.4.1 Reduction to two single-source problems

Let $D = (V, A)$ be a strongly connected digraph. Let s be a fixed but arbitrary vertex of D. The proof of the following lemma is immediate.

▶ **Lemma 3.11.** *For any pair of vertices u and v: If there is an arc $a \in A(D)$ such that $u \not\rightsquigarrow v$ in $D \setminus a$, then either $u \not\rightsquigarrow s$ in $D \setminus a$ or $s \not\rightsquigarrow v$ in $D \setminus a$.*

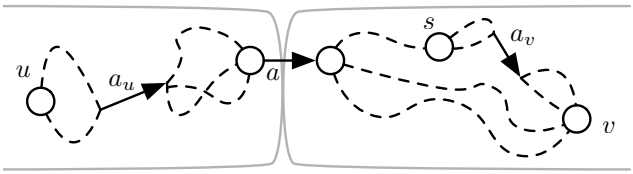


Figure 11 Illustration of the first case of Lemma 3.12.

Let $\mathcal{P}_{u,s}$ be the family of all paths from u to s and let $\mathcal{P}_{s,v}$ be the family of all paths from s to v. We denote by a_u the first arc on all paths in $\mathcal{P}_{u,s}$, and by a_v the last arc on all paths in $\mathcal{P}_{s,v}$. Note that there might be no arc that is on all paths of $\mathcal{P}_{u,s}$: in this case we say that a_u does not exist. If there are several arcs on all paths in $\mathcal{P}_{u,s}$, then they are totally ordered, so it is clear which is the *first* arc (similarly for a_v and $\mathcal{P}_{s,v}$). We now show that in order to search for a separation witness for (u, v), it suffices to focus on a_u and a_v.

▶ **Lemma 3.12.** *If there is some arc a such that $u \not\to v$ in $D \setminus a$, then at least one of the following statements is true:*

- *a_u exists and $u \not\to v$ in $D \setminus a_u$.*
- *a_v exists and $u \not\to v$ in $D \setminus a_v$.*

Proof. If $a = a_u$ or $a = a_v$, the claim is trivial. Otherwise, by Lemma 3.11, we know that $u \not\to s$ or $s \not\to v$ in $D \setminus a$. Let us assume that $u \not\to s$ (See Figure 11). So a lies not only on any path from u to v but also on any path from u to s. As a_u is the first common arc of every path from u to s, a_u also lies on every path from u to a. As all paths from u to v have to go through a, they also have to go through a_u and hence $u \not\to v$ in $D \setminus a_u$. If $s \not\to v$ in $D \setminus a$, we can show that $u \not\to v$ in $D \setminus a_v$ by the same extremality argument for a_v. ◀

Hence, in order to check whether there is an arc that separates u from v in D, it suffices to look at the reachability information in $D \setminus a_u$ (a graph which does not depend on v) and at the reachability information in $D \setminus a_v$ (a graph which does not depend on u). Unfortunately, this is not enough to derive an efficient algorithm, since we would have still to look at as many as $2n$ different graphs (as we explain later, and as it was first shown in [26], there can be at most $2n - 2$ arcs whose removal can affect the strong connectivity of the graph). As a result, computing the transitive closures of all those graphs would require $\mathcal{O}(n^{\omega+1})$ time. The key insight

to reduce the running time to $\mathcal{O}(n^\omega)$ is to construct an auxiliary graph H, whose reachability is identical to $D \setminus a_v$ for any query pair (u, v), and a second auxiliary graph H' whose reachability is identical to $D \setminus a_u$ for any query pair (u, v). Note that the arc that is missing from the graph depends always on one of the two endpoints of the reachability query. As a consequence, we have to consider only n^2 and not n^3 different queries for H and H'.

3.4.2 Strong bridges and dominator tree decomposition

Before we construct these auxiliary graphs, we need some more terminology and prior results.

Flow graphs, dominators, and bridges. A *flow graph* $D_s = (V, A, s)$ is a digraph with a distinguished *start vertex* s. We denote by $D_s^R = (V, A^R, s)$ the reverse flow graph of D_s; the graph resulted by reversing the direction of all arcs $a \in A$. Vertex u is a *dominator* of a vertex v (u *dominates* v) if every path from s to v in D_s contains u; u is a *proper dominator* of v if u dominates v and $u \neq v$. The dominator relation is reflexive and transitive. Its transitive reduction is a rooted tree, the *dominator tree* DT: u dominates v if and only if u is an ancestor of v in DT, see Figure 12 and Figure 13 for examples. If $v \neq s$, the parent of v in DT, denoted by $d(v)$, is the *immediate dominator* of v: it is the unique proper dominator of v that is dominated by all proper dominators of v. For any vertex v, we let $D(v)$ denote the set of descendants of v in DT, i.e., the vertices dominated by v. Lengauer and Tarjan [31] presented an algorithm for computing dominators in $\mathcal{O}(m\alpha(m, n))$ time for a flow graph with n vertices and m arcs, where α is a functional inverse of Ackermann's function [51]. Dominators can be computed in linear time [3, 8, 16]. An arc (x, y) is a *bridge* of the flow graph D_s if all paths from s to y include (x, y).

Strong bridges. Let $D = (V, A)$ be a strongly connected digraph. An arc a of D is a *strong bridge* if $D \setminus a$ is no longer strongly connected. Let s be an arbitrary start vertex of D. Since D is strongly connected, all vertices are reachable from s and reach s, so we can view both D and D^R as flow graphs with start vertex s, denoted respectively by D_s and D_s^R.

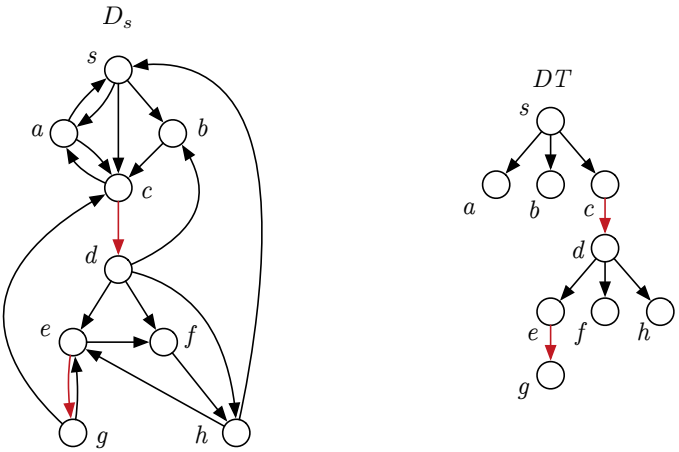


Figure 12 A flow graph and its dominator tree. Arcs marked in red are bridges.

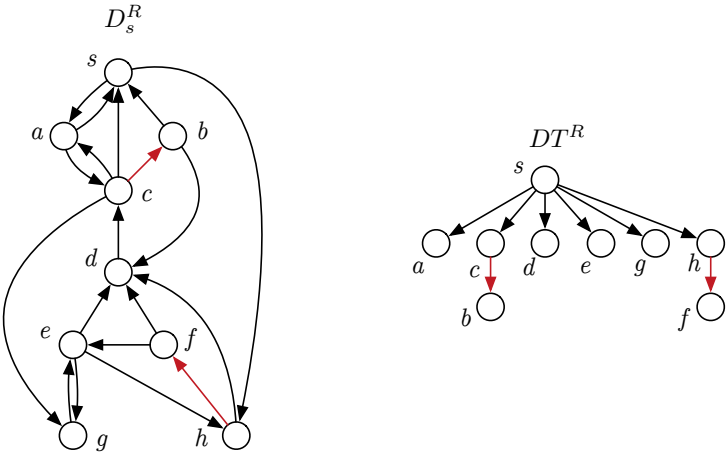


Figure 13 A reverse flow graph and its dominator tree. Arcs marked in red are bridges.

Property 3.13. ([26]) *Let s be an arbitrary start vertex of D. An arc $a = (x, y)$ is a strong bridge of D if and only if it is a bridge of D_s or a bridge of D_s^R (or both).*

As a consequence of Property 3.13, all the strong bridges of the digraph D can be obtained from the bridges of the flow graphs D_s and D_s^R, and thus there can be at most $(2n-2)$ strong bridges in a digraph D. Using the linear time algorithms for computing dominators, we can thus compute all strong bridges of D in time $\mathcal{O}(m+n) \subseteq \mathcal{O}(n^\omega)$. We use the following lemma from [18] that holds for a flow graph D_s of a strongly connected digraph D.

▶ **Lemma 3.14.** ([18]) *Let D be a strongly connected digraph and let (x, y) be a strong bridge of D. Also, let DT and DT^R be the dominator trees of the corresponding flow graphs D_s and D_s^R, respectively, for an arbitrary start vertex s.*

(a) Suppose $x = d(y)$. Let w be any vertex that is not a descendant of y in DT. Then there is a path from w to x in D that does not contain any proper descendant of y in DT. Moreover, all simple paths in D from w to any descendant of y in DT must contain the arc $(d(y), y)$.

(b) Suppose $y = d^R(x)$. Let w be any vertex that is not a descendant of x in DT^R. Then there is a path from x to w in D that does not contain any proper descendant of x in DT^R. Moreover, all simple paths in D from any descendant of x in DT^R to w must contain the edge $(x, d^R(x))$.

Bridge decomposition. After deleting from the dominator trees DT and DT^R respectively the bridges of D_s and D_s^R, we obtain the *bridge decomposition* of DT and DT^R into forests $\mathcal{DT}$ and $\mathcal{DT}^R$. Throughout this section, we denote by T_v (resp., T_v^R) the tree in $\mathcal{DT}$ (resp., $\mathcal{DT}^R$) containing vertex v, and by r_v (resp., r_v^R) the root of T_v (resp., T_v^R). Given a digraph $D = (V, A)$, and a set of vertices $S \subseteq V$, we denote by $D[S]$ the subgraph induced by S. In particular, $D[D(r)]$ denotes the subgraph induced by the descendants of vertex r in DT.

3.4.3 Overview of the algorithm and construction of auxiliary graphs

The high-level idea of our algorithm is to compute two auxiliary graphs H and H' from D and D^R, respectively, with the following property: Given

two vertices u and v, we have that $u \rightsquigarrow_{2a} v$ in D if and only $u \rightsquigarrow v$ in H and $v \rightsquigarrow u$ in H'. To construct the auxiliary graphs H and H', we use the bridge decompositions of DT and DT^R, respectively.

The two extremal arcs a_u and a_v, defined above, can also be defined in terms of the bridge decompositions. In particular, a_v is the bridge entering the tree T_v of the bridge decomposition of DT, so $a_v = (d(r_v), r_v)$, and a_u is the reverse bridge entering the tree DT^R_u of the bridge decomposition of DT^R, so $a_u = (r^R_u, d^R(r^R_u))$. Hence if there exists a path from u to v avoiding each of the strong bridges a_v and a_u, then $u \rightsquigarrow_{2a} v$ in D. By Lemma 3.12, it is enough if H models the reachability of $D \setminus a_v$ and H' the reachability of $D \setminus a_u$. So H is responsible for answering whether u has a path to v avoiding a_v, while H' is responsible for answering whether u has a path to v avoiding a_u. Then, if any of the reachability queries in H and H' returns false, we immediately have an arc that appears in all paths from u to v.

We next show to compute the auxiliary graphs H and H' in $\mathcal{O}(n^2)$ time.

Definition 3.15 (Auxiliary graph construction)**.** *The auxiliary graph $H = (V, A')$ of the flow graph $D_s = (V, A, s)$ is constructed as follows. Initially, $A' = A \setminus BR$, where BR is the set of bridges of D_s. For all bridges (p, q) of D_s do the following: For each arc $(x, y) \in A$ such that $x \in D(q), y \notin D(q)$, we add the arc (p, y) in A', i.e., we set $A' = A' \cup (p, y)$.*

A detailed implementation is provided in Algorithms 20 and 21. Together with graph H, the algorithm outputs an array of arcs ("witnesses") W, such that for each vertex $v \neq s$, $W[v] = (d(r_v), r_v)$ is a candidate separating arc for v and any other vertex. The computation of H' is completely analogous.

Once H and H' are computed, their transitive closure can be computed in $\mathcal{O}(n^\omega)$ time, after which reachability queries can be answered in constant time. Thus, we can preprocess a strongly connected digraph D in total time $\mathcal{O}(n^\omega)$ and answer 2-reachability queries in constant time, as claimed by Theorem 3.10.

▶ **Lemma 3.16.** *The auxiliary graph H can be computed in $\mathcal{O}(n^2)$ time and space.*

Proof. For each root r of a tree $T_r \in \mathcal{DT}$, we maintain a set $R(r) \subseteq V$, initially set to $\emptyset$. The value $R(r)$ contains all such endpoints y of arcs

Algorithm 20 2-reachability closure in strongly connected graphs

Require: Strongly connected digraph D on n vertices.
Ensure: 2-reachability closure of D is returned.

```
1:  procedure ClosureSCC(D)
2:      s ← arbitrary vertex of D
3:      H, W ← AuxiliaryGraph(D, s)
4:      H', W' ← AuxiliaryGraph(D^R, s)
5:      OUT ← n × n matrix
6:      for all OUT[i][j] do
7:          if (i, j) ∉ H then
8:              OUT[i][j] ← W[j]
9:          else
10:             if (j, i) ∉ H' then
11:                 OUT[i][j] ← W'[i]^R
12:             else
13:                 OUT[i][j] ← ⊤
14:             end if
15:         end if
16:     end for
17: end procedure
```

Algorithm 21 Auxiliary graph construction for strongly connected graphs

```
1: procedure AuxiliaryGraph(D, s)
2:     H ← D
3:     W, R ← arrays of size n
4:     DT ← dominator tree of D_s
5:     for tree T, rooted at r, in bottom-up order of the bridge decomposition of DT do
6:         R[r] ← ⋃ {R[r'] : d(r') ∈ V(T)}          ▷ r' is root of children component of T
7:         D[r] ← V(T) ∪ ⋃ {D[r'] : d(r') ∈ V(T)}
8:         for (x, y) ∈ V(T) × V(D) do
9:             if (x, y) ∈ D then
10:                R[r] ← R[r] ∪ {y}
11:            end if
12:        end for
13:        R[r] ← R[r] \ D[r]
14:        if s ∉ T then
15:            p ← d(r)          ▷ (p, r) is a bridge connecting the parent of T to T
16:            for y ∈ R[r] do
17:                H ← H ∪ (p, y)
18:            end for
19:            for x ∈ V(T) do
20:                W[x] ← (p, r)
21:            end for
22:            H ← H \ (p, r)
23:        end if
24:    end for
25:    return TransitiveClosure(H), W
26: end procedure
```

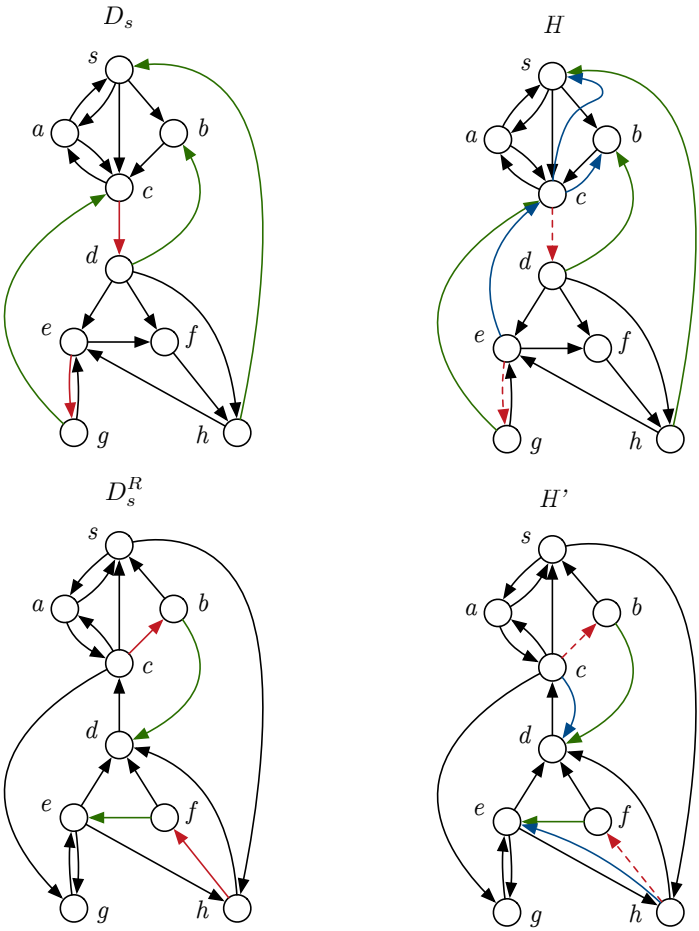


Figure 14 Auxiliary graphs H and H' which are derived from D_s and D_s^R, respectively. The deleted edges, the bridges of D_s and D_s^R, are shown in red, the newly added edges are shown in blue. The blue edges are drawn along those green edges from D_s and D_s^R which are the reason for their insertion. Here we see, that for example b is 2-reachable from e, since there are two (arc and vertex) disjoint paths (e, g, c, d, b) and (e, f, h, s, b) in D. In H, e reaches b through the path (e, c, b), and in H', b reaches e through the path (b, s, h, e). We also see, that edge (c, d) separates a and f in D, and even though f reaches a in H' through the path (f, d, c, a), a does not reach f in H. To illustrate why both H and H' are relevant in Lemma 3.18, consider the following example: vertex c is unreachable from b in $D \setminus (b, c)$, which we also detect as there is no c-b path in H' (even though there is a b-c path in H).

(x, y) such that $x \in D(r)$ and $y \notin D(r)$. We process the trees of the bridge decomposition in a bottom-up order of their roots. For each root r that we visit, we compute $R(r)$ in the following three steps. First, for each bridge (p, q) of D_s such that $p \in T_r$, we update $R(r)$ by setting $R(r) = R(r) \cup R(q)$. Second, for every arc (x, y) such that $x \in T_r$ we insert y into $R(r)$. Finally, we remove all $D(r)$, that is $R(r) = R(r) \setminus D(r)$. We execute this final step since we are only interested whether there is an arc (x, y) such that $x \in D(r)$ and $y \notin D(r)$. Clearly, after these steps the set $R(r)$ contains only the desired endpoints.

Note that we actually wish to insert arcs to $d(r)$ for each root r of a tree on the bridge decomposition. Therefore, after computing for each root r its set $R(r)$, we insert to H an arc $(d(r), y)$ for every $y \in R(r)$ (notice that the outgoing edges of $d(r)$ in D, except $(d(r), r)$, are also outgoing edges of $d(r)$ in H). Overall, by representing sets as bitmasks, all the $R(r)$ sets can be computed in $\mathcal{O}(n^2)$ time. We spend $\mathcal{O}(n^2)$ time in the third step, since we visit each vertex at most one. Since we traverse every arc only once, the second step takes $\mathcal{O}(n + m)$ in total. The bound follows. ◀

▶ **Lemma 3.17.** *For all $w \in V$, no arc $(x, y) \in A(H)$ exists with $x \notin D(r_w)$ and $y \in D(r_w)$.*

Proof. Assume, by contradiction, that there is an arc $(x, y) \in A(H)$ such that $x \notin D(r_w)$ and $y \in D(r_w)$. Since $(d(r_w), r_w)$ is a strong bridge in D_s, (x, y) does not exist in D (by Lemma 3.14). Hence, by construction, there is an arc $(z, y) \in A(D)$ where $z \in D(x) \setminus x$ and $y \notin D(x)$. Therefore, x cannot be an ancestor of w in DT, which implies $z \notin D(r_w)$ and $z \neq d(r_w)$ since $D(z) \cap D(r_w) = \emptyset$. This is a contradiction, since (z, y), where $z \notin D(r_w)$ and $y \in D(r_w)$, cannot exist in D by Lemma 3.14. ◀

To show the correctness of our approach, we consider queries where we are given an ordered pair of vertices (u, v), and we wish to return whether there exists an arc a such that $u \not\rightsquigarrow v$ in $D \setminus a$. We can answer this query in constant time by answering the queries $u \rightsquigarrow v$ in H and $v \rightsquigarrow u$ in H'. Given Lemma 3.12, it is sufficient to prove the following:

▶ **Lemma 3.18.** *The auxiliary graphs H and H' satisfy these two conditions:*

- *If a_v exists, then $u \rightsquigarrow v$ in $D \setminus a_v$ if and only if $u \rightsquigarrow v$ in H.*
- *If a_u exists, then $u \rightsquigarrow v$ in $D \setminus a_u$ if and only if $v \rightsquigarrow u$ in H'.*

We prove the lemma in two separate steps, one for each direction of the two equivalences.

▶ **Lemma 3.19.** *If $u \rightsquigarrow v$ in H then $u \rightsquigarrow v$ in $D \setminus a_v$ (and if $v \rightsquigarrow u$ in H' then $u \rightsquigarrow v$ in $D \setminus a_u$).*

Proof. By Lemma 3.17, there are no arcs incoming into $D(r_v)$ in H. So for a path P from u to v in H to exist, u must lie in $D(r_v)$ and P must lie within $H[D(r_v)]$. Clearly, if P contains only arcs from $A(D[D(r_v)])$, then P is also a valid path from u to v in $D[D(r_v)]$ and thus also in $D \setminus a_v$ (recall that $a_v = (d(r_v), r_v)$) and we are done.

Otherwise, we iteratively substitute auxiliary arcs of P, with paths in $D \setminus a_v$, so that in the end, P is fully contained within $D \setminus a_v$. Let $a^* = (x^*, y^*)$ be the first arc of P such that $a^* \notin A(D[D(r_v)])$, i.e., an auxiliary arc of H. By $a^o = (x^o, y^o)$, we denote the original arc for which we inserted a^* into H. Then $x^o \in D(x^*) \setminus x^*$ and $y^* = y^o$. Since all paths from s to x^o contain x^*, and all simple paths from x^* to x^o avoid vertices from $V \setminus D(r_v)$ (otherwise, if all paths contained such a vertex w then s would have a path to x^o in D avoiding x^* by Lemma 3.14), it follows that x^* has a path $P_{x^*x^o}$ to x^o in $D[D(r_v)]$. If we now replace a^* in P by $P_{x^*x^o} \cdot a^o$, then P contains a path from u to y^* containing only arcs in $D \setminus a_v$. We repeat this argument as long as P contains auxiliary arcs and get a path from u to v in $D \setminus a_v$.

The statement for H' and $D \setminus a_u$ can be shown with completely analogous arguments. ◀

▶ **Lemma 3.20.** *If $u \rightsquigarrow v$ in $D \setminus a_v$ then $u \rightsquigarrow v$ in H (and if $u \rightsquigarrow v$ in $D \setminus a_u$ then $v \rightsquigarrow u$ in H').*

Proof. By Lemma 3.14 (a), a_v is the only edge in D entering $D(r_v)$. So any path P from u to v in $D \setminus a_v$ can only use edges in $A(D[D(r_v)])$. If P only contains edges in H, we are done. Otherwise, let $a^* = (x^*, y^*)$ be the first edge of P that is in $A(D[D(r_v)])$ but not in H, hence a^* is a bridge of D_s. Let z^* be the first vertex on P after a^* that is not a descendant of y^* in DT. Such a vertex z^* exists since P ends at v but v is not a descendant of y^* (recall that a^* is a bridge and lies within $D[D(r_v)]$). Thus, we can replace the subpath of P between a^* and z^* (including a^*) by the edge (x^*, z^*) which is an auxiliary edge of H, by the definition of H. We repeat this argument as long as P contains bridges of D_s and get a path from u to v in H.

The statement for $D \setminus e_u$ and H' can be shown with completely analogous arguments. ◀

3.5 All-pairs 2-Reachability in General Graphs

In this section, we show how to compute the 2-reachability of a general digraph by suitably combining the previous algorithms for DAGs and for strongly connected digraphs. First, note that the 2-reachability closure of a strongly connected graph D can be constructed as follows: $D^{\rightsquigarrow 2a}[i,j] = \top$ if i has two arc-disjoint paths to j and $D^{\rightsquigarrow 2a}[i,j] \in A$ if there is an arc $a \in A$ such that $i \not\rightsquigarrow j$ in $D \setminus a$. No entry of $D^{\rightsquigarrow 2a}$ contains $\bot$ since D is strongly connected. After $\mathcal{O}(n^\omega)$ time preprocessing all the above queries can be answered in constant time. Therefore, the 2-reachability closure can be computed in $\mathcal{O}(n^\omega)$ time.

Let D be a general digraph. The condensation of D is the DAG resulting after the contraction of every strongly connected component of D into a single vertex. We assume, without loss of generality, that the vertices are ordered as follows: The vertices in the same strongly connected component of D appear consecutively in an arbitrary order, and the strongly connected components are ordered with respect to the topological ordering of the condensation of D. Moreover, we assume that we have access to a function StrongConnect(u, v) that answers whether the vertices u and v are strongly connected.

The key insight is that every idea presented in Section 3.3 never truly used the fact that the input graph is a DAG, just the properties of an arc split, that is finding an arc partition into two sets so that no vertex has incoming arcs from the second set and outgoing arcs from the first set simultaneously. If we are able to extend the definition of an arc split to a general graph in the way highlighted above, and the definitions of $\mathrm{repr}(), \mathrm{repr}_R()$ and $\mathrm{repr}_L()$, then all of the results from Section 3.3 carry over to a general graph D. Note that given an *arbitrary* path family $\mathcal{P}$, $\mathrm{repr}_L(\mathcal{P})$ and $\mathrm{repr}_R(\mathcal{P})$ might be ill-defined, since paths in an arbitrary path family might not share the order of common arcs. However, we are only using this notation for path families containing exactly all of the paths connecting a given pair of vertices in the graph: for such families, the order of common arcs is shared.

The high-level idea behind our approach is to extend the 2-reachability closure algorithm for DAGs, as follows. At each recursive call, the algo-

Algorithm 22 2-reachability closure in general graphs

Require: Matrix D of dimension $N \times N$, with vertices ordered w.r.t. some fixed topological order of the strongly connected components.
Ensure: 2-reachability closure of D is returned.

procedure Closure(D)
  if $N = 1$ **then**
    return $\begin{bmatrix} \top \end{bmatrix}$
  else
    if $\forall i, j \in \{1, \ldots, N\}$ StrongConnect$(i, j) =$ true **then**
      return ClosureSCC(D)
    end if
  end if
  $I = \{i : 1 \leq i < N, \text{StrongConnect}(i, i+1) = \text{false}\}$
  $N' \leftarrow \arg\min_{i \in I} \{|i - N/2|\}$
  $A \leftarrow D[1 \,..\, N'][1 \,..\, N']$
  $B \leftarrow D[1 \,..\, N'][(N'+1) \,..\, N]$, 0 replaced with $\perp$ and 1 with arc labels
  $C \leftarrow D[(N'+1) \,..\, N][(N'+1) \,..\, N]$
  $A' \leftarrow$ Recovery(Closure(A), left)
  $C' \leftarrow$ Recovery(Closure(C), right)
  return $\begin{bmatrix} A' & \text{Mul}(A', \text{Mul}(B, C')) \\ 0 & C' \end{bmatrix}$
end procedure

rithm attempts to find a balanced separation of the set of vertices, with respect to their fixed precomputed order, into two sets such that there is no pair across the two sets that is strongly connected. If such a balanced separation can be found, then the instance is (roughly) equally divided into two instances. Otherwise, if there is no balanced separation of the set of vertices into two subsets, then one of the following properties holds: *(i)* the larger instance is a strongly connected component, or *(ii)* the recursive call on the larger instance separates a large strongly connected component, on which we can compute the 2-reachability closure in $\mathcal{O}(n^\omega)$ time. We provide pseudocode for this in Algorithm 22.

▶ **Theorem 3.21.** *Algorithm 22 computes a 2-reachability closure of any graph on n vertices in time $\mathcal{O}(n^\omega \log n)$.*

Proof. The algorithm's correctness follows from the correctness of Algorithm 19 and the fact that Algorithm 22 separates the input matrix D with dimensions $N \times N$ into submatrices $A = D[1 \,..\, N'][1 \,..\, N']$, $B = D[1 \,..\, N'][(N'+1) \,..\, N]$ and $C = D[(N'+1) \,..\, N][(N'+1) \,..\, N]$, and such that $\forall i \in [1, N'], j \in [N'+1, N] : D[j, i] = 0$ as required by Lemma 3.8. Now we show that Algorithm 22 has the same asymptotic running time as Algorithm 19.

The recurrence that provides the running time is the following (we denote by N_0 the size of the original graph)

$$T(N) = T(\min\{N', N - N'\}) + T(\max\{N', N - N'\}) + \mathcal{O}(N^\omega \log N_0),$$

where N' is defined as in Algorithm 22, that is, N' is the i that minimizes $|i - N/2|$ and satisfies $\textsc{StrongConnect}(i, i+1) = \mathsf{false}$. Without loss of generality, assume that $N' \geq N/2$.

Denote by $N'' < N/2$ the start of the strongly connected component that ends at position N'. Observe that this component size satisfies $N' - N'' \geq 2(N' - N/2)$. We consider two cases, which intuitively distinguish whether the strongly connected component in the middle of the order is small or large:

1. If $N' \geq 2/3N$, then since $N/2 - N'' \geq N' - N/2$ (from the fact that N' is closest to $N/2$), we get $N'' \leq N - N' = (N - 3/2N') + N'/2 \leq N'/2$. This means that N'' is a splitting point in a recursive call on range $[0, N']$, and we get the bound

$$\begin{aligned} T(N) &= T(N'') + \mathcal{O}((N' - N'')^\omega) + T(N - N') + \mathcal{O}(N^\omega \log N_0) \\ &\leq T(1/3N) + T(1/3N) + \mathcal{O}(N^\omega \log N_0), \end{aligned}$$

   where we have used that our claimed runtime bound $T()$ is nondecreasing, so we can use bounds $N'' \leq 1/3N$ and $N - N' \leq 1/3N$.

2. If $N' \leq 2/3N$, then by the fact that $N - N' \leq 1/2N$

$$\begin{aligned} T(N) &= T(N') + T(N - N') + \mathcal{O}(N^\omega \log N_0) \\ &\leq T(N/2) + T(2/3N) + \mathcal{O}(N^\omega \log N_0). \end{aligned}$$

It is easy to see that $T(N) = C \cdot N^\omega \log N_0$ satisfies both of the recursive bounds (since $\omega \geq 2$), given large enough constant C. Thus plugging $N_0 = N$ for the total running time yields the claimed bound. ◀

3.6 Matching Lower Bounds

A simple construction shows that any 2-reachability oracle for strongly connected graphs can also be used as a reachability oracle for any graph. Let D be a DAG. We create a graph $\widehat{D} = (\widehat{V}, \widehat{A})$ from D as follows: we add two new vertices s and t together with the arc (s,t), and for each vertex $v \in V(D)$ we add the arcs (v,s) and (t,v). Clearly, $\widehat{D}$ is strongly connected since we added paths from each vertex u to any other vertex v, namely the path $\langle u,s,t,v\rangle$. All the new paths between vertices in $V(D)$ contain the arc (s,t). Therefore, a vertex u has two arc-disjoint paths to v in $\widehat{D}$, where $u,v \in V(D)$, if and only if u has a path to v in D.

Additionally, for general graphs, there cannot be a significantly faster all-pairs 2-reachability algorithm (than by a logarithmic factor), as our construction can produce all dominator trees, which by definition encode the necessary information to answer reachability queries in constant time. As computing reachability is asymptotically equivalent to matrix multiplication [13], there is no hope to solve all-pairs 2-reachability in $o(n^\omega)$.

3.7 Extension to Vertex-Disjoint Paths

Our approach can be modified so that it reports the existence of two vertex-disjoint (rather than arc-disjoint) paths for any pair of vertices. Although we can formulate the algorithms of Sections 3.3 and 3.4 so that they use separating vertices (rather than separating arcs), here we sketch how to obtain the same result via a standard reduction, which uses vertex-splitting. The details of the reduction are as follows. From the original digraph $D = (V,A)$, we compute a modified digraph $\widehat{D} = (\widehat{V}, \widehat{A})$ by replacing each vertex $v \in V$ by two vertices $v^+, v^- \in \widehat{V}$, together with the arc $(v^-, v^+) \in \widehat{A}$, and replacing each arc $(u,v) \in A$ by $(u^+, v^-) \in \widehat{A}$. (Thus, v^+ has the arcs leaving v, and v^- has the arcs entering v.) Then, for any pair of vertices $u,v \in V$, D contains two vertex-disjoint paths from u to v if and only if $\widehat{D}$ contains two arc-disjoint paths from u^+ to v^-. Suppose that we apply our algorithm to $\widehat{D}$. Let u,v be any vertices in D. If v is reachable from u in D but all paths from u to v in D contain a common vertex, then the algorithm reports a separating arc $a \in \widehat{A}$ for the vertices u^+ and v^-. If $a = (x^-, x^+)$, then x is a separating vertex for all paths from u to v in G. Otherwise, if $e = (x^+, y^-)$, then (x,y) is a separating arc for all paths from u to v in D and so both x and y are separating

vertices.

3.8 An Application: Computing All Dominator Trees

Let s be an arbitrary vertex of D. Recall the bridge decomposition $\mathcal{DT}$ of DT (Section 3.4.2), which is the forest obtained from DT after deleting the bridges of the flow graph D_s with start vertex s. As noted earlier, T_v is the tree in $\mathcal{D}$ that contains vertex v, and r_v denotes the root of T_v.

We define the *arc-dominator tree* $\widetilde{DT}$ of D with start vertex s, as the tree that results from DT after contracting all vertices in each tree T_v into its root r_v. For any vertex v and arc $a = (x, y)$, a is contained in all paths in D from s to v if and only if (r_x, r_y) is in the path from s to r_v in $\widetilde{DT}$. We denote by $\widetilde{d}(y)$ the parent of a vertex in $\widetilde{DT}$. (Both y and $\widetilde{d}(y)$ are roots in $\mathcal{DT}$.)

▶ **Theorem 3.22.** *All sources vertex- and arc-dominator trees can be computed from $D_R^{\rightsquigarrow 2a}$ in $\mathcal{O}(n^2)$ total time.*

Proof. To construct the arc-dominator tree $\widetilde{DT}$ with start vertex s, we look at the entries $D_R^{\rightsquigarrow 2a}[s, v]$, for all vertices v. We have the following cases: (a) If $D_R^{\rightsquigarrow 2a}[s, v] = \perp$ then v is not reachable from s and we set $r_v = \mathsf{undefined}$. (b) If $D_R^{\rightsquigarrow 2a}[s, v] = \top$ then we set $r_v = s$. (c) If $D_R^{\rightsquigarrow 2a}[s, v] = (x, y)$ then (x, y) is the last common arc in all paths from s to v. We mark y, set $r_v = y$, and temporarily assign $\widetilde{d}(y) = x$ (note that r_x may be unknown at this point). After we have processed the entries $D_R^{\rightsquigarrow 2a}[s, v]$ for all v, we make another pass over the marked vertices. Let y be a marked vertex, for which we temporarily assigned $\widetilde{d}(y) = x$. Then we set $\widetilde{d}(y) = r_x$. This completes the construction of $\widetilde{DT}$, which clearly takes $\mathcal{O}(n)$ time. By repeating this procedure for each vertex in V as a start vertex, we can compute all the arc-dominator trees, each rooted at a different vertex, in total $\mathcal{O}(n^2)$ time. The construction of DT is very similar, we apply the standard trick of splitting each vertex into an in- and out-vertex. ◀

Theorem 3.22 enables us to obtain the following results.

Reachability queries after the deletion of an arc or vertex. We preprocess each arc-dominator tree $\widetilde{DT}$ in $\mathcal{O}(n)$ so as to answer ancestor-descendant relations in constant time [50]. We also compute in $\mathcal{O}(n)$ time

the number of descendants in DT of every root r in $\mathcal{D}$. This allows us to answer various queries very efficiently:

- Given a pair of vertices s and t and an arc $a = (x, y)$, we can test if $D \setminus a$ contains a path from s to t in constant time. This is because a is contained in all paths from s to t in D if and only if the following conditions hold: a is a bridge of the flow graph D with start vertex s (i.e., $r_y = y$ and $\widetilde{d}(y) = r_x$) and y is an ancestor of r_t in $\widetilde{DT}$.
- Similarly, given a vertex s and an arc $a = (x, y)$, we can report in constant time how many vertices become unreachable from s if we delete a from D. If a is a bridge of the flow graph D with start vertex s, then this number is equal to the number of descendants of y in DT.

By computing all vertex-dominator trees of D, we can answer analogous queries for vertex-separators.

Computing junctions. A vertex s is a *junction* of vertices u and v in D, if D contains a path from s to u and a path from s to v that are vertex-disjoint (i.e., s is the only vertex in common in these paths). Yuster [56] gave a $\mathcal{O}(n^\omega)$ algorithm to compute a single junction for every pair of vertices in a DAG. By having all dominator trees of a digraph D, we can also answer the following queries.

- Given vertices s, u and v, test if s is a junction of u and v. This is true if and only if u and v are descendants of distinct children of s in DT. Hence, we can perform this test in constant time.
- Similarly, we can report all junctions of a given a pair of vertices in $\mathcal{O}(n)$ time. Note that two vertices may have n junctions (e.g., in a complete graph).

Computing critical nodes and critical edges. Let D be a directed graph. Define the *reachability function* $f(D)$ as the number of vertex pairs $\langle u, v\rangle$ such that u reaches v in D, i.e., there is a directed path from u to v. Here we consider how to compute the *most critical node* (resp., *most critical arc*) of D, which is the vertex v (resp., arc a) that minimizes $f(D \setminus v)$ (resp., $f(D \setminus a)$). This problem was considered by Boldi et al. [7] who provided an empirical study of the effectiveness of various heuristics. A related problem, where we wish to find the vertex v (resp., arc a) that minimizes the pairwise strong connectivity of $D \setminus v$ (resp., $D \setminus a$), i.e.,

$\sum_i \binom{|C_i|}{2}$, where C_i are the strongly connected components of $D \setminus v$ (resp., $D \setminus a$), can be solved in linear time [19, 37].

The naive solution to find the most critical node of D is to calculate the transitive closure matrix of $D \setminus v$, for all vertices v, and choose the vertex v that minimizes the number of nonzero elements. This takes $\mathcal{O}(n^{\omega+1})$ time. Similarly, we can compute the most critical arc in $\mathcal{O}(mn^{\omega})$ time. Here we provide faster algorithms that exploit the computation of all dominator trees. We let D_u and DT_u denote, respectively, the flow graph with start vertex u and its dominator tree. Also, we denote by $D_u(v)$ the subtree of DT_u rooted at vertex v.

Computing the most critical node. Observe that $f(D \setminus v) = \sum_u (|DT_u| - |D_u(v)|)$, since for each vertex u, the vertices that become unreachable from u after deleting v are exactly the descendants of v in D_u. Hence, we can process all dominator trees in $\mathcal{O}(n^2)$ time and compute $|D_u(v)|$ for all vertices u, v. Then, it is straightforward to compute $f(D \setminus v)$, for a single vertex v, in $\mathcal{O}(n)$ time. Thus, we obtain an algorithm that computes the most critical node of D in $\mathcal{O}(n^{\omega} \log n)$ total time.

Computing the most critical arc. Almost the same idea works for computing the most critical arc of D. Here, we observe that v becomes unreachable from u in $D \setminus a$ if and only if $a = (x, y)$ is a bridge of D_u and $v \in D_u(y)$. To exploit this observation, we store for each vertex y a list L_y of pairs $\langle u, x \rangle$ such that (x, y) is a bridge in D_u. Note that $\langle u, x \rangle \in L_y$ implies that x is the parent of y in DT_u. Thus, each list L_y has at most $n - 1$ pairs. Now, for each arc $a = (x, y)$, we have $f(D \setminus a) = \sum_{u:\langle u,x \rangle \in L_y} (|DT_u| - |D_u(y)|)$. So, it is straightforward to compute $f(D \setminus a)$, for a single arc a, in $\mathcal{O}(n)$ time. To compute $f(D \setminus a)$ for all arcs a, observe that is suffices to process only the pairs in all the L_y lists. Specifically, for each arc (x, y) we maintain a count $\mathit{unreach}(x, y)$, initially set to zero. When we process a pair $\langle u, x \rangle \in L_y$, we increment $\mathit{unreach}(x, y)$ by $|DT_u| - |D_u(y)|$. Clearly, after processing all pairs, the most critical arc is the one with maximum *unreach* value. Since there are $\mathcal{O}(n^2)$ pairs overall in all lists L_y, we obtain an algorithm that computes the most critical arc of G in $\mathcal{O}(n^{\omega} \log n)$ total time.

4

k-Reachability in DAGs

In this chapter, we study the generalization of the all-pairs reachability problem with a threshold k: For all vertex pairs $u, v \in V$, we want to distinguish whether there are 1, 2, ..., $k-1$, or at least k arc-disjoint paths from u to v.

The algorithms that we will present in this chapter will only work on acyclic graphs. On the positive side, instead of just reporting the size of the all-pairs max-flows, these algorithms will also produce witnesses of the min-cuts, if they are at most of size $k-1$.

4.1 Preliminaries

We start with some terminology and well-known results on graphs and cuts

Flows and cuts. We follow the notation used by Ford and Fulkerson [14]. Let $D = (V, A)$ be a digraph, where each arc a has a nonnegative capacity $c(a)$. For a pair of vertices s and t, an s-t flow of D is a function f on A such that $0 \leq f(a) \leq c(a)$, and for every vertex $v \neq s, t$ the incoming flow is equal to outgoing flow, i.e., $\sum_{(u,v)\in A} f(u,v) = \sum_{(v,u)\in A} f(v,u)$. The value of the flow is defined as $|f| = \sum_{(s,v)\in A} f(s,v) - \sum_{(v,s)\in A} f(v,s)$. We denote the existence of a path from s to t by $s \rightsquigarrow t$ and by $s \not\rightsquigarrow t$ the lack

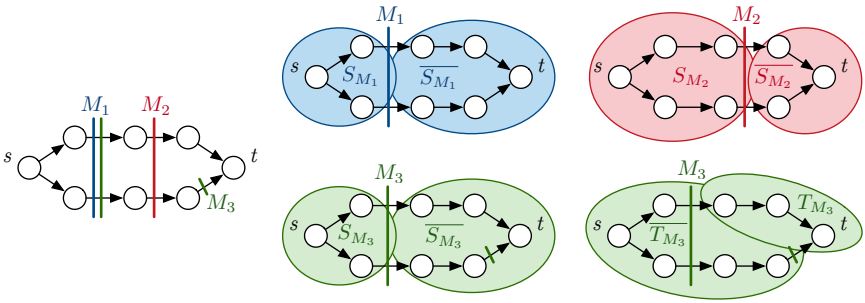


Figure 15 A digraph with three s-t-cuts M_1, M_2, M_3. While M_1 and M_2 are minimal, M_3 is not. Hence, the source side and target side differ only for M_3. This illustrates that the earlier and later orders might not be symmetric for non-minimal cuts. We have $M_3 < M_2$ yet $M_2 \not> M_3$ (and also $M_3 \leq M_2$ yet $M_2 \not\geq M_3$). Additionally, $M_1 \not< M_3$ yet $M_3 > M_1$ (yet both $M_1 \leq M_3$ and $M_3 \geq M_1$).

of such a path. Any set $M \subseteq A$ is an *s-t-cut* if $s \not\rightsquigarrow t$ in $D \setminus M$. M is a *minimal* s-t-cut if no proper subset of M is s-t-cut. For an s-t-cut M, we say that its *source side* is $S_M = \{x \mid s \rightsquigarrow x \text{ in } D \setminus M\}$ and its *target side* is $T_M = \{x \mid x \rightsquigarrow t \text{ in } D \setminus M\}$. We also refer to the source side and the target side as *s-reachable* and *t-reaching*, respectively and denote by $\overline{S_M} = V \setminus S_M$ and $\overline{T_M} = V \setminus T_M$. An s-t k-cut is a minimal cut of size k. A set $\mathcal{M}$ of s-t cuts of size at most k is called a set of s-t $\leq k$-cuts.

Order of cuts. An s-t cut M is *later* (respectively *earlier*) than an s-t cut M' if and only if $T_M \subseteq T_{M'}$ (resp. $S_M \subseteq S_{M'}$), and we denote it by $M \geq M'$ (resp. $M \leq M'$). Note that those relations are not necessarily complementary if the cuts are not minimal (see Figure 15 for an example). We make these inequalities strict (i.e., '$>$' or '$<$') whenever the inclusions are proper. We compare a cut M and an arc a by defining $a > M$ whenever both endpoints of a are in T_M. Additionally, $a \geq M$ includes the case where $a \in M$. Definitions of the relations '$\leq$' and '$<$' follow by symmetry. We refer to Figure 16 for illustrations.

This partial order of cuts also allows us to define cuts that are extremal with respect to all other s-t cuts in the following sense:

Definition 4.1 (s-t-latest cuts [32]). *An s-t cut is* s-t-latest *(resp. s-t-*earliest*) if and only if there is no later (resp. earlier) s-t cut of smaller or equal size.*

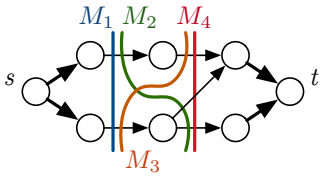


Figure 16 A digraph with several s-t cuts. Bold arcs represent parallel arcs which are too expensive to cut. M_1 is the earliest s-t min-cut and M_3 is the latest s-t min-cut. M_2 is later than M_1, but M_2 is not s-t-latest, as M_4 is later and not larger than M_2.

Informally speaking, a cut is s-t-latest if we would have to cut through more arcs whenever we would like to cut off fewer vertices. This naturally extends the definition of an s-t-latest *min*-cut as used by Ford and Fulkerson [14, Section 5]. The notion of latest cuts has first been introduced by Marx [32] (under the name of *important* cuts) in the context of fixed-parameter tractable algorithms for multi(way) cut problems. Since we need both earliest and latest cuts, we do not refer to latest cuts as important cuts. Additionally, we use the term *s-t-extremal* cuts to refer to the union of s-t-earliest and s-t-latest cuts. To avoid repetitions below, we state some results only for *latest* cuts. However, all of them naturally extend to earliest cuts.

▶ **Lemma 4.2 (Latest s-t min-cut [14, Theorem 5.5]).** *For any directed graph $G = (V, A)$, any maximum s-t flow f defines the same set of t-reaching vertices $T_{s,t}$ and thus defines an s-t cut $M = A \cap (\overline{T_{s,t}} \times T_{s,t})$, with $T_{s,t} = \{x \in V \mid \exists\ x\text{-}t \text{ path in residual graph of } G \text{ under flow } f\}$. For any s-t min-cut M', we have $M' \leq M$.*

Maximum flows are not necessarily unique, but Lemma 4.2 shows that the t-reaching cut M is.

▶ **Corollary 4.3.** *For any digraph G and vertices s and t, the latest s-t min-cut is unique.*

4.2 Overview

Our contribution here is twofold. We first prove some properties of the structure of the s-t-latest k-cuts and of the s-t-latest $\leq k$-cuts, which might be of independent interest. This gives us some crucial insights on the

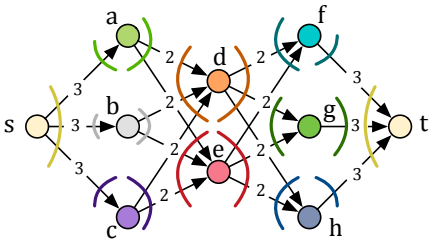


Figure 17 A digraph where each arc appears in at least one s-v or one v-t min-cut. The numbers on the arcs denote the number of parallel arcs. Note that neither of the two s-t min-cuts of size 9 (marked in yellow) are contained within the union of any two s-v or v-t min-cuts. Thus, finding all those min-cuts and trying to combine them in pairs in a divide-and-conquer-style approach is not sufficient to find an s-t min-cut.

structure of the cuts, and allows us to develop an algorithmic framework which is used to solve *all-pairs k-reachability*. As a second contribution, we exploit our new algorithmic framework in two different ways, leading to two new algorithms which run in $\mathcal{O}(mn^{1+o(1)})$ time for $k = o(\sqrt{\log n})$ and in $\mathcal{O}(n^{\omega+o(1)})$ time for $k = o(\log \log n)$.

Let $D = (V, A)$ be a DAG. Consider some arbitrary pair of vertices s and t, and any s-t-cut M. For every intermediate vertex v, M must be either a s-v-cut, or a v-t-cut. The knowledge of all s-v and all v-t min-cuts does not allow us to convey enough information for computing an s-t min-cut of size at most k quickly, as illustrated in Figure 17. However, we are able to compute an s-t min-cut by processing all the *s-v-earliest* cuts and all the *v-t-latest* cuts, of size at most k. We build our approach around this insight. We note that the characterization that we develop is particularly useful, as it has been shown that the number of all earliest/latest u-v $\leq k$-cuts can be upper bounded by $2^{\mathcal{O}(k)}$, independently of the size of the graph.

For a more precise formulation on how to recover a min-cut (or extremal $\leq k$-cuts) from cuts to and from intermediate vertices, consider the following. Let A_1, A_2 be an *arc split*, that is a partition of the arc set A with the property that any path in D consists of a (possibly empty) sequence of arcs from A_1 followed by a (possibly empty) sequence of arcs from A_2 (see Definition 3.1). Assume that for each vertex v we know all the s-v-earliest $\leq k$-cuts in $D_1 = (V, A_1)$ and all the v-t-latest $\leq k$-cuts in

$D_2 = (V, A_2)$. We show that a set of arcs M that contains as a subset one s-v-earliest $\leq k$-cut in D_1, or one v-t-latest $\leq k$-cut in D_2 for every v, is a s-t-cut. Moreover, we show that all the s-t-cuts of arcs with the above property include all the s-t-latest $\leq k$-cuts. Hence, in order to identify all s-t-latest $\leq k$ cuts, it is sufficient to identify all sets M with that property. We next describe how we use these structural properties to compute all s-t-extremal $\leq k$-cuts.

We formulate the following combinatorial problem over families of sets, which is independent of graphs and cuts, that we can use to compute all s-t-extremal $\leq k$-cuts. The input to our problem are c families of sets $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_c$, where each family $\mathcal{F}_i$ consists of at most K sets, and each set $F \in \mathcal{F}_i$ contains at most k elements from a universe U. The goal is to compute all minimal subsets $F^* \subset U, |F^*| \leq k$, for which there exists a set $F \in \mathcal{F}_i$ such that $F \subseteq F^*$, for all $1 \leq i \leq c$. We refer to this problem as Witness Superset. To create an instance (s, t, A_1, A_2) of the Witness Superset problem, we set $c = |V|$ and $\mathcal{F}_v$ to be all s-v-earliest $\leq k$-cuts in D_1 and all v-t-latest $\leq k$-cuts in D_2. Informally speaking, the solution to the instance (s, t, A_1, A_2) of the Witness Superset problem picks all sets of arcs that cover at least one earliest or one latest cut for every vertex. In a post-processing step, we filter the solution to the Witness Superset problem on the instance (s, t, A_1, A_2) in order to extract all the s-t-latest $\leq k$-cuts. We follow an analogous process to compute all the s-t-earliest $\leq k$-cuts.

Algorithmic framework. Next, we define a common algorithmic framework for solving all-pairs k-reachability, as follows. We pick a partition of the vertices V_1, V_2, such that there is no arc in $V_2 \times V_1$. Such a partition can be trivially computed from a topological order of the input DAG. Let $A_1, A_2, A_{1,2}$ be the sets of arcs in $D[V_1]$, in $D[V_2]$, and in $A_{1,2} = A \cap (V_1 \times V_2)$.

- First, we recursively solve the problem in $D[V_1]$ and in $D[V_2]$. The recursion returns without doing any work whenever the graph is a singleton vertex.
- Second, for each pair of vertices (s, t), such that $s \in V_1$ has an outgoing arc from $A_{1,2}$ and $t \in V_2$, we solve the instance $(s, t, A_{1,2}, A_2)$ of Witness Superset. Notice that the only non-empty earliest cuts in $(V, A_{1,2})$ for the pair (x, y) are the arcs $(x, y) \in A_{1,2}$.

- Finally, for each pair of vertices (s,t), such that $s \in V_1, t \in V_2$, we solve the instance $(s,t,A_1,A_{1,2} \cup A_2)$ of Witness Superset.

The Witness Superset problem can be solved naively as follows. Let $\mathcal{F}_v$ be the set of all s-v-earliest $\leq k$-cuts and all v-t-latest $\leq k$-cuts. Assume we have $\mathcal{F}_{v_1}, \mathcal{F}_{v_2}, \ldots, \mathcal{F}_{v_c}$, for all vertices $v_1, v_2, \ldots, v_c$ that are both reachable from s in $(V, A_{1,2})$ and that reach t in (V, A_2). Each of these sets contains $2^{\mathcal{O}(k)}$ cuts. We can identify all sets M of arcs that contain at least one cut from each $\mathcal{F}_i$, in time $\mathcal{O}(k \cdot (2^{\mathcal{O}(k)})^c)$. This yields an algorithm with super-polynomial running time. However, we speed up this naive procedure by applying some judicious pruning, achieving a better running time of $\mathcal{O}(c \cdot 2^{\mathcal{O}(k^2)} \cdot \mathrm{poly}(k))$, which is polynomial for $k = o(\sqrt{\log n})$. In the following, we sketch the two algorithms that we develop for solving the k-reachability problem efficiently.

Iterative division. For the first algorithm, we process the vertices in reverse topological order. When processing a vertex v, we define $V_1 = \{v\}$ and V_2 to be the set of vertices that appear after v in the topological order. Notice that V_1 has a trivial structure, and we already know all s-t-latest $\leq k$-cuts in $D[V_2]$. In this case, we present an algorithm for solving the instance $(v,t,A_{1,2},A_2)$ of the Witness Superset problem in time $\mathcal{O}(2^{\mathcal{O}(k^2)} \cdot c \cdot \mathrm{poly}(k))$, where $c = |A_{1,2}|$ is the number of arcs leaving v. We invoke this algorithm for each v-w pair such that $w \in V_2$. For $k = o(\sqrt{\log n})$ this gives an algorithm that runs in time $\mathcal{O}(\deg^+(v) \cdot n^{1+o(1)})$ for processing v, and $\mathcal{O}(mn^{1+o(1)})$ in total.

Recursive division. For the second algorithm, we recursively partition the set of vertices evenly into sets V_1 and V_2 at each level of the recursion. We first recursively solve the problem in $D[V_1]$ and in $D[V_2]$. Second, we solve the instances $(s,t,A_{1,2},A_2)$ and $(s,t,A_1,A_{1,2} \cup A_2)$ of Witness Superset for all pairs of vertices from $V_1 \times V_2$. Notice that the number of vertices that are both reachable from s in (V,A_1) and reach t in $(V, A_{1,2} \cup A_2)$ can be as high as $\mathcal{O}(n)$. This implies that even constructing all $\Theta(n^2)$ instances of the Witness Superset problem, for all s,t, takes $\Omega(n^3)$ time. To overcome this barrier, we take advantage of the power of fast matrix multiplications by applying it into suitably defined matrices of binary codes (codewords). At a very high-level, this approach was used by Fischer and Meyer [13] in their $\mathcal{O}(n^\omega)$ time algorithm for transitive closure in DAGs – there the binary codes where of size 1 indicating whether there exists an

arc between two vertices.

Algebraic framework. In order to use coordinate-wise boolean matrix multiplication with the entries of the matrices being codewords we first encode all s-t-earliest and all s-t-latest $\leq k$-cuts using binary codes. The bitwise boolean multiplication of such matrices with binary codes in its entries allows a serial combination of both s-v cuts and v-t cuts based on AND operations, and thus allows us to construct a solution based on the OR operation of pairwise AND operations. We show that *superimposed codes* are suitable in our case, i.e., binary codes where sets are represented as bitwise-OR of codewords of objects, and small sets are guaranteed to be encoded uniquely. Superimposed codes provide a unique representation for sets of up to k elements from a universe of size $\text{poly}(n)$ with codewords of length $\text{poly}(k \log n)$. In this setting, the union of sets translates naturally to bitwise-OR of their codewords.

Tensor product of codes. To achieve our bounds, we compose several identical superimposed codes into a new binary code, so that encoding *set families* with it enables us to solve the corresponding instances of Witness Superset. Our composition has the cost of an exponential increase in the length of the code. Let $\mathcal{F} = F_1, \ldots, F_c$ be the set family that we wish to encode, and let $S_1, \ldots, S_c$ be their superimposed codes in the form of vectors. We construct a c-dimensional array M where $M[i_1, \ldots, i_c] = 1$ iff $S_j[i_j] = 1$, for each $1 \leq j \leq c$. In other words, the resulting code is the tensor product of all superimposed codes. This construction creates enough redundancy so that enough information on the structure of the set families is preserved. Furthermore, we show how we can extract the encoded information from the bitwise-OR of several codewords. The resulting code is of length $\mathcal{O}((k \log n)^{\mathcal{O}(K)})$, where K is the upperbound on the allowed number of sets in each encoded set family. In our case we have $K \approx 4^k$, which results to only a logarithmic dependency on n at the price of a doubly-exponential dependency on k, thus making the problem tractable for small values of k.

From slices to Witness Superset. Finally, we show how the Witness Superset can be solved using tensor product of superimposed codes. Consider the notion of cutting the code of dimension K with an axis-parallel hyperplane of dimension $K - 1$. We call this resulting shorter codeword

a *slice* of the original codeword. A slice of a tensor product is a tensor product of one dimension less, or an empty set, and a slice of a bitwise-OR of tensor products is as well a bitwise-OR of tensor products (of one dimension less). Thus, taking a slice of the bitwise-OR of the encoding of families of sets is equivalent to removing a particular set from some families and to dropping some other families completely and then encoding these remaining, reduced families. Thus, we can design a non-deterministic algorithm, which at each step of the recursion picks k slices, one slice for each element of the solution we want to output, and then recurses on the bitwise-OR of those slices, reducing the dimension by one in the process. This is always possible, since each element that belongs to a particular solution of Witness Superset satisfies one of the following: it either has a witnessing slice and thus it is preserved in the solution to the recursive call; or it is dense enough in the input so that it is a member of each solution and we can detect this situation from scanning the diagonal of the input codeword. This described nondeterministic approach is then made deterministic by simply considering every possible choice of k slices at each of the K steps of the recursion. This does not increase the complexity of the decoding procedure substantially, since $\mathcal{O}(((K \cdot \mathrm{poly}(k \log n))^k)^K)$ for $K \approx 4^k$ is still only doubly-exponential in k.

4.3 Structure of Cuts

In this section, we study the dependence of the latest s-t cuts on the s-v cuts and the v-t cuts, for all vertices $v \notin \{s, t\}$. None of the results contained in this section rely on the input graph D being acyclic.

We begin by introducing some notation for sets of extremal cuts and their transitive order. Then, building on the uniqueness of the latest min-cut (Corollary 4.3), we constructively define an operation which we call *arc replacement* in an s-t-latest cut. We refer to Figures 18 and 19 for illustrations.

Transitive reduction. By $\mathcal{F}_{s,t}$ we denote the set of s-t-latest cuts, by $\mathcal{F}^k_{s,t}$ the set of s-t-latest k-cuts. Sets of earliest cuts are denoted by $\mathcal{E}_{s,t}$ and $\mathcal{E}^k_{s,t}$ respectively. We also denote $\mathcal{F}^{\leq k}_{s,t} = \bigcup_{i=1}^{k} \mathcal{F}^i_{s,t}$. Since '$>$', the partial order on cuts, is a transitive relation, we can consider its transitive reduction. We say that $M' \in \mathcal{F}_{s,t}$ is *immediately later* than $M \in \mathcal{F}_{s,t}$, if $M' > M$ and there is no $M'' \in \mathcal{F}_{s,t}$ such that $M' > M'' > M$.

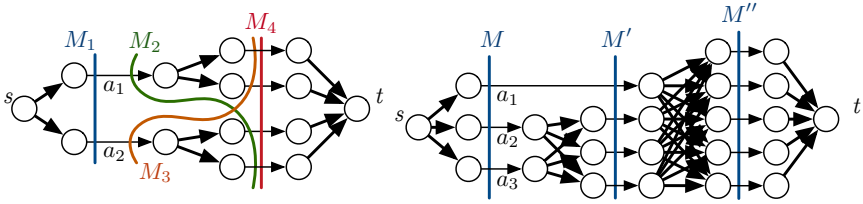


Figure 18 Examples of digraphs with several s-t cuts. Bold arcs represent parallel arcs which would be too expensive to cut. (left) We have $M_2 > M_1$, $M_3 > M_1$, $M_4 > M_2$ and $M_4 > M_3$ and all four cuts are s-t-latest. M_2 and M_3 are incomparable, neither of them is later than the other. M_2 is the arc replacement of M_1 and a_2, M_3 is the arc replacement of M_1 and a_1. M_4 is both the arc replacement of M_2 and a_1 and of M_3 and a_2. (right) The cut $M = \{a_1, a_2, a_3\}$ is the only s-t min-cut. M' is the only s-t-latest cut of size 4 and M'' is the only s-t-latest cut of size 5. Note that we have $M'' > M' > M$. M'' is the arc replacement of M and a_1, while M' is the arc replacement of M and a_2 and also of M and a_3.

Definition 4.4 (Arc replacement). *Given $M \in \mathcal{F}_{s,t}$ and $a = (u, v) \in M$, let $D' = (V', A')$ be a copy of D, where all vertices in $S_M \cup \{v\}$ are contracted to vertex s. We call the unique latest s-t min-cut M^* in D' the* arc replacement of M and a in D *(or say that it does not exist if s and t got contracted into the same vertex in D' whenever $v = t$).*

Note that the arcs A' in D' correspond to a subset of the arcs A in D as we think of the contraction as a relabeling of some of the endpoints without changing any identifiers. We note that a similar operation with respect to latest min-cuts was used by Baswana, Choudhary, and Roditty [5], but in a different way. Given a digraph D and two vertices s, t, Baswana et al. [5] use an operation to compute a set A_t of incoming arcs to t with the following property: Let D' be the subgraph of D where the only arcs entering t are the arcs in A_t. Then, there exist k arc-disjoint paths from s to t in D iff there exist k arc-disjoint paths from s to t in D'. Here, we use the arc replacement operation to relate all s-t-latest cuts, as we show next.

▶ **Lemma 4.5.** *For any s-t-latest cuts M and M', if M' is immediately later than M then M' is the arc replacement of M and arc a, where a can be any arc in $M \setminus M'$.*

Proof. For a given s-t-latest cut M, take any arc $a^* \in M \setminus M'$ and build the corresponding arc replacement graph D'. Let M^* be the latest s-t

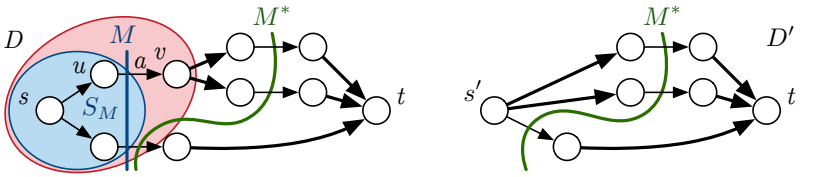


Figure 19 (left) The original graph D, with bold arcs representing parallel arcs and the s-t-latest (min-)cut M. To replace $a = (u, v) \in M$ in D, the source side S_M (blue set) together v (red set) gets contracted into a new source vertex s'. (right) The graph D' after the contraction. The latest s'-t min-cut M^* in D' corresponds to the arc replacement of M and a in D.

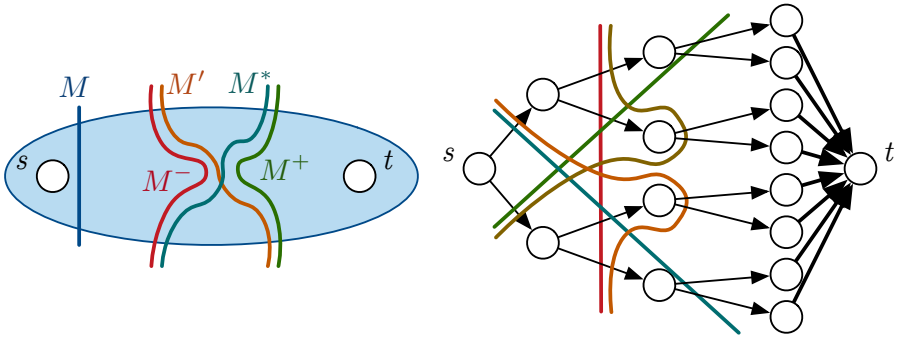


Figure 20 (left) The cuts involved in the proof of Lemma 4.5. If M' and M^* differ, M^+ would be both later and not larger than M', hence M' could not be s-t-latest. (right) A tight example for Lemma 4.6 (bold arcs represent parallel arcs): A digraph with all $C_3 = 5$ many s-t-latest 4-cuts.

min-cut in D' (which is unique by Corollary 4.3). As we have $a' \geq M$ for all $a' \in A'$, we get $M' > M$ in D. Also, $M^* \in \mathcal{F}_{s,t}$ (all cuts later than M^* have the same cardinality in D and in D' and hence cannot be smaller). The cut M' has size $|M'|$ in D' as all arcs $a' \in M'$ satisfy $a' \geq M$ (as $M' \geq M$) and $a' \neq a^*$ and thus are not contracted in D'. By minimality of M^*, we get $|M^*| \leq |M'|$.

For the sake of reaching a contradiction, let us assume that $M^* \neq M'$. M^* and M' can not be comparable: $M' > M^*$ would contradict M' being immediately later than M, and $M^* > M'$ would contradict M' being in $\mathcal{F}_{s,t}$. We now define two auxiliary cuts (see Figure 20 (left) for an illustration):

$$M^+ = \{a \in M' \cup M^* \mid a \geq M' \text{ and } a \geq M^*\}$$
$$M^- = \{a \in M' \cup M^* \mid M' \geq a \text{ and } M^* \geq a\}$$

As M^+ corresponds to $(X, \overline{X})$ with $\overline{X} = \overline{T_{M'}} \cap \overline{T_{M^*}}$ and M^- to $(Y, \overline{Y})$ with $\overline{Y} = \overline{T_{M'}} \cup \overline{T_{M^*}}$, these two arc sets really correspond to minimal s-t-cuts. We have $M^+ \geq M', M^* \geq M^-$ and $|M^-| + |M^+| = |M'| + |M^*|$, which combined with $|M^*| \leq |M^+|, |M^-|, |M'|$ gives $|M^+| \leq |M'|$. However, this contradicts $M' \in \mathcal{F}_{s,t}$ as M^+ would both be later and not larger than M'. ◀

Note that the reverse direction does not hold: Some arc replacements result in cuts that are not immediately later, as illustrated by M'' in Figure 18 (right). The following lemma extends the uniqueness of min-cuts (Lemma 4.3) to bounding the number of s-t-latest cuts in general.

▶ **Lemma 4.6 (Theorem 8.11 in [9]).** *For any $k \geq 1$ there are at most $C_{k-1} = \frac{1}{k}\binom{2k-2}{k-1}$ s-t-latest k-cuts, and at most 4^k s-t-latest $\leq k$-cuts.*

Note that the bound of C_{k-1} is tight: consider a full binary tree with arcs directed away from the root s, and an extra vertex t with incoming arcs from every leaf of the tree. If the tree is large enough, any s-t k-cut is latest and corresponds to a binary subtree with k leaves. We refer to Figure 20 (right) for an example.

We recall the definition of an arc split from Definition 3.1. In an acyclic graph, an arc split is easily obtained from a topological order, e.g., by partitioning V into a prefix of the order V_1 and a suffix V_2, and assigning $A \cap (V_1 \times V_1)$ to A_1, $A \cap (V_2 \times V_2)$ to A_2 and arcs from $A \cap (V_1 \times V_2)$ arbitrarily.

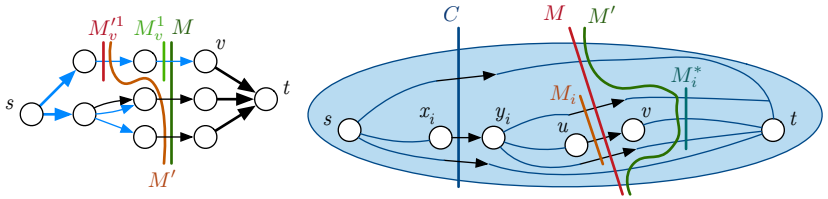


Figure 21 (left) Illustration for the argument in Theorem 4.8. For the arc split A_1, A_2, the arcs in A_1 are shown in blue, the arcs in A_2 in black. Bold arcs represent parallel arcs that cannot be cut. M is an s-t min-cut, but it does not satisfy Property 4.7 for vertex v. Replacing M_v^1 by $M_v'^1$ gives M' which does satisfy Property 4.7. (right) Illustration for the argument in Theorem 4.10. C is any cut that is not later than any cut in $\mathcal{F}_{s,t}$. For any s-t-latest cut M without Property 4.9, we can find a subset M_i, that is a y_i-t cut and that can be replaced by some later y_i-t cut M_i^*. This gives us a later s-t cut M' of equal or smaller size, contradicting M being s-t latest.

Property 4.7 (Split-covering sets). *We say that a set $M \subseteq A$ is* split-covering *with respect to arc split A_1, A_2 and vertices s,t iff for any $v \in V$, there exists $M_v \subseteq M$ such that at least one of the following conditions holds*

- *$s \neq v \neq t$ and v is either unreachable from s in (V, A_1) or v does not reach t in (V, A_2).*
- *M_v is an s-v-earliest cut in (V, A_1),*
- *M_v is a v-t-latest cut in (V, A_2).*

▶ **Theorem 4.8.** *Fix an arbitrary arc split A_1, A_2 and vertices s,t. Any split-covering set (w.r.t. A_1, A_2 and s,t) is an s-t-cut in D. Moreover, there exists an s-t min-cut that is split-covering (w.r.t. A_1, A_2 and s,t).*

Proof. To argue that any set M that is split-covering is an s-t-cut in D, consider any s-t path P in D. We only consider vertices v that are both reachable from s in (V, A_1) (or $v = s$) and reach t in (V, A_2) (or $v = t$). Let v be the last vertex in P that is reached using arcs in A_1 ($v = s$ if P has no edges from A_1). Now the set $M_v \subseteq M$ is either an s-v-earliest cut (which implies $s \neq v$) and then M_v intersects P before v or M_v is a v-t-latest cut (which implies $v \neq t$) and then M_v intersects P after v.

To see that there is an s-t min-cut that is split-covering, take any s-t min-cut M. Intuitively speaking, we now argue that we can incrementally push its arcs in A_1 towards s and its arcs in A_2 towards t until all its sub-

cuts M_v become earliest/latest. Throughout these changes M is always an s-t cut and never increases in size. We refer to Figure 21 (left) for an illustration.

For any $v \in V$, we define M_v^1 as a minimum subset of M such that M_v^1 is an s-v cut in (V, A_1) and M_v^2 as a minimum subset of M such that M_v^2 is a v-t cut in (V, A_2). Note that while only one of the two sets might exist, at least one does, otherwise M would not be an s-t cut. Assume w.l.o.g. that M_v^1 exists. If M_v^1 is an earliest s-v cut in (V, A_1), vertex v already satisfies Property 4.7 and no change is required for v. Otherwise there is another cut ${M'}_v^1$ that is earlier than M_v^1 and satisfies $|{M'}_v^1| \leq |M_v^1|$. Now we define a new set $M' = (M \setminus M_v^1) \cup {M'}_v^1$ which satisfies $|M'| \leq |M|$.

We claim that M' is still an s-t min-cut in D. For the sake of reaching a contradiction, assume otherwise. Any s-t path P that avoids M' must use an arc a in M_v^1. Let P' denote the prefix of P from s to a. As $a \in A_1$, we have $P' \subseteq A_1$. By definition of M_v^1, there is a path Q from a to v in A_1 (otherwise a would not need to be in M_v^1). As ${M'}_v^1 \leq M_v^1$, we can pick Q such that it avoids ${M'}_v^1$. Thus concatenating P' and Q gives an s-v path in $A_1 \setminus {M'}_v^1$, a contradiction. Hence, M' is also an s-t cut and as $|M'| \leq |M|$ it is also an s-t min-cut. Applying this argument repeatedly for all v without an earliest/latest M_v, will end with an s-t min-cut that satisfies Property 4.7 after finitely many repetitions. ◀

Property 4.9 (Late-covering sets). *Let $C = \{(x_1, y_1), \ldots, (x_j, y_j)\}$ be any s-t cut such that $M' \geq C$ for all $M' \in \mathcal{F}_{s,t}$. We say that a set $M \subseteq A$ is* late-covering *w.r.t. C if for each $1 \leq i \leq j$, at least one of the following conditions holds*

- $(x_i, y_i) \in M$,
- *there is $M_i \subseteq M$ such that M_i is a y_i-t-latest cut.*

In order to show that there always exists a set C satisfying Property 4.9, consider the set $C = \{(s, v) : v \text{ reaches } t \text{ in } G\}$ which is an s-t cut where for each s-t latest cut M it holds $M \geq C$.

▶ **Theorem 4.10.** *Fix s-t cut C as in Property 4.9. Any late-covering set w.r.t. C is an s-t-cut. Every s-t-latest cut is late covering w.r.t. C.*

Proof. To argue that any late-covering set M is an s-t-cut, consider any s-t path P in D. As C is an s-t cut, P must use some arc (x_i, y_i) in C. Since M is late-covering, either (x_i, y_i) or some y_i-t cut is part of M, hence

M intersects P either at (x_i, y_i) or in some later arc. Therefore, P is not in $D \setminus M$, and since P was arbitrary, M is an s-t-cut in D.

To see that any s-t-latest cut is late-covering, assume, for the sake of reaching a contradiction, that M is an s-t-latest cut that is not late-covering. As $M \neq C$ (C is trivially late-covering) and M is s-t-latest, we have $C < M$. Hence $S_M \supseteq S_C$ and so $x_i \in S_M$ for all i. For any $(x_i, y_i) \notin M$, M must contain some y_i-t cut M_i (to ensure that M is an s-t cut). As M is not late-covering, there exists i such that M_i can not be chosen to be y_i-t-latest. So for this M_i, there is another y_i-t cut M_i^* of size $|M_i^*| \leq |M_i|$ that is later than M_i. We refer to Figure 21 (right) for an illustration.

We now argue that $M' = (M \setminus M_i) \cup M_i^*$ is an s-t cut that is later and not larger than M, which contradicts M being s-t-latest. Clearly, $|M'| \leq |M|$ as $|M_i^*| \leq |M_i|$.

We first argue that M' is an s-t cut. Take any s-t path P that avoids M'. P must use an arc a in M_i. Let P' denote the suffix of P from a to t. As P' avoids M', P' is in $D \setminus M_i^*$. As $M_i^* > M_i$, there is a y_i-a path Q in $D \setminus M_i^*$. Concatenating Q and P' gives a y_i-t path in $D \setminus M_i^*$, hence P cannot exist.

It remains to argue that M' is later than M, so $T_{M'} \subset T_M$. Let us first argue that $T_{M'} \subseteq T_M$ by considering any $v \in T_{M'} \setminus T_M$ and reaching a contradiction. Any v-t path P in $D \setminus M'$ must contain an arc $a \in M$ (as $v \notin T_M$) and we have $a \in M_i \setminus M_i^*$ (as $M_i \setminus M_i^*$ is where M and M' differ). Let P' be the suffix of P from a to t and let Q be any y_i-a path in $G \setminus M_i^*$ (exists as $a \in M_i$ and $M_i^* > M_i$). The concatenation of Q and P' would form a y_i-t path in $D \setminus M_i^*$, a contradication. Finally, $T_{M'} \neq T_M$, as for any $(u, v) \in M_i \setminus M_i^*$, we have $v \in T_M$ (otherwise (u, v) would make M non-minimal) and $v \notin T_{M'}$ (otherwise there would be a y_i-t path in $D \setminus M_i^*$). Hence $M' > M$ and $|M'| \leq |M|$, so M is not s-t-latest. ◀

▶ **Corollary 4.11.** *In Theorem 4.8, if s-t min-cuts are of size at most k, then there is an s-t min-cut that is split-covering in a way that every M_v is either an s-v-earliest $\leq k$-cut or a v-t-latest $\leq k$-cut. In Theorem 4.10, every s-t-latest $\leq k$-cut is late covering in a way that every M_i is a y_i-t-latest $\leq k$-cut.*

4.4 Deterministic Algorithms with Witnesses

4.4.1 All-pairs k-reachability for $k = o(\sqrt{\log n})$

We now develop a first algorithm for computing small cuts, which considers the vertices of the DAG one by one. A first step is to consider a problem highlighted in Theorem 4.8, namely how to pick a set of arcs that covers at least one earliest or one latest cut for every vertex. We formalize this problem, independently of graphs and cuts, as follows:

Problem 4.12 (Witness Superset)**.** *Given a collection of c set families $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_c$, where each $\mathcal{F}_i$ is of size at most K, and each member of $\mathcal{F}_i$ is a subset of size at most k of some universe U, find all sets $W \subseteq U$ such that:*

- [cover] *for all i, there is $W_i \subseteq W$ such that $W_i \in \mathcal{F}_i$,*
- [size] *$|W| \leq k$,*
- [minimal] *no proper subset of W satisfies the [cover] condition.*

A naive solution to the Witness Superset problem is to iterate through all K^c possible unions of sets, one from each $\mathcal{F}_i$. However, using pruning as soon as the [size] constraint is violated, we can achieve a linear dependency on c while keeping the exponential dependency on k.

▶ **Lemma 4.13.** *Algorithm 23 solves the* Witness Superset *problem in time $\mathcal{O}(K^{k+1} \cdot c \cdot poly(k))$ and outputs a list of $\mathcal{O}(K^k)$ sets.*

Proof. Whenever the function SingleFamilyWitness is called on input i, we can inductively argue that the candidate set S is guaranteed to cover a set in each of the first $i-1$ families. For the i-th family $\mathcal{F}_i$, SingleFamilyWitness adds new elements to S only if necessary. If S already covers some $F \in \mathcal{F}_i$, S remains unchanged. Otherwise, we try all $F \in \mathcal{F}$ whose addition to S do not break the [size] constraint separately by adding them to S and evaluating recursively.

The correctness of Algorithm 23 follows from the fact that this search skips only over those of the K^c possible solutions that are either larger than k or are not minimal, so those violating conditions [size] and [minimal] of the Witness Superset problem.

To analyze the running time, we analyze the shape of the call tree T of SingleFamilyWitness. Any root-to-leaf path in T has length c but

Algorithm 23 Solving Witness Superset by recursion with pruning

```
1: procedure SingleFamilyWitness(i, S)
2:     if i = c + 1 then                          ▷ Found a potential witness S
3:         if no proper subset of S satisfies the [cover] condition then
4:             output S
5:         end if
6:         return
7:     end if
8:     if ∃F ∈ F_i : F ⊆ S then                   ▷ F_i is already covered
9:         SingleFamilyWitness(i + 1, S)
10:    else
11:        for F ∈ F_i do                         ▷ Try adding a new set to cover F_i
12:            if |S ∪ F| ≤ k then
13:                SingleFamilyWitness(i + 1, S ∪ F)
14:            end if
15:        end for
16:    end if
17: end procedure
18: procedure WitnessSuperset(F = {F_1, F_2, ..., F_c})
19:    SingleFamilyWitness(1, ∅)
20: end procedure
```

only visits at most k branching nodes, as each branching node increases the size of S by at least one. Also, each branching node has out-degree at most K. Hence T is a tree of depth c with at most K^k leaves, which establishes our output size and implies $|T| \leq K^k \cdot c$. The amount of work required to manipulate S in each step is in $\mathcal{O}(\text{poly}(k))$. Note that the final check before outputting S can be done in $\mathcal{O}(K \cdot c \cdot \text{poly}(k))$ by just checking condition [cover] of the Witness Superset problem for $S \setminus \{s\}$ for every $s \in S$ explicitly. Combining $|T| \cdot \mathcal{O}(K \cdot \text{poly}(k))$ with $K^k \cdot \mathcal{O}(K \cdot c \cdot \text{poly}(k))$ gives our running time. ◀

▶ **Theorem 4.14.** *All latest cuts of size at most k for all pairs of vertices of a DAG can be found in $\mathcal{O}(2^{\mathcal{O}(k^2)} \cdot mn)$ total time.*

Proof. We perform dynamic programming by repeatedly combining families of latest cuts while iterating through the vertices in a reverse topological order of D. Let this order be denoted by $v_n, v_{n-1}, \ldots, v_1$. When

processing vertex v_i in this order, we compute all small cuts originating from v_i, so all v_i-v_j-latest $\leq k$-cuts for all $i \leq j$. Note that for any $i > j$ all v_i-v_j cuts are trivial since D is a DAG. To find the non-trivial cuts, consider $D' = (V', A')$, the subgraph of D induced by vertices $V' = \{v_i, v_{i+1}, \ldots, v_n\}$. Since D is a DAG, cuts between those vertices are preserved in D'. Consider the arc split of D' with A_1 being all arcs leaving v_i and $A_2 = A' \setminus A_1$.

For every j such that $i < j \leq n$, we build an instance $\mathcal{I}_{i,j}$ of the Witness Superset problem to find all v_i-v_j-latest $\leq k$-cuts. We set c equal to the out-degree of v_i and let $N^+(v_i) = \{v_{i_1}, \ldots, v_{i_c}\}$ denote all the heads of the arcs leaving v_i. Then, for every x such that $1 \leq x \leq c$, the set family $\mathcal{F}_x$ is composed of all the cuts in $\mathcal{F}^{\leq k}_{v_{i_x}, v_j}$ and the singleton $\{(v_i, v_{i_x})\}$.

Note that $\{(v_i, v_{i_x})\}$ is the only earliest v_i-v_{i_x} cut in (V', A_1). Furthermore, for all other vertices $v \in V' \setminus N^+(v_i)$, the empty set is the only earliest v_i-v cut in (V', A_1). Since all arcs leaving v_i are in A_1, the empty set is the only v_i-t-latest cut in (V', A_2). Hence the families $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_c$ contain all earliest cuts in (V', A_1) and all latest cuts in (V', A_2) for all of V'. Therefore, we can apply Theorem 4.8 and any solution to $\mathcal{I}_{i,j}$ corresponds to a v_i-v_j-cut with Property 4.7.

As $(S_{A_1}, \overline{S_{A_1}}) = (\{v_i\}, V' \setminus (\{v_i\}))$ is the earliest possible v_i-v_j cut, we can apply Theorem 4.10 and Corollary 4.11 as well, using $C = (S_{A_1}, \overline{S_{A_1}})$. Notice that by Theorem 4.10 every v_i-v_j-latest cut is late covering w.r.t. C, and as the solution to the Witness Superset problem finds all late covering cuts we know that all v_i-v_j-latest $\leq k$-cuts appear among the solutions to $\mathcal{I}_{i,j}$.

Since all solutions to $\mathcal{I}_{i,j}$ are guaranteed to be minimal, all non-minimal v_i-v_j-cuts with properties 4.7 and/or 4.9 get filtered out and only the minimal v_i-v_j cuts remain. But some of the v_i-v_j cuts among the solutions to $\mathcal{I}_{i,j}$ might not be v_i-v_j-*latest*.

To enumerate the solutions to $\mathcal{I}_{i,j}$ we call Algorithm 23, which takes $\mathcal{O}(2^{\mathcal{O}(k^2)} \cdot c)$ time, by Lemma 4.13 and the $K \in 2^{\mathcal{O}(k)}$ bound on the number of latest $\leq k$-cuts from Lemma 4.6. This time bound, summed over all source vertices and all target vertices gives the claimed total runtime $\mathcal{O}(2^{\mathcal{O}(k^2)} \cdot mn)$.

To complete the proof, we need to argue that we can filter the solutions to $\mathcal{I}_{i,j}$ resulting only in the v_i-v_j-latest cuts, without additionally changing the time complexity. Let $\mathcal{M}$ be the output of Algorithm 23, so all the v_i-v_j

cuts satisfy Property 4.9. We show inductively that we can compute the later-relation for the cuts within $\mathcal{F}^{\leq k}_{v_i,v_j} \subseteq \mathcal{M}$ while doing the filtering.

We explicitly compute the relative order for all the v_i-v_j cuts in $\mathcal{M}$. That is, we test all pairs $M, M' \in \mathcal{M}$, for the following relation:

$$\begin{aligned} M' \geq M \Leftrightarrow \forall x \in [c], \forall M_x, M'_x \in \mathcal{F}_x : \\ M_x \subseteq M \text{ and } M'_x \subseteq M' \text{ implies } M'_x \geq M_x. \end{aligned} \tag{5}$$

For this order to be well-defined, we need the later-relation $M'_x \geq M_x$ to be defined for all the sets within $\mathcal{F}_x$. This is not immediate since $\mathcal{F}_x$ also contains the singleton set $\{(v_i, v_{i_x})\}$ which might not disconnect v_i from v_j. To fix this, we just define that $\{(v_i, v_{i_x})\}$ shall be considered earlier than all cuts $M \in \mathcal{F}^{\leq k}_{v_{i_x},v_j}$. This suffices to extend the order to all of $\mathcal{F}_x$ as we already know the relative order for all pairs of v_{i_x}-v_j-latest $\leq k$-cuts, by the inductive assumption.

To argue about the correctness of the equivalence in (5), observe that for every x, and $M_x, M'_x \in \mathcal{F}_x$, $M'_x \geq M_x$ holds if and only if every path from v_i to v_j using (v_i, v_{i_x}) as its first arc intersects M_x no later than intersecting M'_x. Thus, this partial later-order on $\mathcal{F}_x$ properly extends to the partial later-order that we want on (minimal) cuts, and to filter $\mathcal{M}$ it is enough to keep those $M \in \mathcal{M}$ such that for no $M' \in \mathcal{M}$ there is $M' > M$ and $|M'| \leq |M|$.

The total amount of work for the filtering for a single pair of i, j is thus quadratic in $|\mathcal{M}| = 2^{\mathcal{O}(k^2)}$, the number of cuts in $\mathcal{M}$, linear in c, the number of set families $\mathcal{F}_x$, and quadratic in $2^{\mathcal{O}(k)}$, the size of each set family. As $\text{poly}(2^{\mathcal{O}(k^2)}) \cdot c \cdot \text{poly}(2^{\mathcal{O}(k)}) = 2^{\mathcal{O}(k^2)} \cdot c$, where c is the outdegree of v_i, we get the claimed bound on the runtime. ◀

▶ **Corollary 4.15.** *All latest cuts of size at most $k = o(\sqrt{\log n})$ for all pairs of vertices of a DAG can be found in $\mathcal{O}(mn^{1+o(1)})$ total time.*

4.4.2 Coding for the Witness Superset Problem

Binary codes. A binary code $\mathcal{C}$ of length q on a universe of size u is a function $[u] \to 2^{[q]}$. Note that we employ set formalism to describe codes, i.e., each element of the universe is mapped to a subset of the code set. This is equivalent to mapping to binary codes of length q, used for instance in implementations. Correspondingly, we talk about the bitwise-OR operation on codewords, which is equivalent under the set formalism to taking the union of two characteristic sets.

Tensor products and powers. Given two binary codes $\mathcal{C}_1$, $\mathcal{C}_2$, we define the *tensor product* $\mathcal{C}_1 \otimes \mathcal{C}_2 : [u_1] \times [u_2] \to 2^{[q_1]\times[q_2]}$ of the two codes $\mathcal{C}_1$ and $\mathcal{C}_2$ as the function $(\mathcal{C}_1 \otimes \mathcal{C}_2)(w_1, w_2) = \mathcal{C}_1(w_1) \times \mathcal{C}_2(w_2)$.[1] The tensor product resembles the construction of concatenated codes: instead of the outer code $\mathcal{C}_1(w_1)$, each '1' in $\mathcal{C}_1(w_1)$ is replaced with $\mathcal{C}_2(w_2)$, and each '0' with a sequence of '0' of the appropriate length. We call the replacement of each entry of $\mathcal{C}_1(w_1)$ (that is, by $\mathcal{C}_2(w_2)$ or by '0's) a *column* of the codeword $(\mathcal{C}_1 \otimes \mathcal{C}_2)(w_1, w_2)$.

We define the p-th *tensor power* $\mathcal{C}^{\otimes p} : [u]^p \to 2^{[q]^p}$ of a code $\mathcal{C}$ as the tensor product of p copies of $\mathcal{C}$:

$$\mathcal{C}^{\otimes p} = \underbrace{\mathcal{C} \otimes \ldots \otimes \mathcal{C}}_{p \text{ times}},$$

which is equivalent to taking the tensor product after applying the code to each argument:

$$(\mathcal{C}^{\otimes p})(w_1, w_2, \ldots, w_p) = \mathcal{C}(w_1) \times \mathcal{C}(w_2) \times \ldots \times \mathcal{C}(w_p).$$

Superimposed codes. To apply the code to a subset X of $[u]$, we write $\mathcal{C}(X) = \bigcup_{x\in X} \mathcal{C}(x)$. A binary code $\mathcal{C}$ is called d-superimposed [34], if the union of at most d codewords is uniquely decodable, or equivalently

$$\forall_{X:|X|\le d} \forall_{y\notin X} \mathcal{C}(y) \not\subseteq \mathcal{C}(X).$$

We refer to Figure 22 for an illustration.

A standard deterministic construction based on Reed-Solomon error correction codes (cf. [40]) gives d-superimposed codes of length $d \log^2 u$ (cf. Kautz and Singleton [28]). This construction is very close to the information-theoretic lowerbound of $\Omega(d \log_d u)$. By decoding a superimposed code, we understand computing $X = \mathcal{C}^{-1}(Y)$, if $|X| \le d$, or deciding that there is no such X. When considering decoding time, Indyk et al. [25] gave the first (randomized) construction of superimposed codes decodable in time $\mathcal{O}(\text{poly}(d \log u))$ that is close to the lower-bound on length of codes, while Ngo et al. [36] provided a derandomized construction. However, for our purposes any superimposed code of length and decoding time $\mathcal{O}(\text{poly}(d \log u))$ is sufficient, thus we use the following folklore result.

[1] $\mathcal{C}_1 \times \mathcal{C}_2$ is interpreted as a code from natural bijection between $[u_1] \times [u_2]$ and $[u_1 \cdot u_2]$.

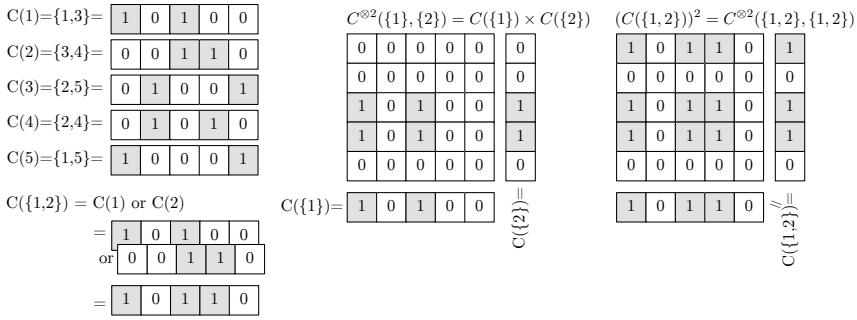


Figure 22 (left) A 2-superimposed code $\mathcal{C}$ on the universe $[5] = \{1, 2, 3, 4, 5\}$. Notice how the encoding is unique for any set of size at most 2. For instance, the code $\mathcal{C}(\{1, 2\})$ shown is uniquely decodable to $\{1, 2\}$. This is not true for larger sets however, e.g. $\mathcal{C}(\{1, 2, 3\}) = \mathcal{C}(\{2, 3, 5\})$. (right) The second tensor power $\mathcal{C}^{\times 2}$ of the code $\mathcal{C}$ illustrated as a two dimensional matrix and applied once to the two singleton sets $\{1\}$ and $\{2\}$, and once to the set $\{1, 2\}$ in both dimensions.

▶ **Theorem 4.16.** *Let $\mathcal{C}_{slow}$ be an arbitrary d-superimposed code of length $\mathcal{O}(poly(d \log u))$ (for example by the Kautz-Singleton construction), and let $\mathcal{C}_1$ be a 1-superimposed code decodable in $\mathcal{O}(\log u)$ time and space (for example the code adding a parity bit after every bit of the input's binary representation). Then $\mathcal{C}_{fast} = \mathcal{C}_{slow} \otimes \mathcal{C}_1$ is a d-superimposed code decodable in $\mathcal{O}(poly(d \log u))$ time and space.*

Proof. $\mathcal{C}_{\text{fast}}$ being d-superimposed follows from it having non-zero columns only in the positions corresponding to non-zero bits of $\mathcal{C}_{\text{slow}}$. Since $\mathcal{C}_{\text{slow}}$ is d-superimposed, for any X such that $|X| \leq d$, for all elements x of X, there exists a coordinate that is in the encoding of X only because of x, i.e., there is exists i such that $i \in \mathcal{C}_{\text{slow}}(x)$ and $i \notin \mathcal{C}_{\text{slow}}(X \setminus \{x\})$. Thus, given any $S \subseteq [q_{\text{slow}}] \times [q_1]$ to be decoded by $\mathcal{C}_{\text{fast}}$, it is enough to first find the set $\mathcal{I}$ of all i such that the i-th column of S is a proper code-word of $\mathcal{C}_1$, to set $X = \cup_{i \in \mathcal{I}} C_1^{-1}(i\text{-th column of } S)$, and to test if $(\mathcal{C}_{\text{slow}} \times \mathcal{C}_1)(X) = S$. ◀

One can see superimposed codes as a short encoding of sets, that are decodable up to a certain size, and preserve set union under set codeword bitwise-OR. Now we develop codes that allow for encoding of set families, with the goal of decoding solutions to Witness Superset from results of the bitwise-OR of encoded inputs.

More specifically, assume $\mathcal{C}$ is a k-superimposed code (on a universe

of size u), and consider the code $\mathcal{C}^{\otimes K}$. If $\mathcal{F} = \{F_1, F_2, \ldots, F_K\}$ is a set family of K sets of at most k elements from $[u]$, then by slightly bending the notation there is

$$(\mathcal{C}^{\otimes K})(\mathcal{F}) = \mathcal{C}(F_1) \times \ldots \times \mathcal{C}(F_K),$$

(the order of sets $F_1, \ldots, F_K$ is chosen arbitrarily). If $|\mathcal{F}| < K$ then we append several copies of, i.e., F_1 in the encoding. Moreover, we need to guarantee that $\mathcal{F} \neq \emptyset$.

The intuition behind this construction is to assign a separate dimension for each set of the set family. This creates enough redundancy so that the code has some desired properties under bitwise-OR. Simpler codes, for example concatenating instead of taking a tensor product, do not have those properties.

Taking slices. We define taking *slices* of smaller dimension from multidimensional sets, by fixing one (or more) coordinates to specific values, that is taking a subset of the original set that has fixed coordinates equal to the desired ones, and then eliminating those coordinates from the tuples. For example, for a set $\{(0,1,2),(2,2,0),(3,1,2)\}$ setting its second coordinate to $'1'$ results in the slice $\{(0,2),(3,2)\}$. This is a simple operation that reduces the level of redundancy in the code: observe that a slice of $\mathcal{C}^{\otimes K}(\mathcal{F})$ is just an encoding $\mathcal{C}^{\otimes K'}(\mathcal{F}')$ for some $\mathcal{F}' \subset \mathcal{F}$ and $K' < K$.

Solving Witness Superset. We now develop Algorithm 24, which takes an encoded input S of an instance of Witness Superset and solves it by recursively taking unions S' of at most k different slices of S. This way, each recursive call reduces the dimension by one from $\mathcal{C}^{\otimes K}$ to $\mathcal{C}^{\otimes K-1}$. Correctness follows from formalizing the following property: for each solution W, we are sure that (at least) one recursive call considers a slice-union S' for which each element $x \in W$ is either easily observed already in S or we are sure that x is still necessary within S'. We refer to Figure 23 for an illustration.

▶ **Theorem 4.17.** *Algorithm 24 solves the* Witness Superset *problem from the bitwise-OR of*

$$(\mathcal{C}^{\otimes K})(\mathcal{F}_1), \ldots, (\mathcal{C}^{\otimes K})(\mathcal{F}_c)$$

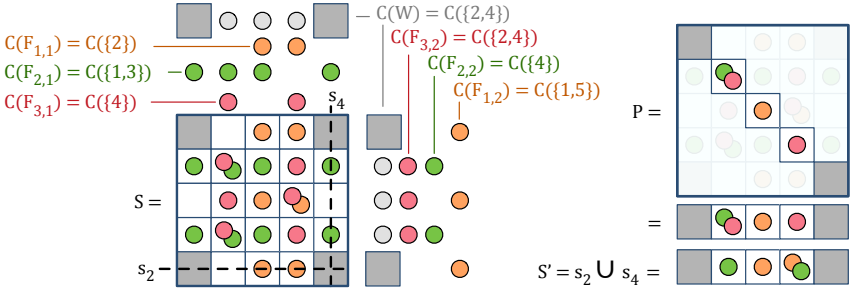


Figure 23 Illustration of Algorithm 24 solving an instance of Witness Superset, namely the set families $\mathcal{F}_1 = \{\{2\},\{1,5\}\}$, $\mathcal{F}_2 = \{\{1,3\},\{4\}\}$ and $\mathcal{F}_3 = \{\{4\},\{2,4\}\}$, so we have $u = 5$, $c = 3$, $K = 2$ and $k = 2$. We use the same superimposed code $\mathcal{C}$ of length $q = 5$ as in Figure 22 to form the input S. S is the bitwise-OR encoding of the set families, namely $S = (\mathcal{C}(F_{1,1}) \times \mathcal{C}(F_{1,2})) \cup (\mathcal{C}(F_{2,1}) \times \mathcal{C}(F_{2,2})) \cup (\mathcal{C}(F_{3,1}) \times \mathcal{C}(F_{3,2}))$. The only solution to this instance is $W = \{2,4\}$. As shown by Lemma 4.18, we have $S \cap ([q] \setminus \mathcal{C}(W))^2 = \emptyset$, i.e., all gray boxes are free from colored dots. Note that both cases of the argument in Theorem 4.17 apply here. To see that the first case applies for $x = 4$, note that $\mathcal{F}_3$ is 4-W-critical, and that we have $F_{3,1} \subseteq W$ and $F_{3,2} \subseteq W$. Thus, $\mathcal{C}(4) = \{2,4\} \subseteq P$. The set P corresponds to the diagonal of the matrix S. In this specific example, we even have $\mathcal{C}(W) \subseteq P$ because $P = \{r \mid (r,r) \in S\} = \{2,3,4\}$ contains element 3 as well since $(3,3) \in \mathcal{C}(\mathcal{F}_1) \subseteq S$. For the second case, the two slices s_2 and s_4 are shown as dotted lines. For $x = 2$, $\mathcal{F}_1$ is x-W-critical and $F_{1,2} \not\subseteq W$. Hence, we can still observe 2 within slice s_2. Analogously for $x = 4$, $\mathcal{F}_2$ is x-W-critical and $F_{2,1} \not\subseteq W$ and thus s_4 still enforces that $W \setminus \{4\}$ is not a solution. In the recursion for $S' = s_2 \cup s_4$, we get $S' = \mathcal{C}(W)$, which corresponds to the base case of the induction.

Algorithm 24 decoding of $\mathcal{C}^{\otimes K}$

Require: S, bitwise-OR of set families encoded with $\mathcal{C}^{\otimes K}$
Ensure: decodeWitness returns all possible solutions to the Witness Superset problem on S

procedure collapse(S)
  $ans \leftarrow \emptyset$
  if $K = 1$ **then**
    ans.insert(S)
  else
    $P \leftarrow \{r \mid (r, r, \ldots, r) \in S\}$ ▷ P are those coordinates that have to be in any solution
    for $s_1, s_2, \ldots, s_k \in$ all $(K-1)$-dimensional slices of S **do**
      $S' \leftarrow \bigcup_i s_i$
      for $\mathcal{I} \in$ collapse(S') **do**
        ans.insert($\mathcal{I} \cup P$)
      end for
    end for
  end if
  return ans
end procedure
procedure decodeWitness(S)
  $solutions \leftarrow \emptyset$
  $q \leftarrow$ length of $\mathcal{C}$
  for $\mathcal{I} \in$ collapse(S) **do**
    if $\mathcal{I}$ is decodable by $\mathcal{C}$ into a set of size at most k **then**
      if $S \cap \left([q] \setminus \mathcal{I}\right)^K = \emptyset$ **then**
        $W \leftarrow \mathcal{C}^{-1}(\mathcal{I})$
        if $\forall\, W' \subset W\colon\ S \cap \left([q] \setminus \mathcal{C}(W')\right)^K \neq \emptyset$ **then**
          $solutions$.insert(W) ▷ W is minimal
        end if
      end if
    end if
  end for
end procedure

in time $\mathcal{O}((Kk\log u)^{\mathcal{O}(K\cdot k)})$*, assuming all set families* $\mathcal{F}_i$ *are over the universe* $[u]$*, and* $\mathcal{C}$ *is of size* $\mathcal{O}(poly(k\log u))$ *and has a fast decoding procedure (i.e., the construction from Theorem 4.16).*

Proof. Let $q = \mathcal{O}(\text{poly}(k\log u))$ be the length of codewords in $\mathcal{C}$. Denote $S \subseteq [q]^K$ as the input to be decoded, and denote the (to us unknown) sets from the set families of the input as: $\mathcal{F}_i = \{F_{i,1}, F_{i,2}, \ldots, F_{i,K}\}$. First, we show that there is a characterization of solutions in the language of tensor products.

▶ **Lemma 4.18.** *For any* $W \subseteq [u]$*:* W *satisfies [cover] if and only if* $S \cap ([q] \setminus \mathcal{C}(W))^K = \emptyset$*.*

Proof. We start with the definition of [cover] (1) and the fact that $\mathcal{C}$ properly represents set containment for sets of size at most k (2):

$$W \text{ satisfies [cover]} \overset{(1)}{\Leftrightarrow} \forall i\ \exists j\colon\ F_{i,j} \subseteq W \ \overset{(2)}{\Leftrightarrow}\ \forall i\ \exists j\colon\ \mathcal{C}(F_{i,j}) \subseteq \mathcal{C}(W).$$

We use $\mathcal{C}(\mathcal{F}_i) = \mathcal{C}(F_{i,1}) \times \cdots \times \mathcal{C}(F_{i,j}) \times \cdots \times \mathcal{C}(F_{i,K})$ and $\mathcal{C}(F_{i,j'}) \subseteq [q]$ for all $j' \neq j$ (3) and the equivalence $A \subseteq B \Leftrightarrow A \cap ([q] \setminus B) = \emptyset$ (4) to get

$$\begin{aligned}
&\forall i\ \exists j\colon\ \mathcal{C}(F_{i,j}) \subseteq \mathcal{C}(W)\\
\overset{(3)}{\Leftrightarrow}\ &\forall i\ \exists j\colon\ \mathcal{C}(\mathcal{F}_i) \subseteq [q]^{j-1} \times \mathcal{C}(W) \times [q]^{K-j}\\
\overset{(4)}{\Leftrightarrow}\ &\forall i\ \exists j\colon\ \mathcal{C}(\mathcal{F}_i) \cap ([q]^{j-1} \times ([q] \setminus \mathcal{C}(W)) \times [q]^{K-j}) = \emptyset.
\end{aligned}$$

We now exploit that the codes $\mathcal{C}(\mathcal{F}_i)$ are tensor products (5) to get the following equivalence

$$\begin{aligned}
&\forall i\ \exists j\colon\ \mathcal{C}(\mathcal{F}_i) \cap ([q]^{j-1} \times ([q] \setminus \mathcal{C}(W)) \times [q]^{K-j}) = \emptyset\\
\overset{(5)}{\Leftrightarrow}\ &\forall i\colon\ \mathcal{C}(\mathcal{F}_i) \cap ([q] \setminus \mathcal{C}(W))^K = \emptyset\\
\overset{(6)}{\Leftrightarrow}\ &(\bigcup_i \mathcal{C}(\mathcal{F}_i)) \cap ([q] \setminus \mathcal{C}(W))^K = \emptyset\\
\overset{(7)}{\Leftrightarrow}\ &S \cap ([q] \setminus \mathcal{C}(W))^K = \emptyset.
\end{aligned}$$

Taking the union (6) and applying the definition of S (7) concludes the proof. ◀

Now, consider any set W that we output. Obviously $|W| \leq k$, and applying Lemma 4.18 to the checks on lines (21) and (23) of Algorithm 24

ensures that W satisfies [cover] and [minimal]. Thus, W is a solution to the Witness Superset problem.

Next, we reason that any solution W to the Witness Superset problem is generated by the collapse function, by induction on K. If $K = 1$, then S is the bitwise-OR (union) of the encoded sets, and since collapse outputs S, the condition is trivially satisfied.

For the inductive step, we first observe that any $(K-1)$-dimensional slice s_i of S is the bitwise-OR of $\mathcal{C}^{\otimes K-1}$ encoded set families, and thus so is S'.

Consider any slice s of S by fixing a single dimension to a value $\alpha \notin \mathcal{C}(W)$. Then, similar to the arguments in Lemma 4.18, we have $s \cap ([q] \setminus \mathcal{C}(W))^{K-1} = \emptyset$. Additionally, observe that $P \subseteq \mathcal{C}(W)$ as otherwise $([q] \setminus \mathcal{C}(W))^K$ would intersect S.

Now, fix any $x \in W$. Since W is inclusion minimal, there is at least one family $\mathcal{F}_i$ that requires x to be in W, which we call x-W-critical.

Property 4.19 (critical family). *$\mathcal{F}_i$ is x-W-critical, iff for all j, if $F_{i,j} \subseteq W$ then $x \in F_{i,j}$.*

We now consider two, not necessarily disjoint cases based on all those families in $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_c$ that are x-W-critical (see Figure 23 for an illustration):

- There is a x-W-critical family $\mathcal{F}_i$ such that for all j, $F_{i,j} \subseteq W$. By Property 4.19 we have for all j, $x \in F_{i,j}$, and thus $(\mathcal{C}(x))^K \subseteq \mathcal{C}(\mathcal{F}_i) \subseteq S$ and hence $\mathcal{C}(x) \subseteq P$.
- There is at least one x-W-critical family $\mathcal{F}_i$, such that for at least one value of j' we have $F_{i,j'} \not\subseteq W$. Hence $\mathcal{F}_i \setminus \{F_{i,j'}\}$ is also x-W-critical. Now consider the slice s_x of S by fixing its j'-th dimension to some arbitrarily chosen $\alpha \in \mathcal{C}(F_{i,j'}) \setminus \mathcal{C}(W)$. Since slice s_x is a hyperplane going through $\mathcal{C}^{\otimes K}(\mathcal{F}_i)$, s_x contains $\mathcal{C}(\mathcal{F}_i \setminus \{F_{i,j'}\})$, and hence $W \setminus \{x\}$ is not a solution for s_x, but W is.

Thus, we reach the conclusion that for any $x \in W$, there either exists a slice s_x of S that requires x in at least one of its minimal solutions, and has W as a solution, or x is encoded in P. Let $W^* = \{x \in W \mid s_x \text{ exists}\}$. Thus, there is $S' = \bigcup_{x \in W^*} s_x$ such that S' has a minimal solution W' and such that $\mathcal{C}(W') \cup P = \mathcal{C}(W)$. Since Algorithm 24 exhaustively searches all combinations of at most k slices deterministically, S' is used in at least one of the recursive calls of collapse.

To bound the running time, observe that the number of slices on a single level of the recursion is $K \cdot q$, thus the branching factor of COLLAPSE is upper-bounded by $(Kq)^k$, with recursion depth K. All codes are of size $\mathcal{O}(q^K)$, and computing $\mathcal{C}$ and $\mathcal{C}^{-1}$ takes time $\text{poly}(q)$. All in all, this leads to the claimed time $\mathcal{O}((Kk \log u)^{\mathcal{O}(Kk)})$. ◀

4.4.3 All-pairs k-reachability for $k = o(\log \log n)$

▶ **Theorem 4.20.** *All latest cuts of size at most k for all pairs of vertices of a DAG can be found in $\mathcal{O}((k \log n)^{4^{k+o(k)}} \cdot n^\omega)$ total time.*

Proof. We show a divide-and-conquer algorithm. Without loss of generality, assume that n is even (if not, add one unique isolated vertex). Let $V_1 = v_1, v_2, \ldots, v_{n/2}$ and $V_2 = v_{n/2+1}, \ldots, v_{n-1}, v_n$, and let $A_1 = A(D[V_1])$, $A_{1,2} = A[V_1, V_2]$ and $A_2 = A(D[V_2])$. It is enough to show how to find all pairs earliest/latest $\leq k$-cuts in D, having recursively computed all earliest/latest $\leq k$-cuts in (V_1, A_1) and in (V_2, A_2) in the claimed time bound, since the recursive equation for the runtime $T(n) = 2T(n/2) + \mathcal{O}((k \log n)^{4^{k+o(k)}} \cdot n^\omega)$ has the desired solution.

Notice that the pairwise cuts between vertices in V_1 are the same in D as in (V_1, A_1), and the same holds for V_2 and (V_2, A_2). Thus, all we need is to find pairwise cuts from V_1 to V_2 in D. We proceed as follows. First of all, we merge the information on pairwise cuts in (V_2, A_2) with arcs from $A_{1,2}$ to compute all pairwise cuts from V_1 to V_2 in the graph $(V, A_{1,2} \cup A_2)$. The second step is to merge the result of the first step with all pairwise cuts in (V_1, A_1) to compute the desired pairwise cuts in D. Since both procedures involve essentially the same steps, we describe in detail only the first one.

Encoding: Let $K \leq 4^k$ be the bound from Lemma 4.6. We use $\mathcal{C}^{\otimes K}$ described in Theorem 4.17 with a universe of size m to represent pairwise latest/earliest cuts[2]. We then build two encoded matrices of dimension $n/2 \times n/2$. Matrix Y, defined by $Y_{i,j} = \mathcal{C}^{\otimes K}(\mathcal{E}^{\leq k}_{v_i, v_j})$, encodes all earliest v_i-v_j $\leq k$-cuts in the graph (V_2, A_2). Matrix X encodes cuts from V_1 to V_2 in graph $(V, A_{1,2})$, which has a much simpler structure: the cut is the set of all arcs from v_i to v_j, or there is no cut if there are more than k parallel arcs.

Matrix multiplication: The following procedure is used

[2] For multigraphs, we require $m \leq 2^{\text{polylog}(n)}$ for our bound to hold.

1. Lift X into matrix X', changing each entry from a K-dimensional tensor product to $2K$-dimensional, by setting $X'_{i,j} = X_{i,j} \times [t]^K$.
2. Lift Y into Y' by setting $Y'_{i,j} = [t]^K \times Y_{i,j}$.
3. Compute the coordinate-wise Boolean matrix product of X' and Y', resulting in Z'.

We denote the above steps as $X \star Y = Z'$. We note that if for some indices a, b, c, the entry $X_{a,b}$ encodes some set family $\mathcal{E}$ and $Y_{b,c}$ encodes some set family $\mathcal{F}$, then the bitwise product of $X'_{a,b}$ and $Y'_{b,c}$ encodes the set family $\mathcal{E} \cup \mathcal{F}$. Thus, every entry $Z'_{i,j}$ is a bitwise-OR of all v_i-v_a-earliest $\leq k$-cuts in $(V, A_{1,2})$ and all v_a-$v_{j+n/2}$-latest $\leq k$-cuts encoded with $\mathcal{C}^{\otimes 2K}$. Hence, by Theorem 4.17, Algorithm 24 applied to each entry of Z' solves WITNESS SUPERSET. Since $A_{1,2}, A_2$ is an arc split of $(V, A_{1,2} \cup A_2)$, by Theorem 4.8 each solution in the output is a v_i-$v_{j+n/2}$ cut in $(V, A_{1,2} \cup A_2)$ and at least one solution is a min-cut, if the v_i-$v_{j+n/2}$ min-cut is of size at most k.

Fixing: So far, we have only found all pairwise min-cuts, if smaller than k. As a final step, we describe how to extract all latest $\leq k$-cuts (earliest $\leq k$-cuts follow by a symmetrical approach). For any pair v_i,v_j, let $y_1, y_2, \ldots, y_d$ be all heads of the v_i-v_j min-cut found in the previous step. We first recursively ask for all latest y_1-v_j, y_2-v_j, $\ldots$, y_d-v_j $\leq k$-cuts, for the particular pairs that were not computed already. Then, we build the corresponding instance of WITNESS SUPERSET, which by Lemma 4.13 can be solved by Algorithm 23 in time $2^{\mathcal{O}(k^2)} \cdot \mathrm{poly}(k) = 2^{\mathcal{O}(k^2)}$, and by Theorem 4.10 contains among its solution all desired latest cuts. A filtering procedure as in the proof of Theorem 4.14 is applied as a final step. As we compute the latest cuts exactly once for each pair i, j, in total this post-processing takes at most $2^{\mathcal{O}(k^2)} n^2$ steps.

Finally, we note that $C^{\otimes 2K}$ codes are of length $\mathcal{O}((\mathrm{poly}(k \log n))^{4^k}) = \mathcal{O}((k \log n)^{4^{k+o(k)}})$, and Algorithm 24 runs in time $\mathcal{O}((4^k \mathrm{poly}(k \log n))^{\mathcal{O}(4^k \cdot k)}) = \mathcal{O}((k \log n)^{4^{k+o(k)}})$, giving the desired runtime per matrix entry. ◀

▶ **Corollary 4.21.** *All latest cuts of size at most $k = o(\log \log n)$ for all pairs of vertices of DAG can be found in $\mathcal{O}(n^{\omega+o(1)})$ total time.*

5
Concluding Remarks

In conclusion, this thesis contributes to three variants of the digraph reachability problem. On fully dynamic graphs of partial functions, we present an algorithm that supports updates and queries in time $\mathcal{O}(\log n)$. For static graphs, we develop a solution for 2-reachability running in time $\mathcal{O}(n^\omega \log n)$ using a novel path algebra and dominator trees. For k-reachability on DAGs, we invent tools on top of extremal $\leq k$-cuts to come up with two algorithms: one running in time $\mathcal{O}(2^{\mathcal{O}(k^2)} \cdot mn)$, interesting for sparse graphs and $k \in o(\sqrt{\log n})$, and one running in time $\mathcal{O}((k \log n)^{4^k + o(k)} \cdot n^\omega)$, interesting for dense graphs and $k \in o(\log \log n)$.

It remains to be investigated whether it is worth implementing our algorithms in practice. That both our 2-reachability algorithm and our second k-reachability DAG-algorithm rely on fast matrix multipliciation might make them worth implementing only for very large and dense graphs. While the lower bound from transitive closure prohibits efficient solutions for general large graphs, many real-world graphs, like street networks, might have a structured 2-reachability matrix which one might exploit in algorithms running faster than $\mathcal{O}(n^\omega)$.

Another interesting open question is whether we can come up with algorithms for k-reachability on cyclic graphs. Since our tools rely on the existence of arc splits in topological orders, even deciding 3-reachability on 2-strongly connected graphs can not be tackled easily.

We are confident that our results will stimulate further research into this intriguing and fundamental area of graph algorithms.

Bibliography

[1] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *55th IEEE Symposium on Foundations of Computer Science, FOCS'14*, pages 434–443. IEEE, 2014. doi:10.1109/FOCS.2014.53.

[2] Noga Alon, Zvi Galil, Oded Margalit, and Moni Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *33rd Symposium on Foundations of Computer Science FOCS'92*, pages 417–426. IEEE, 1992. doi:10.1109/SFCS.1992.267748.

[3] Stephen Alstrup, Dov Harel, Peter W. Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–32, 1999. doi:10.1137/S0097539797317263.

[4] Stephen Alstrup, Jens Peter Secher, and Maz Spork. Optimal on-line decremental connectivity in trees. *Information Processing Letters (IPL)*, 64(4):161–164, 1997. doi:10.1016/S0020-0190(97)00170-1.

[5] Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant subgraph for single source reachability: generic and optimal. In *48th ACM Symposium on Theory of Computing, STOC'16*, pages 509–518, 2016. doi:10.1145/2897518.2897648.

[6] Samuel W Bent, Daniel D Sleator, and Robert E Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985. doi:10.1137/0214041.

[7] Paolo Boldi, Marco Rosa, and Sebastiano Vigna. Robustness of social and web graphs to node removal. *Social Network Analysis and Mining*, 3(4):829–842, 2013. doi:10.1007/s13278-013-0096-x.

[8] Adam L. Buchsbaum, Loukas Georgiadis, Haim Kaplan, Anne Rogers, Robert E. Tarjan, and Jeffery R. Westbrook. Linear-time algorithms for dominators and other path-evaluation problems. *SIAM Journal on Computing*, 38(4):1533–1573, 2008. doi:10.1137/070693217.

[9] Marek Cygan, Fedor V Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 3. Springer, 2015. doi:10.1007/978-3-319-21275-3.

[10] Artur Czumaj, Mirosław Kowaluk, and Andrzej Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, July 2007. doi:10.1016/j.tcs.2007.02.053.

[11] Erik Demaine. Lecture 19: Link-cut trees. http://courses.csail.mit.edu/6.851/spring14/lectures/L19.html.

[12] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*, pages 9–1–9–28. Chapman & Hall/CRC, 2010.

[13] Michael J. Fischer and Albert R. Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory, SWAT'71*, pages 129–131. IEEE, 1971. doi:10.1109/SWAT.1971.4.

[14] Lester Randolph Ford Jr. and Delbert Ray Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[15] D Language Foundation. Mixins in D. https://dlang.org/mixin.html.

[16] Wojciech Fraczak, Loukas Georgiadis, Andrew Miller, and Robert E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. doi:doi:10.1016/j.jda.2013.10.003.

[17] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing, STOC'89*, pages 345–354. ACM, 1989. doi:10.1145/73007.73040.

[18] Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms (TALG)*, 13(1):9:1–9:24, 2016. doi:10.1145/2968448.

[19] Loukas Georgiadis, Giuseppe F. Italiano, and Nikos Parotsidis. Strong connectivity in directed graphs under failures. In *28th ACM-SIAM Symposium on Discrete Algorithms, SODA'17*, pages 1880–1899, 2017. doi:10.1137/1.9781611974782.123.

[20] Fabrizio Grandoni and Virginia V. Williams. Improved distance sensitivity oracles via fast single-source replacement paths. In *53rd IEEE Symposium on Foundations of Computer Science, FOCS'12*, pages 748–757, Oct 2012. doi:10.1109/FOCS.2012.17.

[21] Bernhard Haeupler, Siddhartha Sen, and Robert E Tarjan. Rank-balanced trees. *ACM Transactions on Algorithms (TALG)*, 11(4):30, 2015. doi:10.1145/2689412.

[22] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 46(4):502–516, 1999. doi:10.1145/320211.320215.

[23] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001. doi:10.1145/502090.502095.

[24] Thore Husfeldt. Fully dynamic transitive closure in plane dags with one source and one sink. In *3rd European Symposium on Algorithms, ESA'95*, pages 199–212, 1995. doi:10.1007/3-540-60313-1_144.

[25] Piotr Indyk, Hung Q. Ngo, and Atri Rudra. Efficiently decodable non-adaptive group testing. In *21st ACM-SIAM Symposium on Discrete Algorithms, SODA'10*, pages 1126–1142, 2010. doi:10.1137/1.9781611973075.91.

[26] Giuseppe F. Italiano, Luigi Laura, and Federico Santaroni. Finding strong bridges and strong articulation points in linear time. *Theoretical Computer Science*, 447:74–84, 2012. doi:10.1016/j.tcs.2011.11.011.

[27] Giuseppe F. Italiano, Alberto Marchetti-Spaccamela, and Umberto Nanni. Dynamic data structures for series parallel digraphs. In *1st Workshop on Algorithms and Data Structures, WADS'89*, pages 352–372, 1989. doi:10.1007/3-540-51542-9_30.

[28] William H. Kautz and Richard C. Singleton. Nonrandom binary superimposed codes. *IEEE Transactions on Information Theory*, 10(4):363–377, October 1964. doi:10.1109/TIT.1964.1053689.

[29] Valerie King and Garry Sagert. A fully dynamic algorithm for maintaining the transitive closure. *Journal of Computer and System Sciences*, 65(1):150 – 167, 2002. doi:10.1006/jcss.2002.1883.

[30] François Le Gall. Powers of tensors and fast matrix multiplication. In *39th International Symposium on Symbolic and Algebraic Computation, ISSAC'14*, pages 296–303, 2014. doi:10.1145/2608628.2608664.

[31] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–41, 1979. doi:10.1145/357062.357071.

[32] Dániel Marx. Parameterized graph separation problems. *Theoretical Computer Science*, 351(3):394–406, 2006. doi:10.1007/978-3-540-28639-4_7.

[33] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10:96–115, 1927.

[34] Calvin N. Mooers. *Application of Random Codes to the Gathering of Statistical Information.* 1948.

[35] Ian Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters (IPL)*, 1(2):56–58, 1971. doi:10.1016/0020-0190(71)90006-8.

[36] Hung Q. Ngo, Ely Porat, and Atri Rudra. Efficiently decodable error-correcting list disjunct matrices and applications - (extended abstract). In *38th International Colloquium on Automata, Languages and Programming, ICALP'11*, pages 557–568, 2011. doi:10.1007/978-3-642-22006-7_47.

[37] Nilakantha Paudel, Loukas Georgiadis, and Giuseppe F. Italiano. Computing critical nodes in directed graphs. In *19th Workshop on Algorithm Engineering and Experiments, ALENEX'17*, pages 43–57, 2017. doi:10.1145/3228332.

[38] Mihai Pătraşcu. Towards polynomial lower bounds for dynamic problems. In *42nd ACM Symposium on Theory of Computing, STOC'10*, pages 603–610. ACM, 2010. doi:10.1145/1806689.1806772.

[39] Mihai Pătraşcu and Erik D Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. doi:10.1137/S0097539705447256.

[40] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. doi:10.1137/0108018.

[41] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *36th ACM Symposium on Theory of Computing, STOC'04*, pages 184–191. ACM, 2004. doi:10.1145/1007352.1007387.

[42] Liam Roditty and Uri Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. *SIAM Journal on Computing*, 45(3):712–733, 2016. doi:10.1137/13093618X.

[43] Piotr Sankowski. Dynamic transitive closure via dynamic matrix inverse. In *45th IEEE Symposium on Foundations of Computer Science, FOCS'04*, pages 509–517, 2004. doi:10.1109/FOCS.2004.25.

[44] Piotr Sankowski. Maximum weight bipartite matching in matrix multiplication time. *Theoretical Computer Science*, 410(44):4480–4488, October 2009. doi:10.1016/j.tcs.2009.07.028.

[45] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *Journal of the ACM (JACM)*, 28(1):1–4, 1981. doi:10.1145/322234.322235.

[46] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

[47] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985. doi:10.1145/3828.3835.

[48] Sairam Subramanian. A fully dynamic data structure for reachability in planar digraphs. In *1st European Symposium on Algorithms, ESA'93*, pages 372–383, 1993. doi:10.1007/3-540-57273-2_72.

[49] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. doi:10.1109/SWAT.1971.10.

[50] Robert Endre Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974. doi:10.1137/0203006.

[51] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975. doi:10.1145/321879.321884.

[52] Robert Endre Tarjan. Linking and cutting trees. *Data Structures and Network Algorithms*, pages 59–70, 1983. doi:10.1137/1.9781611970265.

[53] Peter Van Emde Boas. Machine models and simulations. *Handbook of Theoretical Computer Science*, A:1–66, 2014. doi:10.1016/B978-0-444-88071-0.50006-0.

[54] Virginia Vassilevska Williams. Faster replacement paths. In *22nd ACM-SIAM Symposium on Discrete Algorithms, SODA'11*, pages 1337–1346, 2011. doi:10.1137/1.9781611973082.102.

[55] Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):14:1–14:13, 2013. doi:10.1145/2438645.2438646.

[56] Raphael Yuster. All-pairs disjoint paths from a common ancestor in $\widetilde{O}(n^\omega)$ time. *Theoretical Computer Science*, 396(1):145–150, 2008. doi:10.1016/j.tcs.2008.01.032.

[57] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002. doi:10.1145/567112.567114.

Nomenclature

ω	The matrix multiplication exponent
A	The set of arcs of digraph D
D	Digraph with vertex set V and arc set A
$D(v)$	Set of descendents v in DT
$d(v)$	Immediate dominator of v in DT
D^R	Digraph with all arcs of D reversed
D_s	Flow graph of digraph D with source vertex s
DT	Dominator tree of flow graph D_s
H	Auxiliary graph for a strongly connected digraph D
H'	Auxiliary graph for a strongly connected digraph D^R
R_D	The transitive closure matrix of digraph D
V	The set of vertices of digraph D

Curriculum Vitae

Daniel Wolleb-Graf

born on September 9, 1990 in St. Gallen, Switzerland

Education

- Matura, Swiss High School Degree – Kantonsschule Sargans, Switzerland – August 2005 to July 2009
- Bachelor of Science in Computer Science – ETH Zurich, Switzerland – September 2010 to August 2013
- Exchange Semester – University of Washington, Seattle, USA – April to August 2013
- Master of Science in Computer Science – ETH Zurich, Switzerland – September 2013 to August 2015
- PhD Studies in Computer Science – ETH Zurich, Switzerland – started September 2015

Work Experience

- Teaching Assistant, ETH Zurich, Fall 2012, Spring 2014, Spring 2015 - Fall 2018, Zurich
- Software Engineering Internship at Google, Youtube Content ID, September to December 2014, Zurich

- Software Engineering Internship at Apple, Apple Maps Traffic, February to May 2018, Sunnyvale, CA

Publications

- *A Framework for Searching in Graphs in the Presence of Errors.* Joint work with Dariusz Dereniowski, Stefan Tiegel, and Przemysław Uznański. *2nd Symposium on Simplicity in Algorithms, SOSA'19.*
- *Faster Algorithms for All-Pairs Bounded Min-Cuts.* Joint work with Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, and Przemysław Uznański. ArXiv preprint arXiv:1807.05803.
- *Hamming Distance Completeness and Sparse Matrix Multiplication.* Joint work with Karim Labib and Przemysław Uznański. *45th International Colloquium on Automata, Languages, and Programming, ICALP'18* Brief Announcement.
- *Collective fast delivery by energy-efficient agents.* Joint work Andreas Bärtschi and Matúš Mihalák. *43rd International Symposium on Mathematical Foundations of Computer Science, MFCS'18.*
- *Truthful Mechanisms for Delivery with Mobile Agents.* Joint work with Andreas Bärtschi and Paolo Penna. *17th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, ATMOS'17.*
- *All-Pairs 2-Reachability in $\mathcal{O}(n^\omega \log n)$ Time.* Joint work with Loukas Georgiadis, Giuseppe F. Italiano, Nikos Parotsidis and Przemysław Uznański. *44th International Colloquium on Automata, Languages, and Programming, ICALP'17.*
- *Energy-efficient Delivery by Heterogeneous Mobile Agents.* Joint work with Andreas Bärtschi, Jérémie Chalopin, Shantanu Das, Yann Disser, Jan Hackfeld and Paolo Penna. *34th International Symposium on Theoretical Aspects of Computer Science, STACS'17.*
- *On Computing the Total Displacement Number via Weighted Motzkin Paths.* Joint work with Andreas Bärtschi, Barbara Geissmann, Tomas Hruz, Paolo Penna and Thomas Tschager. *27th International Workshop on Combinatorial Algorithms, IWOCA'16.*

- *Watch them Fight! Creativity Task Tournaments of the Swiss Olympiad in Informatics.* Joint work with Samuel Grütter and Benjamin Schmid. *10th Olympiads in Informatics Journal, IOI'16.*
- *Collaborative Delivery with Energy-Constrained Mobile Robots.* Joint work with Andreas Bärtschi, Jérémie Chalopin, Shantanu Das, Yann Disser, Barbara Geissmann, Arnaud Labourel and Matúš Mihalák. *23rd International Colloquium on Structural Information and Communication Complexity, SIROCCO'16* and *Theoretical Computer Science Journal 2017.*
- *How to Sort by Walking on a Tree. 23rd European Symposium on Algorithms, ESA'15.*
- *How to Sort by Walking and Swapping on a Tree. Algorithmica Journal 2017.*
- *Mobile Image Retargeting.* Joint work with Daniele Panozzo and Olga Sorkine-Hornung. *18th International Workshop on Vision, Modeling and Visualization, VMV'13.*