All-Pairs 2-Reachability in $\mathcal{O}(n^\omega \log n)$ Time*

Loukas Georgiadis[1], Daniel Graf[2], Giuseppe F. Italiano[†3], Nikos Parotsidis[4], and Przemysław Uznański[5]

1 **University of Ioannina, Ioannina, Greece**
loukas@cs.uoi.gr
2 **Department of Computer Science, ETH Zürich, Zürich, Switzerland**
daniel.graf@inf.ethz.ch
3 **University of Rome Tor Vergata, Roma, Italy**
giuseppe.italiano@uniroma2.it
4 **University of Rome Tor Vergata, Roma, Italy**
nikos.parotsidis@uniroma2.it
5 **Department of Computer Science, ETH Zürich, Zürich, Switzerland**
przemyslaw.uznanski@inf.ethz.ch

Abstract

In the 2-reachability problem we are given a directed graph G and we wish to determine if there are two (edge or vertex) disjoint paths from u to v, for given pair of vertices u and v. In this paper, we present an algorithm that computes 2-reachability information for all pairs of vertices in $\mathcal{O}(n^\omega \log n)$ time, where n is the number of vertices and ω is the matrix multiplication exponent. Hence, we show that the running time of all-pairs 2-reachability is only within a log factor of transitive closure. Moreover, our algorithm produces a witness (i.e., a separating edge or a separating vertex) for all pair of vertices where 2-reachability does not hold. By processing these witnesses, we can compute all the edge- and vertex-dominator trees of G in $\mathcal{O}(n^2)$ additional time, which in turn enables us to answer various connectivity queries in $\mathcal{O}(1)$ time. For instance, we can test in constant time if there is a path from u to v avoiding an edge e, for any pair of query vertices u and v, and any query edge e, or if there is a path from u to v avoiding a vertex w, for any query vertices u, v, and w.

1998 ACM Subject Classification E.1 Graphs and Networks, F.2.2 Computations on Discrete Structures, G.2.2 Graph Algorithms

Keywords and phrases 2-reachability, All Dominator Trees, Directed Graphs, Boolean Matrix Multiplication

Digital Object Identifier 10.4230/LIPIcs.ICALP.2017.74

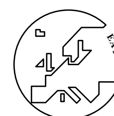
1 Introduction

The *all-pairs reachability problem* consists of preprocessing a directed graph (digraph) $G = (V, E)$ so that we can answer queries that ask if a vertex y is reachable from a vertex x. This problem has many applications, including databases, geographical information systems, social networks, and bioinformatics [11]. A classic solution to this problem is to compute the transitive closure matrix of G, either by performing a graph traversal (e.g., depth-first or breadth-first search) once per each vertex as source, or via matrix multiplication. For a

* A full version of the paper is available at https://arxiv.org/abs/1612.08075.

† Partially supported by MIUR, the Italian Ministry of Education, University and Research, under Project AMANDA (Algorithmics for MAssive and Networked DAta).

© Loukas Georgiadis, Daniel Graf, Giuseppe F. Italiano, Nikos Parotsidis, and Przemysław Uznański;
licensed under Creative Commons License CC-BY



44th International Colloquium on Automata, Languages, and Programming (ICALP 2017).
Editors: Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl; Article No. 74; pp. 74:1–74:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

digraph with n vertices and m edges, the former solution runs in $\mathcal{O}(mn)$ time, while the latter in $\mathcal{O}(n^\omega)$, where ω is the matrix multiplication exponent [4, 13, 17]. Here 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 G so that we can answer fast the following type of queries: For a given vertex pair $x, y \in V$, are there two edge-disjoint (resp., internally vertex-disjoint) paths from x to y? Equivalently, by Menger's theorem [15], we ask if there is an edge $e \in E$ (resp., a vertex $z \in V$) such that there is no path from x to y in $G \setminus e$ (resp., $G \setminus z$). We call such an edge (resp., vertex) *separating* for the pair x, y.

One solution to the all-pairs 2-reachability problem is to compute all the dominator trees of G, with each vertex as source. The dominator tree of G 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 [14]. All the separating edges 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 6). Since a dominator tree can be computed in $\mathcal{O}(m)$ time [2, 3], the overall running time of this algorithm is $\mathcal{O}(mn)$.

Our Results. In this paper, 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, which is asymptotically equivalent to Boolean matrix multiplication [6]. Moreover, our algorithm produces a witness (separating edge or vertex) whenever 2-reachability does not hold. By processing these witnesses, we can find all the dominator trees of G 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 $\mathcal{O}(mn)$ bound for dense graphs. This in turn enables us to answer various connectivity queries in $\mathcal{O}(1)$ time. E.g., we can test in $\mathcal{O}(1)$ time if there is a path from u to v avoiding an edge e, for any pair of query vertices u and v, and any query edge e, 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 edges 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. [9] investigate the effect of an edge or a vertex failure in a digraph G with respect to strong connectivity. Specifically, they show how to preprocess G in $\mathcal{O}(m + n)$ time in order to answer various sensitivity queries regarding strong connectivity in G under an arbitrary edge or vertex failure. For instance, they can compute in $\mathcal{O}(n)$ time the strongly connected components (SCCs) that remain in G after the deletion of an edge or a vertex, or report various statistics such as the number of SCCs in constant time per query (failed) edge 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 edge e leaves two vertices u and v in different SCCs in $G \setminus e$, the algorithm of [9] is not able to distinguish if there is still a path or no path from u to v in $G \setminus e$. Previously, King and Sagert [12] gave an algorithm that can quickly answer sensitivity queries for reachability in a directed acyclic graph (DAG) [12]. Specifically, they show how to process a DAG G so that, for any pair of query vertices x and y, and a query

edge e, one can test in constant time if there is a path from x to y in $G \setminus e$. 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 edge 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. [1], 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 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 G into two auxiliary graphs, so as to reduce 2-reachability queries in G 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.

2 Preliminaries

We assume that the reader is familiar with standard graph terminology, as contained for instance in [5]. Let $G = (V, E)$ be a directed graph (digraph). Given an edge $e = (x, y)$ in E, we denote x (resp., y) as the *tail* (resp., *head*) *of* e. A *directed path* in G is a sequence of vertices v_1, v_2, …, v_k, such that edge $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \ldots, k-1$. The path is said to contain vertex v_i, for $i = 1, 2, \ldots, k$, and edge (v_i, v_{i+1}), for $i = 1, 2, \ldots, k-1$. The *length* of a directed path is given by its number of edges. As a special case, there is a path of length 0 from each vertex to itself. We write $u \leadsto v$ to denote that there is a path from u to v, and $u \not\leadsto v$ if there is no path from u to v. A *directed cycle* is a directed path, with length greater than 0, starting and ending at the same vertex. 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 edge (u, v), u comes before v in the ordering (denoted by $u < v$). A digraph G 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 $G \setminus V'$ the digraph obtained after deleting all the vertices in V', together with their incident edges. Given a subset of edges $E' \subset E$, we denote by $G \setminus E'$ the digraph obtained after deleting all the edges in E'.

2-Reachability and 2-Reachability closure. We write $u \leadsto_{2\mathrm{e}} v$ (resp., $u \leadsto_{2\mathrm{v}} v$) to denote that there are two *edge-disjoint* (resp., internally *vertex-disjoint*) paths from u to v, and $u \not\leadsto_{2\mathrm{e}} v$ (resp., $u \not\leadsto_{2\mathrm{v}} v$) otherwise. As a special case, we assume that $v \leadsto_{2\mathrm{e}} v$ (resp., $v \leadsto_{2\mathrm{v}} v$) for each vertex v in G. We define an abstract set $E^+ = E \cup \{\top, \bot\}$. The semantic of this set is as follows: $e \in E$ corresponds to an edge e separating two vertices, $\top$ corresponds to $\leadsto_{2\mathrm{e}}$ (there is no single separating edge) and $\bot$ corresponds to $\not\leadsto$ (there is no path). Given a digraph G,

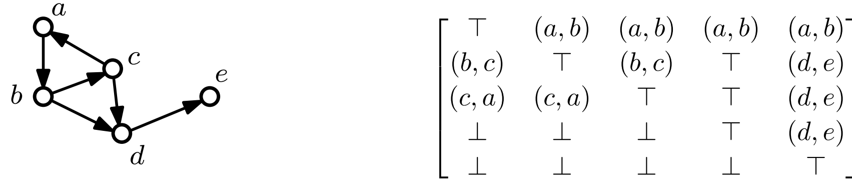


Figure 1 A graph and its (not unique) 2-reachability closure matrix.

we define a 2-*reachability closure* of G, denoted by $G^{\rightsquigarrow_{2e}}$, to be a matrix such that:

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

Since $v \rightsquigarrow_{2e} v$ for each $v \in V$, $G^{\rightsquigarrow_{2e}}[v,v] = \top$. An example of a graph with a 2-reachability closure matrix is given in Figure 1. Note that a 2-reachability closure matrix is not necessarily unique, as there might be multiple separating edges for a given vertex pair. We define the 2-*reachability left closure* $G_L^{\rightsquigarrow_{2e}}$ by replacing *any separating edge* with *first separating edge* and the 2-*reachability right closure* $G_R^{\rightsquigarrow_{2e}}$ by replacing it with *last separating edge*.

Note that if there is only one edge separating u and v, then $G^{\rightsquigarrow_{2e}}[u,v] = G_L^{\rightsquigarrow_{2e}}[u,v] = G_R^{\rightsquigarrow_{2e}}[u,v]$. 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 is completely symmetric). Let u and v be any two vertices. If $G^{\rightsquigarrow_{2e}}[u,v]$ is either $\top$ or $\bot$, then $G_L^{\rightsquigarrow_{2e}}[u,v] = G^{\rightsquigarrow_{2e}}[u,v]$. Otherwise, let $G^{\rightsquigarrow_{2e}}[u,v] = (x,y)$: if $u \rightsquigarrow_{2e} x$ (i.e., if $G^{\rightsquigarrow_{2e}}[u,x] = \top$) then (x,y) is the first separating edge for u and v and $G_L^{\rightsquigarrow_{2e}}[u,v] = (x,y)$; otherwise, $u \not\rightsquigarrow_{2e} x$ (i.e., $G^{\rightsquigarrow_{2e}}[u,x] \neq \top$) and $G_L^{\rightsquigarrow_{2e}}[u,v] = G_L^{\rightsquigarrow_{2e}}[u,x]$. We show how to compute $G_L^{\rightsquigarrow_{2e}}$ and $G_R^{\rightsquigarrow_{2e}}$ from $G^{\rightsquigarrow_{2e}}$ in a total of $\mathcal{O}(n^2)$ worst-case time.

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. Let $G = (V,E)$ be a DAG, and let E_1, E_2 be a partition of its edge set E, $E = E_1 \cup E_2$. We say that a partition is an *edge split* if there is no triplet of vertices x, y, z in G such that $(x,y) \in E_2$ and $(y,z) \in E_1$ simultaneously. Informally speaking, under such split, any path in G from a vertex u to a vertex v consists of a sequence of edges from E_1 followed by a sequence of edges from E_2 (as a special case, any of those sequences can be empty). We denote the edge split by $G = (V, E_1, E_2)$ (See Figure 2). We say that vertex x in $G = (V, E_1, E_2)$ is on the *left* (resp., *right*) *side* of the partition if x is adjacent only to edges in E_1 (resp., E_2). We assume without loss of generality that the vertices of G are given in a topological ordering $v_1, v_2, \ldots, v_n$.

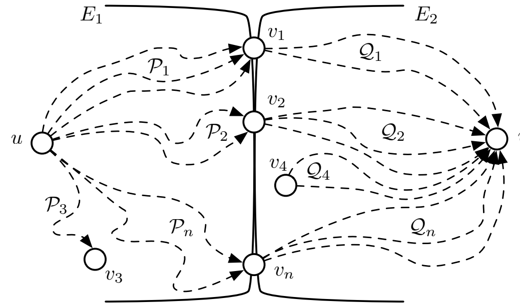


Figure 2 An edge split of a DAG $G = (V, E_1, E_2)$.

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 edge e belongs to every path $P_i \in \mathcal{P}$; or (iii) there is no edge that belongs to all paths in (nonempty) $\mathcal{P}$. To do that, we define the *representation* $\mathrm{repr}(\mathcal{P})$:

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

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* $\mathrm{repr}_L(\mathcal{P}) \in E^+$, where $E^+ = E \cup \{\top, \bot\}$, as follows:

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

A *right representation* $\mathrm{repr}_R(\mathcal{P}) \in E^+$ is defined symmetrically to $\mathrm{repr}_L(\mathcal{P})$, by replacing *minimum* with *maximum*. If $\mathrm{repr}_L(\mathcal{P}) \in E$ (resp., $\mathrm{repr}_R(\mathcal{P}) \in E$), we say that $\mathrm{repr}_L(\mathcal{P})$ (resp., $\mathrm{repr}_R(\mathcal{P})$) is the *first* (resp., *last*) *common edge* 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 $G^{\rightsquigarrow 2e}[u, v]$. Additionally, $G_L^{\rightsquigarrow 2e}[u, v] = \mathrm{repr}_L(\mathcal{P})$ and $G_R^{\rightsquigarrow 2e}[u, v] = \mathrm{repr}_R(\mathcal{P})$. With a slight abuse of notation we also say that $G^{\rightsquigarrow 2e}[u, v] \in \mathrm{repr}(\mathcal{P})$.

▶ **Observation 1.** *Let $G = (V, E_1, E_2)$ be an edge split of a DAG, and let u and v be two arbitrary vertices in G. For $1 \le i \le n$, let $\mathcal{P}_i = \{P \subseteq E_1 : P$ is a path from u to $v_i\}$, and $\mathcal{Q}_i = \{Q \subseteq E_2 : Q$ is a path from v_i to $v\}$ (See Figure 2) 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 (1) yields immediately a polynomial time algorithm for computing $G^{\rightsquigarrow 2e}$. 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 $\mathrm{repr}_R(\mathcal{P})$.

We next define two operations, denoted as *serial* and *parallel*. Although those operations are formally defined on $E^+ = E \cup \{\top, \bot\}$, they have a more intuitive interpretation as

operations on path families. We start with the serial operation $\otimes$. For $a, b \in E^+$, 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 $a \in E^+$: $a \oplus \bot \stackrel{\text{def}}{=} a$, $\bot \oplus a \stackrel{\text{def}}{=} a$, $a \oplus \top \stackrel{\text{def}}{=} \top$, $\top \oplus a \stackrel{\text{def}}{=} \top$, and otherwise, for $e, e' \in E$:

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

We extend the definition of $\oplus$ to operate on elements of $E^+ \times E^+$, 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 edge or to preserve consistently the last common edge, under the union of path families. If for instance we preserve the first common edge, 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 edge e', but both with two different initial edges, 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 2.** *Let $G, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 1. 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* $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (e_1, \top)$ *then* $\text{repr}(\mathcal{S}) \ni e_1$*;*
(c) *if* $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\top, e_2)$ *then* $\text{repr}(\mathcal{S}) \ni e_2$*;*
(d) *if* $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (e_1, e_2)$ *then* $\text{repr}(\mathcal{S}) \ni e_1, e_2$*;*
(e) $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\top, \top)$ *iff* $\text{repr}(\mathcal{S}) = \emptyset$*.*

We now consider the special case where one side of the partition defined in Observation 1 contains only paths of length one. In particular, we say that the edge set $E' \subseteq E$ is *thin*, if there exists no triplet of vertices x, y, z such that $(x, y) \in E'$ and $(y, z) \in E'$.

▶ **Lemma 3.** *Let $G, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 1. Additionally, let E_1 be thin. Then*
(a) $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\bot, \bot)$ *iff* $\text{repr}_R(\mathcal{S}) = \bot$*;*
(b) *if* $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (e_1, \top)$ *then* $\text{repr}_R(\mathcal{S}) = e_1$*;*
(c) *if* $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\top, e_2)$ *then* $\text{repr}_R(\mathcal{S}) = e_2$*;*
(d) *if* $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (e_1, e_2)$ *then* $\text{repr}_R(\mathcal{S}) = e_2$*;*
(e) $\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\top, \top)$ *iff* $\text{repr}_R(\mathcal{S}) = \top$*.*

We define the following projection operator π: $\pi(\bot, \bot) \stackrel{\text{def}}{=} \bot$, $\pi(\top, \top) \stackrel{\text{def}}{=} \top$, $\pi(e', e) = \pi(\top, e) = \pi(e, \top) \stackrel{\text{def}}{=} e$. With this new terminology, Lemma 2 and Lemma 3 can be simply restated as follows:

▶ **Corollary 4.** *Let $G, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 1. Then*
(i) $\pi(\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i))) = \top$ *iff* $\text{repr}(\mathcal{S}) = \emptyset$*,*
(ii) $\pi(\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i))) = \bot$ *iff* $\text{repr}(\mathcal{S}) = \mathbb{U}$*, and*
(iii) $\pi(\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i))) \in \text{repr}(\mathcal{S})$ *otherwise.*

▶ **Corollary 5.** *Let $G, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation 1, and let E_1 be thin. Then* $\pi(\bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i))) = \text{repr}_R(\mathcal{S})$.

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 G are sorted according to a topological ordering. In the following lemma $\mathbf{B}$ represents a thin set of edges (i.e., the set of edges from a subset of vertices to another disjoint subset of vertices).

▶ **Lemma 6.** *Let* $\begin{bmatrix} A & B \\ 0 & C \end{bmatrix}$ *be the adjacency matrix of a DAG* $G = (V, E)$*, where* A, B *and* C *are respectively* $k \times k$*,* $k \times (n-k)$ *and* $(n-k) \times (n-k)$ *submatrices. If* $\mathbf{B}$ *is the matrix containing* $\bot$ *for every* 0 *in* B *and the appropriate* $e \in E$ *for every* 1 *in* B*, then:*

$$\begin{bmatrix} A_L^{\leadsto 2e} & A_L^{\leadsto 2e} \circ (\mathbf{B} \circ C_R^{\leadsto 2e}) \\ \bot & C_R^{\leadsto 2e} \end{bmatrix}$$

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

By Lemma 6, 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 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 efficiently path-based matrix products.

3.2 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 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 edge. The encoding of an edge is a simple 1-superimposed code: concatenation of the edge ID and complement of the edge ID. To be more precise, whenever a bitword represents an edge e in a left-closure, then it is of the form $\mathrm{ID}_e\overline{\mathrm{ID}_e}\{1\}^{4k}$; whenever a bitword represents an edge e in a right-closure, then it is of the form $\{1\}^{4k}\mathrm{ID}_e\overline{\mathrm{ID}_e}$, where $\overline{w}$ denotes the complement of bitword w.

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 result of AND is a concatenation of two bitwords of length $4k$ encoding either $\bot, \top$ or $e \in E$. We observe that $\otimes$ is calculated properly in all cases: (let $e, e_1, e_2 \in E, e_1 \neq e_2$)

1. $e \otimes \top = (e, \top)$ since $\mathrm{ID}_e\overline{\mathrm{ID}_e}\{1\}^{4k} \wedge \{1\}^{8k} = \mathrm{ID}_e\overline{\mathrm{ID}_e}\{1\}^{4k}$,
2. $\top \otimes e = (e, \top)$ since $\{1\}^{8k} \wedge \{1\}^{4k}\mathrm{ID}_e\overline{\mathrm{ID}_e} = \{1\}^{4k}\mathrm{ID}_e\overline{\mathrm{ID}_e}$,
3. $e_1 \otimes e_2 = (e_1, e_2)$ since $\mathrm{ID}_{e_1}\overline{\mathrm{ID}_{e_1}}\{1\}^{4k} \wedge \{1\}^{4k}\mathrm{ID}_{e_2}\overline{\mathrm{ID}_{e_2}} = \mathrm{ID}_{e_1}\overline{\mathrm{ID}_{e_1}}\mathrm{ID}_{e_2}\overline{\mathrm{ID}_{e_2}}$,
4. $e \otimes \bot = \top \otimes \bot = \bot \otimes \bot = \bot \otimes e = \bot \otimes \top = (\bot, \bot)$ since $\{0,1\}^{8k} \wedge \{0\}^{8k} = \{0\}^{8k}$,
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 E^{+} of the form $(e_1, e_2), (e_1, \top), (\top, e_2), (\bot, \bot), (\top, \top)$, 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 encoding. Observe that all cases, except when both bitwords include encoded edges, are managed correctly by the execution of coordinate-wise OR: (let $e \in E$)

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

We are only left to take care of operations of the form $e_1 \oplus e_2$ for $e_1, e_2 \in E$. According to the definition of the parallel operator $\oplus$, we would like $e_1 \oplus e_2 = e \in E$ iff $e_1 = e_2 = e$ and otherwise $e_1 \oplus e_2 = \top$. This special case is handled by the fact that we encode edges using 1-superimposed codes. That is, the binary representation of ID_e has the property that $\mathrm{ID}_e[1 .. 2k] = \overline{\mathrm{ID}_e[2k+1 .. 4k]}$. Moreover, the coordinate-wise OR of two encodings of edges, that is $\mathrm{X} = \mathrm{ID}_{e_1} \vee \mathrm{ID}_{e_2}$, has such property iff $e_1 = e_2$. 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 edge,
3. otherwise the result is $\top$.

The implementation of a projection operator follows trivially.

The operations needed to compute the l-th coordinate of all entries of the final path-based matrix product (before decoding of entries) can be implemented as a Boolean matrix product of the l-th coordinate of the entries of $A_L^{\rightsquigarrow_{2e}}$ and $B_R^{\rightsquigarrow_{2e}}$. All the tools developed in this section allow us to compute the 2-reachability closure for DAGs. Our recursive algorithm follows closely Lemma 6.

▶ **Lemma 7.** *Given a DAG with n vertices, its 2-reachability closure can be computed in time $\mathcal{O}(n^{\omega}\log n)$.*

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 G. But in case that $u \not\rightsquigarrow_{2e} v$ in G, finding a separating edge that appears in all paths from u to v in G can still be a challenge. We show that we can report such an edge in constant time after $\mathcal{O}(n^{\omega})$ preprocessing. The main result of this section is the following theorem.

▶ **Theorem 8.** *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 graph and it will be given in Section 4.3. Its running time will be analyzed in Lemma 13 and its correctness hinges on Lemma 15.

4.1 Reduction to two single-source problems

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

▶ **Lemma 9.** *For any pair of vertices u and v: If there is an edge $e \in E(G)$ such that $u \not\rightsquigarrow v$ in $G \setminus e$, then either $u \not\rightsquigarrow s$ in $G \setminus e$ or $s \not\rightsquigarrow v$ in $G \setminus e$.*

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 e_u the first edge on all paths in $\mathcal{P}_{u,s}$, and by e_v the last edge on all paths in $\mathcal{P}_{s,v}$. Note that there might be no edge that is on all paths of $\mathcal{P}_{u,s}$: in this case we say that e_u does not exist. If there are several edges on all paths in $\mathcal{P}_{u,s}$, then they are totally ordered, so it is clear which is the *first* edge (similarly for e_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 e_u and e_v.

▶ **Lemma 10.** *If there is some e such that $u \not\rightsquigarrow v$ in $G \setminus e$, then at least one of the following statements is true:*

- *e_u exists and $u \not\rightsquigarrow v$ in $G \setminus e_u$.*
- *e_v exists and $u \not\rightsquigarrow v$ in $G \setminus e_v$.*

Hence, in order to check whether there is an edge that separates u from v in G, it suffices to look at the reachability information in $G \setminus e_u$ (a graph which does not depend on u) and at the reachability information in $G \setminus e_v$ (a graph which does not depend on v). 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 [10], there can be at most $2n - 2$ edges 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 $G \setminus e_v$ for any query pair (u, v), and a second auxiliary graph H' whose reachability is identical to $G \setminus e_u$ for any query pair (u, v). Note that the edge 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'.

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* $G_s = (V, E, s)$ is a digraph with a distinguished *start vertex* s. We denote by $G_s^R = (V, E^R, s)$ the reverse flow graph of G_s; the graph resulted by reversing the direction of all edges $e \in E$. Vertex u is a *dominator* of a vertex v (u *dominates* v) if every path from s to v in G_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* D: u dominates v if and only if u is an ancestor of v in D. If $v \neq s$, the parent of v in D, 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 D, i.e., the vertices dominated by v. Dominators can be computed in linear time [2, 3, 7]. An edge (x, y) is a *bridge* of the flow graph G_s if all paths from s to y include (x, y).

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

▶ **Property 11** ([10]). *Let s be an arbitrary start vertex of G. An edge $e = (x, y)$ is a strong bridge of G if and only if it is a bridge of G_s or a bridge of G_s^R (or both).*

As a consequence of Property 11, all the strong bridges of the digraph G can be obtained from the bridges of the flow graphs G_s and G_s^R, and thus there can be at most $(2n-2)$ strong bridges in a digraph G. Using the linear time algorithms for computing dominators, we can thus compute all strong bridges of G in time $\mathcal{O}(m+n) \subseteq \mathcal{O}(n^\omega)$. We use the following lemma from [8] that holds for a flow graph G_s of a strongly connected digraph G.

▶ **Lemma 12** ([8]). *Let G be a strongly connected digraph and let (x, y) be a strong bridge of G. Also, let D and D^R be the dominator trees of the corresponding flow graphs G_s and G_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 D. Then there is a path from w to x in G avoiding all proper descendants of y in D. Moreover, all paths in G from w to any descendant of y in D contain the edge $(d(y), y)$.*

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

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

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 G and G^R, respectively, with the following property: Given two vertices u and v, we have that $u \rightsquigarrow_{2e} v$ in G 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 D and D^R, respectively.

The two extremal edges e_u and e_v, defined in Section 4.1, can be also defined in terms of the bridge decompositions. In particular, e_v is the bridge entering the tree T_v of the bridge decomposition of D, so $e_v = (d(r_v), r_v)$, and e_u is the reverse bridge entering the tree D_u^R of the bridge decomposition of D^R, so $e_u = (r_u^R, d^R(r_u^R))$. Hence if there exists a path from u to v avoiding each of the strong bridges e_v and e_u, then $u \rightsquigarrow_{2e} v$ in G. By Lemma 10, it is enough if H models the reachability of $G \setminus e_v$ and H' the reachability of $G \setminus e_u$. So H is responsible for answering whether u has a path to v avoiding e_v, while H' is responsible for answering whether u has a path to v avoiding e_u. Then, if any of the reachability queries in H and H' returns false, we immediately have an edge 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. In particular, the auxiliary graph $H = (V, E')$ of the flow graph $G_s = (V, E, s)$ is constructed as follows. Initially, $E' = E \setminus BR$, where BR is the set of bridges of G_s. For all bridges (p, q) of G_s do the following: For each edge $(x, y) \in E$ such that $x \in D(q), y \notin D(q)$, we add the edge (p, y)

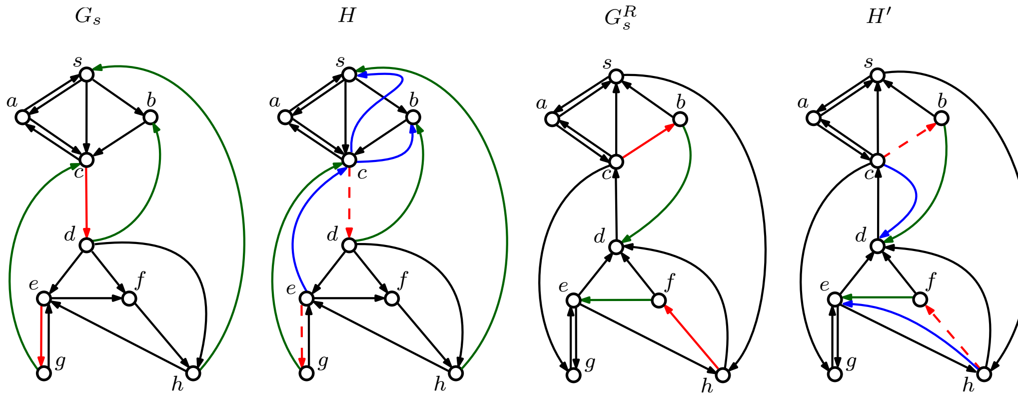


Figure 3 Auxiliary graphs H and H' which are derived from G_s and G_s^R, respectively. The deleted edges, the bridges of G_s and G_s^R, are shown in red, the newly added edges are shown in blue. The blue edges are drawn along the green edges from G_s and G_s^R which are the reason for their insertion. Here we see, that for example b is 2-reachable from e, since there are two (edge and vertex) disjoint paths (e, g, c, d, b) and (e, f, h, s, b) in G. 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 G, 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 15, consider the following example: vertex c is unreachable from b in $G \setminus (b, c)$, which we also detect as there is no c-b path in H' (even though there is a b-c path in H).

in E', i.e., we set $E' = E' \cup (p, y)$. A detailed implementation can be found in full version of the paper. Together with graph H, the algorithm outputs an array of edges ("witnesses") W, such that for each vertex $v \neq s$, $W[v] = (d(r_v), r_v)$ is a candidate separating edge 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 G in total time $\mathcal{O}(n^\omega)$ and answer 2-reachability queries in constant time, as claimed by Theorem 8.

▶ **Lemma 13.** *The auxiliary graph H can be computed in $\mathcal{O}(n^2)$ time and space.*

▶ **Lemma 14.** *For all $w \in V$, no edge $(x, y) \in E(H)$ exists with $x \notin D(r_w)$ and $y \in D(r_w)$.*

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 edge e such that $u \not\rightsquigarrow v$ in $G \setminus e$. 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 10, it is sufficient to prove the following:

▶ **Lemma 15.** *The auxiliary graphs H and H' satisfy these two conditions:*
- *If e_v exists, then $u \rightsquigarrow v$ in $G \setminus e_v$ if and only if $u \rightsquigarrow v$ in H.*
- *If e_u exists, then $u \rightsquigarrow v$ in $G \setminus e_u$ if and only if $v \rightsquigarrow u$ in H'.*

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 G can be constructed as follows: $G^{\rightsquigarrow 2e}[i, j] = \top$ if i has two edge-disjoint paths to j and $G^{\rightsquigarrow 2e}[i, j] \in E$ if there is

an edge $e \in E$ such that $i \not\rightsquigarrow j$ in $G \setminus e$. No entry of $G^{\rightsquigarrow 2e}$ contains $\perp$ since G 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 G be a general digraph. The condensation of G is the DAG resulting after the contraction of every strongly connected component of G 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 G appear consecutively in an arbitrary order, and the strongly connected components are ordered with respect to the topological ordering of the condensation of G. Moreover, we assume that we have access to a function `stronglyConnected`(u, v) that answers whether the vertices u and v are strongly connected.

The key insight is that every idea presented in Section 3 never truly used the fact that the input graph is a DAG, just the properties of an edge split, that is finding edge partition into two sets so that no vertex has incoming edge from second set and outgoing edge from first set simultaneously. If we are able to extend the definition of an edge split to a general graph in a way highlighted above, and the definitions of $\text{repr}(), \text{repr}_R()$ and $\text{repr}_L()$, then all of the results from Section 3 carry over to a general graph G. Note that given *arbitrary* path family $\mathcal{P}$, $\text{repr}_L(\mathcal{P})$ and $\text{repr}_R(\mathcal{P})$ might be ill-defined, since paths in an arbitrary path family might not share the order of common edges. However, we are only using this notation for path families containing exactly all of paths connecting a given pair of vertices in the graph: for such families, the order of common edges 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 algorithm 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.

▶ **Theorem 16.** *The 2-reachability closure for general graphs on n vertices can be computed in time $\mathcal{O}(n^{\omega} \log n)$.*

6 An application: computing all dominator trees

Let s be an arbitrary vertex of G. Recall the bridge decomposition $\mathcal{D}$ of (vertex-)dominator tree D and its tree T_v and root r_v from Section 4.2. We define the *edge-dominator tree* $\widetilde{D}$ of G with start vertex s, as the tree that results from D after contracting all vertices in each tree T_v into its root r_v. For any vertex v and edge $e = (x, y)$, e is contained in all paths in G from s to v if and only if (r_x, r_y) is in the path from s to r_v in $\widetilde{D}$. We denote by $\widetilde{d}(y)$ the parent of a vertex in $\widetilde{D}$. (Both y and $\widetilde{d}(y)$ are roots in $\mathcal{D}$.)

▶ **Theorem 17.** *We can compute all sources vertex- and edge-dominator trees from $G_R^{\rightsquigarrow 2e}$ in time $\mathcal{O}(n^2)$.*

We can preprocess each edge-dominator tree $\widetilde{D}$ in $\mathcal{O}(n)$ so as to answer ancestor-descendant relations in constant time [16]. We can also compute in $\mathcal{O}(n)$ time the number of descendants in D 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 edge $e = (x, y)$, we can test if $G \setminus e$ contains a path from s to t in constant time. This is because e is contained in all paths from s to t

in G if and only if the following conditions hold: e is a bridge of flow graph G with start vertex s (i.e., $r_y = y$ and $\widetilde{d}(y) = r_x$) and y is an ancestor of r_t in $\widetilde{D}$.
- Similarly, given a vertex s and an edge $e = (x, y)$, we can report how many vertices become unreachable from s if we delete e from G. If e is a bridge of flow graph G with start vertex s, then this number is equal to the number of descendants of y in D. Hence, we find the edge whose removal disconnects the most pairs of vertices in time $\mathcal{O}(n^2)$.

By computing all vertex-dominator trees of G, we can answer analogous queries for vertex-separators. Moreover, we can answer efficiently queries regarding junctions. A vertex s is a *junction* of vertices u and v in G, if G 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 [18] 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 G, 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 D. Hence, we 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).

Acknowledgments. We would like to thank Paolo Penna and Peter Widmayer for many useful discussions on the problem.

References

1. N. Alon, Z. Galil, O. Margalit, and M. Naor. Witnesses for boolean matrix multiplication and for shortest paths. In *Proc. of the 33rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 417–426. IEEE, 1992. doi:10.1109/SFCS.1992.267748.
2. S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–21132, 1999. doi:10.1137/S0097539797317263.
3. A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. 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.
4. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990. doi:10.1016/S0747-7171(08)80013-2.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.
6. M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *Proc. of the 12th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 129–131. IEEE, 1971. doi:10.1109/SWAT.1971.4.
7. W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013. doi:10.1016/j.jda.2013.10.003.
8. L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2-edge connectivity in directed graphs. *ACM Transactions on Algorithms*, 13(1):9:1–9:24, 2016. doi:10.1145/2968448.
9. L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures, with applications. In *Proc. of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1880–1899, 2017. doi:10.1137/1.9781611974782.123.
10. G. F. Italiano, L. Laura, and F. 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.

11 R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 595–608, New York, NY, USA, 2008. ACM. doi:10.1145/1376616.1376677.

12 V. King and G. 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.

13 F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proc. of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, New York, NY, USA, 2014. ACM. doi:10.1145/2608628.2608664.

14 T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–41, 1979. doi:10.1145/357062.357071.

15 K. Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927.

16 R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974. doi:10.1137/0203006.

17 V. Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In *Proc. of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 887–898, New York, NY, USA, 2012. ACM. doi:10.1145/2213977.2214056.

18 R. 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.