All-Pairs 2-Reachability in $O(n^\omega \log n)$ Time.

Loukas Georgiadis$^1$, Daniel Graf$^2$, Giuseppe F. Italiano$^3$, Nikos Parotsidis$^3$, and Przemysław Uznański$^2$

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

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 a 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 $O(n^\omega \log n)$ time, where $n$ is the number of vertices and $\omega$ 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 $O(n^2)$ additional time, which in turn enables us to answer various connectivity queries in $O(1)$ 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 $w$.

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 [14]. 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 digraph with $n$ vertices and $m$ edges, the former solution runs in $O(mn)$ time, while the latter runs in $O(n^\omega)$, where $\omega$ is the matrix multiplication exponent [5][16][21]. 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.,
We call such an edge (resp., vertex) \(2\) whenever there is a separating edge for a single pair of vertices. Moreover, their preprocessing time is \(O(x^3)\) and \(O(n^3)\) for reachability in a directed acyclic graph (DAG) \([15]\). Specifically, they show how to preprocess a solution of the all-pairs \(2\)-reachability problem and gives a fast algorithm for it. In recent work Georgiadis et al. \([11]\) 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 \(O(m + n)\) 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 \(O(n)\) 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 \([11]\) 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 \([15]\) gave an algorithm that can quickly answer sensitivity queries for reachability in a directed acyclic graph (DAG) \([15]\). 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 \(O(m)\) queries just to find if there is a separating edge for a single pair of vertices. Moreover, their preprocessing time is \(O(n^3)\).
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 $O(n^{(3+\omega)/2})$ 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 [20], computing shortest paths [27], and finding least common ancestors in DAGs [7] and junctions in DAGs [26]. Our algorithms can be used for constant-time queries on whether there exists a path from vertex $u$ to vertex $v$ avoiding an edge $e$ (called avoiding path). This notion is closely related to a replacement path [12, 23, 25] (for which we additionally require to be shortest in $G \setminus e$).

**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.

**Organization.** The remainder of the paper is organized as follows. After introducing some basic definitions and notation in Section 2, we present our algorithm in three steps. In Section 3 we describe our approach for acyclic graphs, Section 4 covers strongly connected graphs and Section 5 describes their combination for arbitrary digraphs. We provide a matching lower bound and extend our approach to vertex-disjointness in Sections 6 and 7, respectively. Finally, Section 8 lists several applications of our algorithm.

### 2 Preliminaries

We assume that the reader is familiar with standard graph terminology, as contained for instance in [6]. 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 $v_1, v_2, \ldots, v_k$, 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 $u \xrightarrow{*} v$ to denote that there is a path from $u$ to $v$, and $u \not\xrightarrow{*} v$ 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 \( u \sim_{2e} v \) (resp., \( u \sim_{v2e} v \)) to denote that there are two edge-disjoint (resp., internally vertex-disjoint) paths from \( u \) to \( v \), and \( u \not\sim_{2e} v \) (resp., \( u \not\sim_{v2e} v \)) otherwise. As a special case, we assume that \( v \sim_{2e} v \) (resp., \( v \sim_{v2e} v \)) 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 \( \sim_{2e} \) (no single edge separates) and \( \bot \) corresponds to \( \not\sim_{e} \) (no edge is necessary for separation, vertices are already separated).

Given a digraph \( G \), we define the 2-reachability closure of \( G \), denoted by \( G_{2e} \), to be a matrix such that:

\[
G_{2e}^{u,v} = \begin{cases} 
\top & \text{if } u \sim_{2e} v \\
\bot & \text{if } u \not\sim_{v2e} v \\
e & \text{where } e \text{ is any separating edge for } u \text{ and } v.
\end{cases}
\]

Since \( v \sim_{2e} v \) for each \( v \in V \), \( G_{2e}^{v,v} = \top \). 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 \( G_{L}^{2e} \) by replacing any separating edge with first separating edge and the 2-reachability right closure \( G_{R}^{2e} \) by replacing it with last separating edge.

Note that if there is only one edge separating \( u \) and \( v \), then \( G_{2e}^{u,v} = G_{L}^{2e} [u,v] = G_{R}^{2e} [u,v] \).

Given any 2-reachability closure matrix, one can compute efficiently the 2-reachability left and right closures 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 \( G_{2e}^{u,v} \) is either \( \top \) or \( \bot \), then \( G_{L}^{2e} [u,v] = G_{2e}^{u,v} \).

Otherwise, let \( G_{2e}^{u,v} = (x,y) \): if \( u \sim_{2e} x \) (i.e., if \( G_{2e}^{u,x} = \top \)) then \((x,y)\) is the first separating edge for \( u \) and \( v \) and \( G_{L}^{2e} [u,v] = (x,y) \); otherwise, \( u \not\sim_{2e} x \) (i.e., \( G_{2e}^{u,x} \neq \top \)) and \( G_{L}^{2e} [u,v] = G_{L}^{2e} [u,x] \). Algorithm 1 gives the pseudo-code for computing the 2-reachability left and right closures \( G_{L}^{2e} \) and \( G_{R}^{2e} \) in a total of \( O(n^2) \) worst-case time.

### 3 All-pairs 2-reachability in DAGs

In this section, we present our \( O(n^2 \log n) \) 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
Algorithm 1: Closure recovery

**Input:** \( N \times N \) matrix \( IN \) of 2-reachability closure, type \( \in \{ \text{left, right} \} \)

**Output:** Matrix \( OUT \) of 2-reachability left closure.

1. def recovery\((IN, type)\):
2. \( OUT \leftarrow N \times N \) matrix of undefined
3. for \( i \leftarrow 1 \) to \( N \) do
4. \hspace{1em} for \( j \leftarrow 1 \) to \( N \) do
5. \hspace{2em} aux\((i, j, IN, OUT, type)\)
6. return \( OUT \)

7. def aux\((u, v, IN, OUT, type)\):
8. if \( OUT[u][v] \) is undefined then
9. \hspace{1em} if \( IN[u][v] == \top \) or \( IN[u][v] == \bot \) then
10. \hspace{2em} \( OUT[u][v] \leftarrow IN[u][v] \)
11. else
12. \hspace{1em} \( (x, y) \leftarrow IN[u][v] \)
13. \hspace{1em} if \( type == \text{left} \) then
14. \hspace{2em} if \( IN[u][x] == \top \) then
15. \hspace{3em} \( OUT[u][v] \leftarrow (x, y) \)
16. \hspace{2em} else
17. \hspace{3em} \( OUT[u][v] \leftarrow \text{aux}(u, x, IN, OUT, type) \)
18. \hspace{1em} else
19. \hspace{2em} if \( IN[y][v] == \top \) then
20. \hspace{3em} \( OUT[u][v] \leftarrow (x, y) \)
21. \hspace{2em} else
22. \hspace{3em} \( OUT[u][v] \leftarrow \text{aux}(y, v, IN, OUT, type) \)
23. return \( OUT[u][v] \)

Algorithm 1 continues 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 \( E = E_1 \cup E_2 \). 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 a 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 \( v_1, v_2, \ldots, v_n \).

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 \textit{representation} $\operatorname{repr}(\mathcal{P})$:

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

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

We also define a \textit{left representation} $\operatorname{repr}_L(\mathcal{P}) \in E^+$, where $E^+ = E \cup \{\top, \bot\}$, as follows:

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

A \textit{right representation} $\operatorname{repr}_R(\mathcal{P}) \in E^+$ is defined symmetrically to $\operatorname{repr}_L(\mathcal{P})$, by replacing \textit{minimum} with \textit{maximum}. If $\operatorname{repr}_L(\mathcal{P}) \in E$ (resp., $\operatorname{repr}_R(\mathcal{P}) \in E$), we say that $\operatorname{repr}_L(\mathcal{P})$ (resp., $\operatorname{repr}_R(\mathcal{P})$) is the \textit{first} (resp., \textit{last}) \textit{common edge} in $\mathcal{P}$.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{edge_split.png}
\caption{An edge split of a DAG $G = (V, E_1, E_2)$.
}
\end{figure}

\textbf{Observation 3.1.} 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 $1 \leq i \leq n$, let $\mathcal{P}_i = \{P \subseteq E_1 : P \text{ is a path from } u \text{ to } v_i\}$, and $\mathcal{Q}_i = \{Q \subseteq E_2 : Q \text{ is a path from } v_i \text{ to } v\}$ (See Figure 2) and let $\mathcal{S}$ be the family of all paths from $u$ to $v$. Then:

$$\operatorname{repr}(\mathcal{S}) = \bigcap_{i=1}^{n} (\operatorname{repr}(\mathcal{P}_i) \cup \operatorname{repr}(\mathcal{Q}_i))$$

A straightforward application of Observation [3.1] yields immediately a polynomial time algorithm for computing $G^{\sim_{2e}}$. However, this algorithm is not very efficient, since the size of $\operatorname{repr}(\mathcal{P})$ can be as large as $(n - 1)$. In the following we will show how to obtain a faster algorithm, by replacing $\operatorname{repr}(\mathcal{P})$ with a suitable combination of $\operatorname{repr}_L(\mathcal{P})$ and $\operatorname{repr}_R(\mathcal{P})$.

\textbf{Lemma 3.2.} Let $G, \mathcal{P}_i, \mathcal{Q}_i$ and $\mathcal{S}$ be as in Observation [3.1]. If $v_i$ is such that both $u \sim v_i$ in $E_1$ and $v_i \sim v$ in $E_2$ (both $\mathcal{P}_i \neq \emptyset$ and $\mathcal{Q}_i \neq \emptyset$), then

(a) if $\operatorname{repr}_L(\mathcal{S}) = e \in E_1$ and $u \neq v_i$, then $\operatorname{repr}_L(\mathcal{P}_i) = e$;

(b) if $\operatorname{repr}_R(\mathcal{S}) = e \in E_2$ and $v_i \neq v$, then $\operatorname{repr}_R(\mathcal{Q}_i) = e$. 

Proof. We only prove (a), since (b) is completely analogous. Assume by contradiction that \( \text{repr}_L(S) = e \in E_1, u \neq v_i \) and \( v_i \leftrightarrow v \) in \( E_2 \), but \( \text{repr}_L(P_i) = e' \neq e \). Since \( \text{repr}_L(S) = e \in E_1 \), it must be \( e \in \text{repr}(P_i) \), as otherwise we would have a path \( u \leftrightarrow v_i \leftrightarrow v \) avoiding \( e \). Since \( e \in \text{repr}(P_i) \) and \( \text{repr}_L(P_i) = e' \neq e \), all paths in \( P_i \) must go first from \( u \) to edge \( e' \), then to edge \( e \) and finally to \( v_i \). However, since \( \text{repr}_L(S) = e \in E_1 \), then \( e \) is reachable from \( u \) by a path avoiding \( e' \). By definition of the edge split, this path must be fully contained in \( E_1 \), which contradicts the fact that edge \( e' \) precedes \( e \) in all paths in \( P_i \).

It is important to note that Lemma 3.2 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 \( E^+ = E \cup \{ \top, \bot \} \), 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:

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

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

\[
   e \oplus e' \overset{\text{def}}{=} \begin{cases} 
   \top & \text{if } e \neq e' \\
   e & \text{if } e = e'
   \end{cases}
\]

We extend the definition of \( \oplus \) to operate on elements of \( E^+ \times E^+ \), as follows: \( (a_1, b_1) \oplus (a_2, b_2) \overset{\text{def}}{=} (a_1 \oplus a_2, b_1 \oplus b_2) \). 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 3.3.** Let \( G, \mathcal{P}_i, \mathcal{Q}_i \) and \( S \) be as in Observation 3.1 Then:

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

(b) if \( \bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (e_1, \top) \) then \( \text{repr}(S) \ni e_1 \);

(c) if \( \bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (\top, e_2) \) then \( \text{repr}(S) \ni e_2 \);

(d) if \( \bigoplus_{i=1}^n (\text{repr}_L(\mathcal{P}_i) \otimes \text{repr}_R(\mathcal{Q}_i)) = (e_1, e_2) \) then \( \text{repr}(S) \ni e_1, e_2 \);

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

**Proof.** We proceed with a case analysis:
(a) \( \text{repr}(S) = \bigcup \) iff \( S = \emptyset \) iff \( \forall i \ (P_i = \emptyset \text{ or } Q_i = \emptyset) \) iff \( \forall i \ \text{repr}_L(P_i) \otimes \text{repr}_R(Q_i) = (\bot, \bot) \) iff \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (\bot, \bot) \).

(b) Let \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (e_1, \top) \). By definition of \( \oplus \), we must have that \( \forall i (\text{repr}_L(P_i) \in \{e_1, \bot\} \land \text{repr}_R(Q_i) = \bot) \) and there must be at least one \( j \) such that \( \text{repr}_L(P_j) = e_1 \) and \( \text{repr}_R(Q_j) \neq \bot \). Hence, any path in \( S \) must contain \( e_1 \).

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

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

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

To prove the other direction, assume that \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (\top, \top) \) and \( \text{repr}(S) \neq \emptyset \).

From case (a) we know that \( \text{repr}(S) \neq \emptyset \), and thus there exists an edge \( e \in E \) such that \( e \in \text{repr}(S) \). Without loss of generality, assume that \( e \in E_1 \). Then, it must be \( \text{repr}_L(S) = e' \) for some edge \( e' \in E_1 \). Without loss of generality, assume that \( v_1, v_2, \ldots, v_j \), are all the vertices such that simultaneously \( u \leadsto v_i \) in \( E_1 \), \( v_i \leadsto v_j \) in \( E_2 \) and \( u \neq v_i \) (there is at least one such vertex, since \( S \neq \emptyset \) and \( \text{repr}(S) \cap E_1 \neq \emptyset \)). By Lemma 3.2(a), \( e' = \text{repr}_L(P_i) \), \( 1 \leq i \leq j \). Thus:

\[
\bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = \bigoplus_{i=1}^j (e', \text{repr}_R(Q_i)) \oplus \bigoplus_{i=j+1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = \bigoplus_{i=1}^j (e', \text{repr}_R(Q_i)) \oplus \bigoplus_{i=j+1}^n (\bot, \bot) = (e', \bigoplus_{i=1}^j \text{repr}_R(Q_i)) \neq (\top, \top),
\]

where we have used that (i) if \( v_i = u \), then \( Q_i = \emptyset \), as otherwise \( u \leadsto v \) in \( E_2 \), with a path avoiding \( e \in E_1 \), and (ii) by the choice of \( j \), for \( i > j \), \( \text{repr}_L(P_i) \otimes \text{repr}_R(Q_i) = (\bot, \bot) \). Thus we have a contradiction.  \( \square \)

We now consider the special case where one side of the partition defined in Observation 3.1 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 3.4.** Let \( G, P_i, Q_i \) and \( S \) be as in Observation 3.1. Additionally, let \( E_1 \) be thin. Then

(a) \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (\bot, \bot) \) iff \( \text{repr}_R(S) = \bot \);

(b) if \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (e_1, \top) \) then \( \text{repr}_R(S) = e_1 \);

(c) if \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (\top, e_2) \) then \( \text{repr}_R(S) = e_2 \);

(d) if \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (e_1, e_2) \) then \( \text{repr}_R(S) = e_2 \);

(e) \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (\top, \top) \) iff \( \text{repr}_R(S) = \top \).

**Proof.** Since \( E_1 \) is thin, we have that for each \( i \): (i) \( \text{repr}_L(P_i) = (u, v_i) \) iff \( (u, v_i) \in E_1 \), (ii) \( \text{repr}_L(P_i) = \top \) iff \( u = v_i \) and (iii) \( \text{repr}_L(P_i) = \bot \), otherwise. We proceed with a case analysis as in Lemma 3.3.
(a) Since \( \text{repr}_R(S) = \perp \) iff \( \text{repr}(S) = \emptyset \), this case follows immediately from Lemma 3.3(a).

(b) The condition implies that there must be \( v_j \) such that \( e_1 = (u, v_j) \) and \( v_j \sim_2 v \) in \( E_2 \). Additionally, for all \( i \neq j \) such that \( v_i \neq u \), either \( (u, v_i) \notin E_1 \) or \( v_i \not\sim v \) in \( E_2 \), and for \( v_i = u \) there is \( v_i \not\sim v \) in \( E_2 \). It follows that every path in \( G \) from \( u \) to \( v \) must go through vertex \( v_j \), and since \( E_1 \) is thin, this makes \( e_1 \) the separating edge. Since \( v_j \sim_2 v \), edge \( e_1 \) is the only possible separating edge for \( u \) and \( v \). Hence, \( \text{repr}_R(S) = e_1 \).

(c) The condition implies that for any \( 1 \leq j \leq n \), exactly one of the constraints is satisfied: (i) \( u \sim v_j \) in \( E_1 \) (equivalently \( v_j = u \) or \( (u, v_j) \in E_1 \)) and \( \text{repr}_R(Q_j) = e_2 \), (ii) \( u \not\sim v_j \) in \( E_1 \) (that is, \( v_j \neq u \) and \( (u, v_j) \notin E_1 \)) or (iii) \( v_j \not\sim v \) in \( E_2 \). Additionally, unless there exists a \( j \) such that \( v_j = u \) (which would mean \( u \sim_2 v_j \)), the first constraint is satisfied for at least two distinct values of \( j \) since the conditions (ii) and (iii) are not sufficient to satisfy \( \bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i)) = (\top, e_2) \). It follows that \( \text{repr}_R(S) = e_2 \).

(d) The condition implies that for exactly one \( j \), there exists an edge \( e_1 = (u, v_j) \) and \( v_j \sim v \) in \( E_2 \), and \( \text{repr}_R(Q_j) = e_2 \). Additionally, for every \( i \neq j \), either \( u \not\sim v_j \) in \( E_1 \) (that is, \( v_j \neq u \) and \( (u, v_j) \notin E_1 \)) or \( v_j \not\sim v \) in \( E_2 \) (since otherwise there would be a path avoiding \( e_1 \)). Similarly to case (c), it follows that \( \text{repr}_R(S) = e_2 \).

(e) Since \( \text{repr}_L(S) = \top \) iff \( \text{repr}(S) = \emptyset \), this case follows immediately from Lemma 3.3(e).

One could prove a symmetric version of Lemma 3.4 with \( E_2 \) being thin. However, in the remainder of the paper we stick with Lemma 3.4, namely, we choose a partition with a thin left side and thus break case (d) of Lemma 3.4 in favor of the rightmost edge (instead of the leftmost edge, as it would be in the symmetric version). Consistently, we define the following projection operator \( \pi \): \( \pi(\perp, \perp) \overset{\text{def}}{=} \perp, \pi(\top, \top) \overset{\text{def}}{=} \top, \pi(e', e) = \pi(\top, e) = \pi(e, \top) \overset{\text{def}}{=} e \). With this new terminology, Lemma 3.3 and Lemma 3.4 can be simply restated as follows:

Corollary 3.5. Let \( G, P_i, Q_i \) and \( S \) be as in Observation 3.1. Then

(i) \( \pi(\bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i))) = \top \) iff \( \text{repr}(S) = \emptyset \),

(ii) \( \pi(\bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i))) = \perp \) iff \( \text{repr}(S) = \emptyset \), and

(iii) \( \pi(\bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i))) \in \text{repr}(S) \) otherwise.

Corollary 3.6. Let \( G, P_i, Q_i \) and \( S \) be as in Observation 3.1 and let \( E_1 \) be thin. Then \( \pi(\bigoplus_{i=1}^n (\text{repr}_L(P_i) \otimes \text{repr}_R(Q_i))) = \text{repr}(S) \).

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

Lemma 3.7. 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 \( B \) is the matrix containing \( \perp \) for every \( 0 \) in \( B \) and the appropriate \( e \in E \) for every \( 1 \) in \( B \), then:

\[
\begin{bmatrix}
A_L^{\sim_2 e} & A_L^{\sim_2 e} \circ (B \circ C_R^{\sim_2 e}) \\
\perp & C_R^{\sim_2 e}
\end{bmatrix}
\]
is a 2-reaching closure of $G$ (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 $A, B$ and $C$ correspond respectively to all edges from $V_1$ to $V_1$, to all edges from $V_1$ to $V_2$ and to all edges from $V_2$ to $V_2$. We refer to the edge sets represented by those matrices as $E_A, E_B$ and $E_C$. As a consequence of the fact that there are no edges from $V_2$ to $V_1$, any path from $V_1$ to $V_1$ can use only edges from $E_A$, and any path from $V_2$ to $V_2$ can use only edges from $E_C$. Thus:

\[(V, E_A)_R^{2e} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix}_L \Rightarrow \begin{bmatrix} A & \perp \\ \perp & I \end{bmatrix},\]

\[(V, E_B)_R^{2e} = \begin{bmatrix} 0 & \perp \\ 0 & \perp \end{bmatrix}_L \Rightarrow \begin{bmatrix} I & B \\ \perp & I \end{bmatrix},\]

\[(V, E_C)_R^{2e} = \begin{bmatrix} 0 & \perp \\ 0 & \perp \end{bmatrix}_R \Rightarrow \begin{bmatrix} I & \perp \\ \perp & C^R \end{bmatrix},\]

where

\[I \overset{\text{def}}{=} \begin{bmatrix} \top & \cdots & \top \\ \top & \cdots & \top \\ \vdots & \ddots & \vdots \\ \top & \cdots & \top \end{bmatrix}\]

By Corollary 3.6 (since $E_B$ is thin) and by definition of path-based matrix product:

\[(V, E_B \cup E_C)_R^{2e} = (V, E_B)_L^{2e} \circ (V, E_C)_R^{2e} = \begin{bmatrix} I & \perp \\ \perp & I \end{bmatrix} \circ \begin{bmatrix} I & \perp \\ \perp & C^R \end{bmatrix} = \begin{bmatrix} I & B \circ C^R \end{bmatrix}.\]

Finally, by Corollary 3.5

\[(V, E_A)_L^{2e} \circ (V, E_B \cup E_C)_R^{2e} = \begin{bmatrix} A^L \perp \\ \perp & I \end{bmatrix} \circ \begin{bmatrix} I & \perp \\ \perp & C^R \end{bmatrix} = \begin{bmatrix} A^L \perp \\ \perp & C^R \end{bmatrix} \circ (B \circ C^R)\]

is a 2-reaching closure of $G$. 

By Lemma 3.7 the 2-reaching closure can be computed by performing path-based matrix products on the left and right 2-reaching closures of smaller matrices. This gives immediately a recursive algorithm for computing the 2-reaching closure: indeed, as already shown in Section 2, one can compute the left and right 2-reaching closures in $O(n^2)$ time from any 2-reaching 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 $O(\log n)$,
The implementation of this encoding is given in Algorithm 2.

Algorithm 2: Left- and right-side encoding

Input: Matrix $IN$ of dimension $N \times M$ with elements in $E^{+}$; type $\in \{\text{left}, \text{right}\}$.

Output: Matrix $OUT$ of dimension $N \times M$ consisting of bitwords.

1. def encode($IN$, $type$):
2.   forall $IN[i][j]$ do
3.     if $IN[i][j] == \perp$ then
4.         $OUT[i][j] \leftarrow \{0\}^{sk}$
5.     else if $IN[i][j] == \top$ then
6.         $OUT[i][j] \leftarrow \{1\}^{sk}$
7.     else
8.         $\triangleright IN[i][j] \in E$
9.         $(x, y) \leftarrow IN[i][j]$  
10.        $b_1b_2\ldots b_k \leftarrow$ binary encoding of $x$
11.        $c_1c_2\ldots c_k \leftarrow$ binary encoding of $y$
12.        if $type == \text{left}$ then
13.            $OUT[i][j] \leftarrow b_1b_2\ldots b_k c_1c_2\ldots c_k b_1b_2\ldots b_k \bar{c}_1\bar{c}_2\ldots \bar{c}_k \{1\}^{4k}$
14.        else
15.            $OUT[i][j] \leftarrow \{1\}^{4k}b_1b_2\ldots b_k c_1c_2\ldots c_k b_1b_2\ldots b_k \bar{c}_1\bar{c}_2\ldots \bar{c}_k$
16.   return $OUT$

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 $\perp$ as $\{0\}^{sk}$, left-side and right-side $\top$ as $\{1\}^{sk}$. 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: the concatenation of the edge ID and the 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 $ID_e \overline{ID_e} \{1\}^{4k}$; whenever a bitword represents an edge $e$ in a right-closure, then it is of the form $\{1\}^{4k}ID_e \overline{ID_e}$, where $\overline{w}$ denotes the complement of bitword $w$. The implementation of this encoding is given in Algorithm 2.

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 AND is a concatenation of two bitwords of length $4k$ encoding either $\perp, \top$ 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 $ID_e \overline{ID_e} \{1\}^{4k} \land \{1\}^{8k} = ID_e \overline{ID_e} \{1\}^{4k}$
2. $\top \otimes e = (e, \top)$ since $\{1\}^{8k} \land \{1\}^{4k}ID_e \overline{ID_e} = \{1\}^{4k}ID_e \overline{ID_e}$
3. $e_1 \otimes e_2 = (e_1, e_2)$ since $ID_{e_1} \overline{ID_{e_1}} \{1\}^{4k} \land \{1\}^{4k}ID_{e_2} \overline{ID_{e_2}} = ID_{e_1} \overline{ID_{e_1}} ID_{e_2} \overline{ID_{e_2}}$
4. $e \otimes \perp = \top \otimes \perp = \perp \otimes \perp = \perp \otimes e = \perp \otimes \top = (\perp, \top)$ since $\{0,1\}^{8k} \land \{0\}^{8k} = \{0\}^{8k}$
Algorithm 3: Decoding

**Input:** Matrix $IN$ of dimension $N \times M$ consisting of bitwords.

**Output:** Matrix $OUT$ of dimension $N \times M$ consisting of elements in $E^+$.

1. Define $decode(IN)$:
   2. For all $IN[i][j]$ do:
      3. $b_1 b_2 \ldots b_{8k} \leftarrow IN[i][j]$
      4. If $b_1 b_2 \ldots b_{8k} = \{0\}^{8k}$ then:
         5. $OUT[i][j] \leftarrow \bot$
      6. Else if $b_{4k+1} b_{4k+2} \ldots b_{8k} = b_{8k+1} \ldots b_{8k}$ then:
         7. $x \leftarrow$ binary decoding of $b_{4k+1} b_{4k+2} \ldots b_{8k}$
         8. $y \leftarrow$ binary decoding of $b_{5k+1} b_{5k+2} \ldots b_{8k}$
         9. $OUT[i][j] \leftarrow (x, y)$
      10. Else if $b_1 b_2 \ldots b_{2k} = b_{2k+1} b_{2k+2} \ldots b_{4k}$ then:
        11. $x \leftarrow$ binary decoding of $b_1 b_2 \ldots b_k$
        12. $y \leftarrow$ binary decoding of $b_{k+1} b_{k+2} \ldots b_{2k}$
        13. $OUT[i][j] \leftarrow (x, y)$
    14. Else:
        15. $OUT[i][j] \leftarrow \top$
    16. Return $OUT$

5. $\top \otimes \top = (\top, \top)$ since $\{1\}^{8k} \land \{1\}^{8k} = \{1\}^{8k}$

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 of 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 $\{0\}^{4k} \lor \{0\}^{4k} = \{0\}^{4k}$
2. $\bot \oplus e = e \lor \bot = e$ since $ID_e ID_e \lor \{0\}^{4k} = ID_e ID_e$
3. $\bot \oplus \top = \top \lor \bot = \top$ since $\{1\}^{4k} \lor \{0\}^{4k} = \{1\}^{4k}$
4. $e \oplus \top = \top \lor e = \top$ since $ID_e ID_e \lor \{1\}^{4k} = \{1\}^{4k}$

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 $ID_e[1 .. 2k] = ID_e[2k + 1 .. 4k]$. Moreover, the coordinate-wise OR of two encodings of edges, that is $X = ID_{e_1} \lor ID_{e_2}$, has this 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 $X = X[1 .. 2k]X[2k + 1 .. 4k]$):

1. $X = \{0\}^{4k}$, then the result is $\bot$, 
2. $X = \{1\}^{4k}$, then the result is $\top$, 
3. $X = \{0\}^{2k} \land \{1\}^{2k}$, then the result is $\bot$, 
4. $X = \{0\}^{2k} \lor \{1\}^{2k}$, then the result is $\top$, 
5. $X = \{0\}^{2k} \land \{0\}^{2k}$, then the result is $\bot$, 
6. $X = \{0\}^{2k} \lor \{0\}^{2k}$, then the result is $\top$, 
7. $X = \{1\}^{2k} \land \{0\}^{2k}$, then the result is $\bot$, 
8. $X = \{1\}^{2k} \lor \{0\}^{2k}$, then the result is $\top$, 
9. $X = \{1\}^{2k} \land \{1\}^{2k}$, then the result is $\top$, 
10. $X = \{1\}^{2k} \lor \{1\}^{2k}$, then the result is $\bot$. 

11. $X = \{1\}^{4k}$, then the result is $\bot$, 
12. $X = \{0\}^{4k}$, then the result is $\top$, 
13. $X = \{0\}^{2k} \land \{1\}^{2k}$, then the result is $\bot$, 
14. $X = \{0\}^{2k} \lor \{1\}^{2k}$, then the result is $\top$, 
15. $X = \{1\}^{2k} \land \{0\}^{2k}$, then the result is $\bot$, 
16. $X = \{1\}^{2k} \lor \{0\}^{2k}$, then the result is $\top$, 
17. $X = \{1\}^{4k}$, then the result is $\bot$, 
18. $X = \{0\}^{4k}$, then the result is $\top$, 
19. $X = \{0\}^{2k} \land \{1\}^{2k}$, then the result is $\bot$, 
20. $X = \{0\}^{2k} \lor \{1\}^{2k}$, then the result is $\top$, 
21. $X = \{1\}^{2k} \land \{0\}^{2k}$, then the result is $\bot$, 
22. $X = \{1\}^{2k} \lor \{0\}^{2k}$, then the result is $\top$, 
23. $X = \{1\}^{4k}$, then the result is $\bot$, 
24. $X = \{0\}^{4k}$, then the result is $\top$, 
25. $X = \{0\}^{2k} \land \{1\}^{2k}$, then the result is $\bot$, 
26. $X = \{0\}^{2k} \lor \{1\}^{2k}$, then the result is $\top$, 
27. $X = \{1\}^{2k} \land \{0\}^{2k}$, then the result is $\bot$, 
28. $X = \{1\}^{2k} \lor \{0\}^{2k}$, then the result is $\top$. 

12
Algorithm 4: Path-based matrix product

**Input:** Matrices $A$ and $B$ of compatible dimension.

**Output:** Matrix being a path-based product of inputs.

```python
1 def mul(A, B):
2     return decode(encode(A, left) \cdot encode(B, right))

\[ \Delta \text{ here } \cdot \text{ denotes coordinate-wise Boolean matrix multiplication} \]
```

2. $X[1 .. 2k] = X[2k+1 .. 4k]$, then $X$ is the encoding of the resulting edge,

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 $(A_L^{\to 2e} \circ B_R^{\to 2e})[i,j] \overset{\text{def}}{=} \text{decode} \left( \bigvee_k \left( \text{encode}(A_L^{\to 2e}[i,k], \text{left}) \land \text{encode}(B_R^{\to 2e}[k,j], \text{right}) \right) \right)$, where the pseudocode for `decode` is provided in Algorithm 3.

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 appropriate bitword matrices. That is, we apply `encode` to $A_L^{\to 2e}$ and $B_R^{\to 2e}$, 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 3. This is illustrated in Algorithm 4.

All the tools developed in this section allow us to compute the 2-reachability closure for DAGs. Our recursive algorithm follows closely Lemma 3.7, and its implementation in pseudocode is given as Algorithm 5. Since we implemented the right-side version of the projection, we have only to be careful to perform first the right multiplication before the left multiplication.

**Lemma 3.8.** Given a DAG with $n$ vertices, Algorithm 5 computes its 2-reachability closure in time $O(n^\omega \log n)$.

**Proof.** Algorithm 4 computes the path-based matrix product of matrices with every dimension bounded by $n$, if the initial graph size was $n_0$, in time $O(n^\omega \log n_0)$, as it needs to compute $O(\log n_0)$ Boolean matrix products, one for each coordinate of the stored bitwords. Closures are computed in time $O(n^2)$. The recursion that captures the runtime of Algorithm 5 is thus given by the formula $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n^\omega \log n_0)$ which is satisfied by setting $T(n) = O(n^\omega \log n_0)$. The bound follows.

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 $u \Rightarrow v$ in G. But in case that $u \not\Rightarrow 2e v$ 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 $O(n^\omega)$ preprocessing. The main result of this section is the following theorem.

**Theorem 4.1.** The 2-reachability closure of a strongly connected graph can be computed in time $O(n^\omega)$.

Our construction is based on the notion of auxiliary graph and it will be given in Section 4.3. A detailed implementation will be provided in Algorithm 6. Its running time will be analyzed in Lemma 4.7 and its correctness hinges on Lemma 4.9.
then either

is true:

enough to derive an efficient algorithm, since we would have still to look at as many as

reachability information in

look at the reachability information in

goto through

lies not only on any path from

but also on any path from

e.

s

lies on every path from

u

s

1

Lemma 4.2. For any pair of vertices u and v: If there is an edge e ∈ E(G) such that u ̸⇝ v in G \ e, then either u ̸⇝ s in G \ e or s ̸⇝ v in G \ e.

Let P_{u,s} be the family of all paths from u to s and let 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 P_{u,s}, and by e_v the last edge on all paths in P_{s,v}. Note that there might be no edge that is on all paths of P_{u,s}: in this case we say that e_u does not exist. If there are several edges on all paths in P_{u,s}, then they are totally ordered, so it is clear which is the first edge (similarly for e_v and 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 4.3. If there is some e such that u ̸⇝ v in G \ e, then at least one of the following statements is true:

• e_u exists and u ̸⇝ v in G \ e_u.

• e_v exists and u ̸⇝ v in G \ e_v.

Proof. If e = e_u or e = e_v, the claim is trivial. Otherwise, by Lemma 4.2, we know that u ̸⇝ s or s ̸⇝ v in G \ e. Let us assume that u ̸⇝ s (See Figure 3). So e lies not only on any path from u to v but also on any path from u to s. As e_u is the first common edge of every path from u to s, e_u also lies on every path from u to e. As all paths from u to v have to go through e, they also have to go through e_u and hence u ̸⇝ v in G \ e_u. If s ̸⇝ v in G \ e, we can show that u ̸⇝ v in G \ e_v by the same extremality argument for 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 \ e_u (a graph which does not depend on u) and at the reachability information in G \ 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

Algorithm 5: 2-reachability closure for DAGs

Input: Matrix G of dimension N × N, G is a DAG with topological order 1, 2, . . . , n.

Output: 2-reachability closure of G.

1 def closureDAG(G):
2     if N == 1 then
3         return [T]
4     N' ← ⌈N/2⌉
5     A ← G[1 .. N'][1 .. N']
6     B ← G[1 .. N'][N' + 1 .. N], 0’s replaced with ⊥ and 1’s with edge labels
7     C ← G[(N' + 1) .. N][(N' + 1) .. N]
8     A' ← recovery(closureDAG(A), left)
9     C' ← recovery(closureDAG(C), right)
10     return [A' mul(A', mul(B, C'))
               0
               C']

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 4.2. For any pair of vertices u and v: If there is an edge e ∈ E(G) such that u ̸⇝ v in G \ e, then either u ̸⇝ s in G \ e or s ̸⇝ v in G \ e.

Let P_{u,s} be the family of all paths from u to s and let 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 P_{u,s}, and by e_v the last edge on all paths in P_{s,v}. Note that there might be no edge that is on all paths of P_{u,s}: in this case we say that e_u does not exist. If there are several edges on all paths in P_{u,s}, then they are totally ordered, so it is clear which is the first edge (similarly for e_v and 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 4.3. If there is some e such that u ̸⇝ v in G \ e, then at least one of the following statements is true:

• e_u exists and u ̸⇝ v in G \ e_u.

• e_v exists and u ̸⇝ v in G \ e_v.

Proof. If e = e_u or e = e_v, the claim is trivial. Otherwise, by Lemma 4.2, we know that u ̸⇝ s or s ̸⇝ v in G \ e. Let us assume that u ̸⇝ s (See Figure 3). So e lies not only on any path from u to v but also on any path from u to s. As e_u is the first common edge of every path from u to s, e_u also lies on every path from u to e. As all paths from u to v have to go through e, they also have to go through e_u and hence u ̸⇝ v in G \ e_u. If s ̸⇝ v in G \ e, we can show that u ̸⇝ v in G \ e_v by the same extremality argument for 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 \ e_u (a graph which does not depend on u) and at the reachability information in G \ 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

Algorithm 5: 2-reachability closure for DAGs

Input: Matrix G of dimension N × N, G is a DAG with topological order 1, 2, . . . , n.

Output: 2-reachability closure of G.

1 def closureDAG(G):
2     if N == 1 then
3         return [T]
4     N' ← ⌈N/2⌉
5     A ← G[1 .. N'][1 .. N']
6     B ← G[1 .. N'][N' + 1 .. N], 0’s replaced with ⊥ and 1’s with edge labels
7     C ← G[(N' + 1) .. N][(N' + 1) .. N]
8     A' ← recovery(closureDAG(A), left)
9     C' ← recovery(closureDAG(C), right)
10     return [A' mul(A', mul(B, C'))
               0
               C']

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 4.2. For any pair of vertices u and v: If there is an edge e ∈ E(G) such that u ̸⇝ v in G \ e, then either u ̸⇝ s in G \ e or s ̸⇝ v in G \ e.

Let P_{u,s} be the family of all paths from u to s and let 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 P_{u,s}, and by e_v the last edge on all paths in P_{s,v}. Note that there might be no edge that is on all paths of P_{u,s}: in this case we say that e_u does not exist. If there are several edges on all paths in P_{u,s}, then they are totally ordered, so it is clear which is the first edge (similarly for e_v and 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 4.3. If there is some e such that u ̸⇝ v in G \ e, then at least one of the following statements is true:

• e_u exists and u ̸⇝ v in G \ e_u.

• e_v exists and u ̸⇝ v in G \ e_v.

Proof. If e = e_u or e = e_v, the claim is trivial. Otherwise, by Lemma 4.2, we know that u ̸⇝ s or s ̸⇝ v in G \ e. Let us assume that u ̸⇝ s (See Figure 3). So e lies not only on any path from u to v but also on any path from u to s. As e_u is the first common edge of every path from u to s, e_u also lies on every path from u to e. As all paths from u to v have to go through e, they also have to go through e_u and hence u ̸⇝ v in G \ e_u. If s ̸⇝ v in G \ e, we can show that u ̸⇝ v in G \ e_v by the same extremality argument for 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 \ e_u (a graph which does not depend on u) and at the reachability information in G \ 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 [13], 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 $D$, see Figure 4 and Figure 5 for examples. 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 $v$. Lengauer and Tarjan [17] presented an algorithm for computing dominators in $\mathcal{O}(m\alpha(m, n))$ time for a flow graph with $n$ vertices and $m$ edges, where $\alpha$ is a functional inverse of Ackermann’s function [22]. Dominators can be computed in linear time [2, 4, 9]. 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 $G^R_s$.

**Property 4.4.** ([13]) 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 $G^R_s$ (or both).

As a consequence of Property 4.4 all the strong bridges of the digraph $G$ can be obtained from the bridges of the flow graphs $G_s$ and $G^R_s$, 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 [10] that holds for a flow graph $G_s$ of a strongly connected digraph $G$. 

![Figure 3: Illustration of the first case of Lemma 4.3.](image)
Figure 4: A flow graph and its dominator tree. Edges marked in red are bridges.

Lemma 4.5. 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 \( G^R_s \), 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 \( G \) that does not contain any proper descendant of \( y \) in \( D \). Moreover, all simple paths in \( G \) from \( w \) to any descendant of \( y \) in \( D \) must 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 \( G \) that does not contain any proper descendant of \( x \) in \( D^R \). Moreover, all simple paths in \( G \) from any descendant of \( x \) in \( D^R \) to \( w \) must 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 \( G^R_s \), we obtain the bridge decomposition of \( D \) and \( D^R \) into forests \( D \) and \( D^R \). Throughout this section, we denote by \( T_v \) (resp., \( T^R_v \)) the tree in \( D \) (resp., \( D^R \)) containing vertex \( v \), and by \( r_v \) (resp., \( r^R_v \)) the root of \( T_v \) (resp., \( T^R_v \)). 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 \( u \sim_2 v \) in \( G \) if and only \( u \sim v \) in \( H \) and \( v \sim u \) 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 \( D^R_u \) of the bridge decomposition of \( D^R \), so \( e_u = (r^R_u, d^R(r^R_u)) \). Hence if there exists a path from \( u \) to \( v \) avoiding each of the strong bridges \( e_v \) and \( e_u \), then \( u \sim_2 v \) in \( G \). By Lemma 4.3, 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
For all bridges we update we compute which reachability queries can be answered in constant time. Thus, we can preprocess a strongly connected digraph \( G \). Clearly, after these steps the set \( R \) contains only the desired endpoints.

We next show to compute the auxiliary graphs \( H \) and \( H' \) in \( O(n^2) \) 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 \((p, y) \) in \( E' \), i.e., we set \( E' = E' \cup (p, y) \). A detailed implementation is provided in Algorithm 6. 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 \( O(n^\omega) \) time, after which reachability queries can be answered in constant time. Thus, we can preprocess a strongly connected digraph \( G \) in total time \( O(n^\omega) \) and answer 2-reachability queries in constant time, as claimed by Theorem 4.1.

**Definition 4.6** (Auxiliary graph construction). 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 \((p, y) \) in \( E' \), i.e., we set \( E' = E' \cup (p, y) \).

**Lemma 4.7.** The auxiliary graph \( H \) can be computed in \( O(n^2) \) time and space.

**Proof.** For each root \( r \) of a tree \( T_r \in \mathcal{D} \), we maintain a set \( R(r) \subseteq V \), initially set to \( \emptyset \). The value \( R(r) \) contains all such endpoints \( y \) of edge \((x, y) \) 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 tree steps. First, for each bridge \((p, q)\) of \( G_s \) such that \( p \in T_r \), we update \( R(r) \) by setting \( R(r) = R(r) \cup R(q) \). Second, for every edge \((x, y) \) 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 edge \((x, y) \) such that \( x \in D(r) \) and \( y \notin D(r) \). Clearly, after these steps the set \( R(r) \) contains only the desired endpoints.
Algorithm 6: 2-reachability closure in strongly connected graphs

**Input:** Strongly connected graph $G$ on $N$ vertices.

**Output:** 2-reachability closure of $G$.

1. `def closureSCC(G):
2.     s ← arbitrary vertex of $G$
3.     $H, W \leftarrow$ auxiliaryGraph($G, s$
4.     $H', W' \leftarrow$ auxiliaryGraph($G^R, s$
5.     $OUT \leftarrow N \times N$ matrix
6.     forall $OUT[i][j]$ do
7.         if $(i, j) \notin H$ then
8.             $OUT[i][j] \leftarrow W[j$
9.         else if $(j, i) \notin H'$ then
10.            $OUT[i][j] \leftarrow W'[i]^R$
11.        else
12.            $OUT[i][j] \leftarrow \top$
13.        return $OUT$
14. `def auxiliaryGraph($G, s$):
15.     $H \leftarrow G$
16.     $W, R \leftarrow$ arrays of size $N$
17.     $D \leftarrow$ dominator tree of $G_s$
18.     for tree $T$, rooted at $r$, in bottom-up order of bridge decomposition of $D$ do
19.         $R[r] \leftarrow \bigcup \{R[r'] : d(r') \in V(T)\}$ \hspace{1cm} \triangleright r' is root of children component of $T$
20.         $D[r] \leftarrow V(T) \cup \bigcup \{ D[r'] : d(r') \in V(T)\}$
21.         for $(x, y) \in V(T) \times V(G)$ do
22.             if $(x, y) \in G$ then
23.                 $R[r] \leftarrow R[r] \cup \{y\}$
24.                 $R[r] \leftarrow R[r] \setminus D[r]$
25.         if $s \notin T$ then
26.             $p \leftarrow d(r)$ \hspace{1cm} \triangleright (p, r) is a bridge connecting parent of $T$ to $T$
27.             for $y \in R[r]$ do
28.                 $H \leftarrow H \cup (p, y)$
29.             for $x \in V(T)$ do
30.                 $W[x] \leftarrow (p, r)$
31.             $H \leftarrow H \setminus (p, r)$
32.     return transitiveClosure($H, W$)
Lemma 4.9. Let $G_s$ and $G'^R_s$ be the auxiliary graphs derived from $G_s$ and $G'^R_s$, respectively. The deleted edges, the bridges of $G_s$ and $G'^R_s$, 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 $G'^R_s$ 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 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 4.9, 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 $H$).

Note that we actually wish to insert edges 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 edge $(d(r), y)$ for every $y \in R(r)$ (notice that the outgoing edges of $d(r)$ in $G$, 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 $O(n^2)$ time. We spend $O(n^2)$ time in the third step, since we visit each vertex at most one. Since we traverse every edge only once, the second step takes $O(n + m)$ in total. The bound follows.

Lemma 4.8. For all $w \in V$, no edge $(x, y) \in E(H)$ exists with $x \notin D(r_w)$ and $y \in D(r_w)$.

Proof. Assume, by contradiction, that there is an edge $(x, y) \in E(H)$ such that $x \notin D(r_w)$ and $y \in D(r_w)$. Since $(d(r_w), r_w)$ is a strong bridge in $G_s$, $(x, y)$ does not exist in $G$ (by Lemma 4.5). Hence, by construction, there is an edge $(z, y) \in E(G)$ where $z \in D(x) \setminus x$ and $y \notin D(x)$. Therefore, $x$ cannot be an ancestor of $w$ in $D$, 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 $G$ by Lemma 4.5.

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 $u \sim v$ in $G \setminus e$. We can answer this query in constant time by answering the queries $u \sim v$ in $H$ and $v \sim u$ in $H'$. Given Lemma 4.3, it is sufficient to prove the following:

Lemma 4.9. The auxiliary graphs $H$ and $H'$ satisfy these two conditions:

- If $e_v$ exists, then $u \sim v$ in $G \setminus e_v$ if and only if $u \sim v$ in $H$.
- If $e_u$ exists, then $u \sim v$ in $G \setminus e_u$ if and only if $v \sim u$ in $H'$.
We prove the lemma in two separate steps, one for each direction of the two equivalences.

**Lemma 4.10.** If u⇝v in H then u⇝v in G \ e_v (and if v⇝u in H' then u⇝v in G \ e_u).

**Proof.** By Lemma 4.8 there are no edges 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 edges from E(G[D(r_v)]), then P is also a valid path from u to v in G[D(r_v)] and thus also in G \ e_v (recall that e_v = (d(r_v), r_v)) and we are done.

Otherwise, we iteratively substitute auxiliary edges of P, with paths in G \ e_v, so that in the end, P is fully contained within G \ e_v. Let e^* = (x^*, y^*) be the first edge of P such that e^* ∉ E(G[D(r_v)]), i.e. an auxiliary edge of H. By e^o = (x^o, y^o), we denote the original edge for which we inserted e^* into H. Then x^o ∈ D(x^*) \ x^* and y^o = 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 \ D(r_v) (otherwise, if all paths contained such a vertex w then s would have a path to x^o in G avoiding x^* by Lemma 4.5), it follows that x^* has a path P_{x^*x^o} to x^o in G[D(r_v)]. If we now replace e^* in P by P_{x^*x^o} · e^o, then P contains a path from u to y^* containing only edges in G \ e_v. We repeat this argument as long as P contains auxiliary edges and get a path from u to v in G \ e_v.

The statement for H' and G \ e_u can be shown with completely analogous arguments.

**Lemma 4.11.** If u⇝v in G \ e_v then u⇝v in H (and if u⇝v in G \ e_u then v⇝u in H').

**Proof.** By Lemma 4.5 (a), e_v is the only edge in G entering D(r_v). So any path P from u to v in G \ e_v can only use edges in E(G[D(r_v)]). If P only contains edges in H, we are done. Otherwise, let e^* = (x^*, y^*) be the first edge of P that is in E(G[D(r_v)]) but not in H, hence e^* is a bridge of G_x. Let z^* be the first vertex on P after e^* that is not a descendant of y^* in D. Such a vertex z^* exists since P ends at v but v is not a descendant of y^* (recall that e^* is a bridge and lies within G[D(r_v)]). Thus, we can replace the subpath of P between e^* and z^* (including e^*) 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 G_x and get a path from u to v in H.

The statement for G \ e_u and H' can be shown with completely analogous arguments.

## 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: \( G \rightarrow_{2c} [i, j] = 1 \) if i has two edge-disjoint paths to j and \( G \rightarrow_{2c} [i, j] \in E \) if there is an edge e ∈ E such that i ⇝ j in G \ e. No entry of \( G \rightarrow_{2c} \) contains \( \perp \) since G is strongly connected. After \( \mathcal{O}(n^2) \) time preprocessing all the above queries can be answered in constant time. Therefore, the 2-reachability closure can be computed in \( \mathcal{O}(n^2) \) 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.
Theorem 5.1. Algorithm 7 computes

Algorithm 7 computes

we are able to extend the definition of an edge split to a general graph in a way highlighted above,
asymptotic running time as Algorithm 5.

\[ = B \]

Proof. The algorithm correctness follows from the correctness of Algorithm 5 and the fact that Algorithm 7 separates a large strongly connected component, on which we can compute the larger instance is a strongly connected component, or

separation of the set of vertices into two subsets, then one of the following properties holds:

then the instance is (roughly) equally divided into two instances. Otherwise, if there is no balanced

no pair across the two sets that is strongly connected. If such a balanced separation can be found,

the set of vertices, with respect to their fixed precomputed order, into two sets such that there is

DAGs, as follows. At each recursive call, the algorithm attempts to find a balanced separation of

a given pair of vertices in the graph: for such families, the order of common edges is shared.

However, we are only using this notation for path families containing exactly all of paths connecting

ill-defined, since paths in an arbitrary path family might not share the order of common edges.

to a general graph

and the definitions of

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 \( O(n^\omega) \) time. We provide pseudocode for this in Algorithm 7.

The key insight is that every idea presented in Section 5 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 \( \text{repr}(\cdot) \), \( \text{repr}_R(\cdot) \) and \( \text{repr}_L(\cdot) \), 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 \( O(n^\omega) \) time. We provide pseudocode for this in Algorithm 7.

Theorem 5.1. Algorithm 7 computes 2-reachability closure of graph on \( n \) vertices in time \( O(n^{\omega} \log n) \).

Proof. The algorithm correctness follows from the correctness of Algorithm 5 and the fact that Algorithm 7 separates the input matrix \( G \) with dimensions \( N \times N \) into submatrices \( A = G[1 \ldots N'] [1 \ldots N'] \), \( B = G[1 \ldots N'] [(N' + 1) \ldots N] \) and \( C = G[(N' + 1) \ldots N][N' + 1) \ldots N] \), and such that \( \forall i \in [1, N'], j \in [N' + 1, N] : G[j, i] = 0 \) as required by Lemma 3.7. Now we show that Algorithm 7 has the same asymptotic running time as Algorithm 5.

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

Algorithm 7: 2-reachability closure in general graphs

Input: Matrix \( G \) of dimension \( N \times N \), with vertices ordered w.r.t. some fixed topological

order of strongly connected components

Output: 2-reachability closure of \( G \).

1 def closure(G):
2     if \( N == 1 \) then
3         return \( [\top] \)
4     else if \( \forall i, j \in \{1, \ldots, N\} \) stronglyConnected\( (i, j) == \text{true} \) then
5         return closureSCC(G)
6     \( I = \{i : 1 \leq i < N, \text{stronglyConnected}(i, i + 1) == \text{false}\} \)
7     \( N' \leftarrow \arg \min_{i \in I} \{|i - N/2|\} \)
8     \( A \leftarrow G[1 \ldots N'][1 \ldots N'] \)
9     \( B \leftarrow G[1 \ldots N'][(N' + 1) \ldots N], 0 \) replaced with \( \bot \) and 1 with edge labels
10    \( C \leftarrow G[(N' + 1) \ldots N][(N' + 1) \ldots N] \)
11    \( A' \leftarrow \text{recovery}(\text{closure}(A), \text{left}) \)
12    \( C' \leftarrow \text{recovery}(\text{closure}(C), \text{right}) \)
13    return \( \begin{bmatrix} A' & \text{mul}(A', \text{mul}(B, C')) \\ 0 & C' \end{bmatrix} \)
A simple construction shows that any where $u,v \in N$ solve all-pairs computing reachability is asymptotically equivalent to matrix multiplication [8], there is no hope to by definition encode the necessary information to answer reachability queries in constant time. As algorithm (than by a logarithmic factor), as our construction can produce all dominator trees, which from each vertex $v$ as follows: we add two new vertices $G$ used as a reachability oracle for any graph. Let $6 Matching lower bounds

Assume that $\text{stronglyConnected}(i, i - 1) == \text{false}$ or $\text{stronglyConnected}(i, i + 1) == \text{false}$ and that minimizes $|i - \lfloor N/2 \rfloor|$. 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 bound

$$T(N) = T(N'') + O((N' - N'')^\omega) + T(N - N') + O(N^\omega \log N_0) \leq T(1/3N) + T(1/3N) + O(N^\omega \log N_0),$$

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 $N - N' \leq 1/2N$

$$T(N) = T(N') + T(N - N') + O(N^\omega \log N_0) \leq T(N/2) + T(2/3N) + O(N^\omega \log N_0).$$

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. \qed

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 $G$ be a DAG. We create a graph $\hat{G} = (\hat{V}, \hat{E})$ from $G$ as follows: we add two new vertices $s$ and $t$ together with the edge $(s, t)$, and for each vertex $v \in V(G)$ we add the edges $(v, s)$ and $(t, v)$. Clearly, $\hat{G}$ is strongly connected since we added paths from each vertex $u$ to any other vertex $v$, namely the path $u, s, t, v$. All the new paths between vertices in $V(G)$ contain the edge $(s, t)$. Therefore, a vertex $u$ has two edge-disjoint paths to $v$ in $\hat{G}$, where $u, v \in V(G)$, if and only if $u$ has a path to $v$ in $G$.

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 [8], there is no hope to solve all-pairs 2-reachability in $o(n^\omega)$. 

22
7 Extension to vertex-disjoint paths

Our approach can be modified so that it reports the existence of two vertex-disjoint (rather than edge-disjoint) paths for any pair of vertices. Although we can formulate the algorithms of Sections 3 and 4 so that they use separating vertices (rather than separating edges), 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 $G = (V,E)$, we compute a modified digraph $\hat{G} = (\hat{V}, \hat{E})$ by replacing each vertex $v \in V$ by two vertices $v^+, v^- \in \hat{V}$, together with the edge $(v^-, v^+) \in \hat{E}$, and replacing each edge $(u,v) \in E$ by $(u^+, v^-) \in \hat{E}$. (Thus, $v^+$ has the edges leaving $v$, and $v^-$ has the edges entering $v$.) Then, for any pair of vertices $u, v \in V$, $G$ contains two vertex-disjoint paths from $u$ to $v$ if and only if $\hat{G}$ contains two edge-disjoint paths from $u^+$ to $v^-$. Suppose that we apply our algorithm to $\hat{G}$. Let $u, v$ be any vertices in $G$. If $v$ is reachable from $u$ in $G$ but all paths from $u$ to $v$ in $G$ contain a common vertex, then the algorithm reports a separating edge $e \in \hat{E}$ for the vertices $u^+$ and $v^-$. If $e = (x^-, x^+)$, 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 edge for all paths from $u$ to $v$ in $G$ and so both $x$ and $y$ are separating vertices.

8 An application: computing all dominator trees

Let $s$ be an arbitrary vertex of $G$. Recall the bridge decomposition $D$ of $D$ (Section 4.2), which is the forest obtained from $D$ after deleting the bridges of flow graph $G$ with start vertex $s$. As noted earlier, $T_v$ is the tree in $D$ that contains vertex $v$, and $r_v$ denotes the root of $T_v$.

We define the edge-dominator tree $\tilde{D}$ 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 $\tilde{D}$. We denote by $\tilde{d}(y)$ the parent of a vertex in $\tilde{D}$. (Both $y$ and $\tilde{d}(y)$ are roots in $D$.)

**Theorem 8.1.** All sources vertex- and edge-dominator trees can be computed from $G_{R}^{\geq 2e}$ in $O(n^2)$ total time.

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

Theorem 8.1 enables us to obtain the following results.

**Reachability queries after the deletion of an edge or vertex.** We preprocess each edge-dominator tree $\tilde{D}$ in $O(n)$ so as to answer ancestor-descendant relations in constant time [21].
also compute in $O(n)$ time the number of descendants in $D$ of every root $r$ in $D$. 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 $\bar{d}(y) = r_x$) and $y$ is an ancestor of $r_t$ in $D$.

- Similarly, given a vertex $s$ and an edge $e = (x, y)$, we can report in constant time 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 $D$.

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

**Computing 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 [26] gave a $O(n^2)$ 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 can perform this test in constant time.

- Similarly, we can report all junctions of a given a pair of vertices in $O(n)$ time. Note that two vertices may have $n$ junctions (e.g., in a complete graph).

**Computing critical nodes and critical edges.** Let $G$ be a directed graph. Define the reachability function $f(G)$ as the number of vertex pairs $(u, v)$ such that $u$ reaches $v$ in $G$, i.e., there is a directed path from $u$ to $v$. Here we consider how to compute the most critical node (resp., most critical edge) of $G$, which is the vertex $v$ (resp., edge $e$) that minimizes $f(G \setminus v)$ (resp., $f(G \setminus e)$). This problem was considered by Boldi et al. [3] who provided an empirical study of the effectiveness of various heuristics. A related problem, where we wish to find the vertex $v$ (resp., edge $e$) that minimizes the pairwise strong connectivity of $G \setminus v$ (resp., $G \setminus e$), i.e., $\sum_i \left(\left|C_i\right|\right)$, where $C_i$ are the strongly connected components of $G \setminus v$ (resp., $G \setminus e$), can be solved in linear time [11, 19].

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

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

9 Concluding remarks

In this paper we have shown that the all-pairs 2-reachability problem can be solved in $O(n^\omega \log n)$ time. Our algorithm produces a witness (separating edge or separating vertex) for every pair of vertices for which 2-reachability does not hold. An important corollary of this result is that we can compute all dominator trees of a digraph within the same time bound. Our work raises some new, and perhaps intriguing, questions. First, since our algorithms can be used to answer queries on whether there exists a path from vertex $u$ to vertex $v$ avoiding an edge $e$, can we extend our approach to reporting avoiding paths within the same $O(n^\omega \log n)$ (or any sub-cubic) running time? Another interesting question is whether one can compute all-pairs $k$-reachability (or equivalently, the existence of $(k - 1)$-cuts) in $\tilde{O}(n^\omega)$ time, or even in sub-cubic time, for $k \geq 3$. It does not seem easy to extend our techniques in this case.

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 33rd Symposium on Foundations of Computer Science FOCS’92, pages 417–426. IEEE, 1992.

[2] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. SIAM Journal on Computing, 28(6):2117–32, 1999.

[3] 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.

[4] 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.
[5] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[7] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1-2):37–46, July 2007.

[8] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory*, pages 129–131. IEEE, 1971.

[9] W. Fraczak, L. Georgiadis, A. Miller, and R. E. Tarjan. Finding dominators via disjoint set union. *Journal of Discrete Algorithms*, 23:2–20, 2013.

[10] 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.

[11] L. Georgiadis, G. F. Italiano, and N. Parotsidis. Strong connectivity in directed graphs under failures. In *28th ACM-SIAM Symposium on Discrete Algorithms SODA’17*, pages 1880–1899, 2017.

[12] F. Grandoni and V. 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.

[13] 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.

[14] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *ACM SIGMOD International Conference on Management of Data SIGMO’08*, pages 595–608, New York, NY, USA, 2008. ACM.

[15] 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.

[16] F. Le Gall. Powers of tensors and fast matrix multiplication. In *39th International Symposium on Symbolic and Algebraic Computation ISSAC’14*, pages 296–303, New York, NY, USA, 2014. ACM.

[17] 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.

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

[19] N. Paudel, L. Georgiadis, and G. F. Italiano. Computing critical nodes in directed graphs. In *19th Workshop on Algorithm Engineering and Experiments ALENEX’17*, pages 43–57, 2017.

[20] P. Sankowski. Maximum weight bipartite matching in matrix multiplication time. *Theoretical Computer Science*, 410(44):4480–4488, October 2009.
[21] R. E. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, 1974.

[22] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[23] V. Vassilevska Williams. Faster replacement paths. In 22nd *ACM-SIAM Symposium on Discrete Algorithms SODA’11*, pages 1337–1346, Philadelphia, PA, USA, 2011. Society for Industrial and Applied Mathematics.

[24] V. Vassilevska Williams. Multiplying matrices faster than Coppersmith-Winograd. In 44th *ACM Symposium on Theory of Computing STOC’12*, pages 887–898, New York, NY, USA, 2012. ACM.

[25] O. Weimann and R. Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms*, 9(2):14:1–14:13, March 2013.

[26] R. Yuster. All-pairs disjoint paths from a common ancestor in $\tilde{O}(n^\omega)$ time. *Theoretical Computer Science*, 396(1):145–150, 2008.

[27] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, May 2002.