A Linear Time Algorithm for Two-Vertex Bottlenecks

Sam Buss Vijay Ganesh Albert Oliveras

March 16, 2024

Abstract

We describe a linear time algorithm that determines all "two-vertex bottlenecks" in a directed graph. This gives all pairs of vertices that disconnect two given nodes s and t in a directed graph. There may be quadratically many two-vertex bottlenecks, but a compressed representation allows them to all be determined in linear time. Applications include the determination of Dual Implication Points (DIPs) in the CDCL solver conflict graph, as discussed in Buss, Chung, Ganesh, and Oliveras [preprint, 2024]. The algorithm for finding all DIPs is an algorithm for Menger's Theorem on a directed graph that not only verifies that two given vertices are not 3-connected but also finds all possible separating vertex pairs.

1 Introduction

A recent paper of S. Buss, J. Chung, V. Ganesh, and A. Oliveras [3] investigates the use of "dual implication points" (DIPs) in Conflict-Driven Clause Learning (CDCL) algorithms for Satisfiability (SAT solvers). The most commonly used learning algorithms for CDCL algorithms identify a "first unique implication point" (first UIP) to form a learned clause [7] (see also [2, 4]). Each time the CDCL reaches a conflict, it analyzes the conflict graph, which is a directed graph on the vertices at the top decision level, to determine the UIP; a learned clause is obtained with the aid of the UIP. This kind of clause learning allows CDCL to simulate exactly the the strength of resolution refutations, at least if the CDCL solver makes the proper choices for decision literals [5, 1]. Buss, Chung, Ganesh, and Oliveras [3] introduced a notion of dual implication points (DIPs) as a method for identifying new extension variables, with the goal of searching for and generating *extended resolution* refutations.

The UIP can be characterized in graph theoretic terms as follows. The conflict graph for CDCL clause learning is a directed graph on the literals set true at the top decision level. It has a unique sink, which is the decision literal, and it has a distinguished source vertex that represents the conflict. An *implication point* is any literal which is the decision literal, or which when removed from

the directed graph disconnects the decision literal from the conflict vertex. In contrast, the DIPs defined by [3] consist of *pairs* of literals which, when removed from the conflict graph, separate the decision literal from the conflict vertex.

Buss, Chung, Ganesh, and Oliveras [3] use DIPs to add some (limited form of) extension to CSCL solvers, and obtain some striking improvements for SAT solver performance on selected types of instances of SAT, notable instances based on Tseitin tautologies or XOR reasoning. For these results, see [3]. The present paper will instead focus on presenting the efficient, linear-time algorithm to determine all of the DIPs that was used for the experimental results in [3].

We present our results for general directed graphs, not just CDCL conflict graphs; therefore we use the terminology "two-vertex bottleneck" (or "TVB" for short) instead of "DIP", as the latter is specific to conflict graphs. This terminology reflects the fact that a two-vertex bottleneck is a pair of vertices that separates a source vertex s a the sink vertex t.

The basic idea of the TVB algorithm builds on the vertex version of Menger's theorem [8] for directed graphs. Two vertices s and t in a directed graph are called "3-connected" if there are three vertex-disjoint paths from s to t. Menger's theorem states that either s and t are 3-connected or there is a pair of vertices u and v (not equal to s or t) such that the removal of u and v disconnects s and t by removing all paths from s to t. The novel part of our algorithm is that, for vertices s and t that are not 3-connected, the algorithm determines all of the pairs u and v of vertices separating s and t. These pairs are called *two vertex bottlenecks*, or "TVBs" for short.

There can be quadratically many TVBs since a vertex can be re-used in multiple TVBs. For instance, this happens when the graph consists of two equal length, parallel paths connecting s to t which are vertex-disjoint. The TVB extraction algorithm can find all the TVBs by using a compressed representation of all possible TVB pairs. This allows all possible pairs of literals that form TVBs to be computed and reported in linear time.

2 Theoretical discussion

We suppose that G is a directed graph, without loss of generality with no self-loops or parallel edges. The distinct vertices s and t are given with s a source vertex, and t a sink vertex. Also without loss of generality, for every vertex u there is a (directed) path from s to u and from u to t. If this does not hold, all vertices without this property can be discarded in linear time.

In the case that the graph is acyclic (as happens with CDCL conflict graphs), we can assume without loss of generality that the graph is topologically sorted. In general, any directed acyclic graph can be topologically sorted in linear time either by using Kahn's algorithm that decrements vertex indegrees [6] or by using a post-order depth-first traversal [9].[1]

[1] In applications to CDCL clause learning, it is easy to generate such a directed acyclic graph from the conflict trail. Since the literals are on the trail in the order of assignment, one naturally obtains a topologically sorted graph.

For acyclic graphs, the graph is coded by giving, for each vertex u, the degree of u and the list of vertices v such that there is an edge from u to v. The vertices are the set $[n+1] = \{0, 1, \ldots, n\}$, Since the graph is topologically sorted, we can assume $s = 0$ and $t = n$ and the edge relation respects integer inequality. That is, there can be an edge from u to v only if $u < v$.

For general acyclic graphs that may not be acyclic, the graph should be encoded by giving for each vertex u, the in-degree and out-degree of u and the list of vertices with an edge to u and the list of vertices with an edge from v. If just the lists of outgoing edges are given, then the lists of incoming edges can be obtained in linear time (and vice-versa).

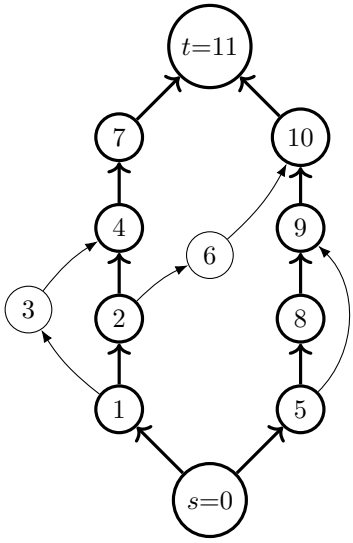
The first stages of the algorithm search for two paths from s to t that are vertex-disjoint. This can done by first arbitrarily choosing a first path, and then using an algorithm similar to augmenting paths to find two paths. If two vertex-disjoint paths do not exist, then there must be at least one vertex t' which is distinct from s and t and which disconnects s and t if it is removed from the graph. The TVB algorithm as described below aborts if it finds such a t' since this does not happen in our intended applications for CDCL solving. However, the algorithm could easily be modified to find the first such t' and replace t with t' before finding all TVBs for s and t'.

Once the two vertex-disjoint paths are found, they are labeled as "path 1" and "path 2". It is a simple observation that any two-vertex bottleneck (TVB) must contain one vertex on path 1 and the other vertex on path 2. Of course, not every pair of vertices chosen in this way will be a TVB since there may be paths that cross between path 1 and path 2. Some of the possible situations are shown in Figure 1.

It is possible that there are $O(n^2)$ manyTVB's. For instance, if the graph has $n+2$ vertices and is formed from s and t and two vertex-disjoint paths both with $n/2$ vertices, then any choice of a vertex u from path 1 and vertex v from path 2 gives a TVB $\{u, v\}$. This makes a total of $n^2/4$ many TVBs. Nonetheless, it is possible to use a compressed representation for TVBs that can be constructed in linear time.

The compressed representation of the TVBs in Figure 1 is shown in Figure 2. The vertices on path 1 that can participate in a TVB are $1, 4, 7$. The vertices on path 2 that can participate in a TVB are $5, 9, 10$. The table of max/min pairs in the figure shows all possible pairings for each participating vertex. For example, the first line of the table shows vertex 1, which lies on path 1: its min/max values are 5 and 10, and this indicates that vertex 1 can be paired with any "participating vertex" from path 2 between 5 and 10 (inclusive). The fifth line shows that vertex 9, which lies on path 2, can be paired only with vertex 1. These pairings given in the compressed representation describe all possible valid TVBs.

It is clear that the compressed representation is linear size. However, we still need to prove that the compressed representation always exists. More generally, we need to characterize which vertices on path 1 and path 2 are participating and to describe how to recognize which pairs of participating vertices can be paired to form TVBs.



All TVBs
$\{1, 5\}$
$\{1, 9\}$
$\{1, 10\}$
$\{4, 10\}$
$\{7, 10\}$

Figure 1: Examples of possible TVBs. Vertices s and t are numbered 0 and 10. Bold circles and bold straight edges show the two paths. The non-bold circles and curved edges are on neither path.

Path	Participating vertices
1	1, 4, 7
2	5, 9, 10

Vertex v	Min pair	Max pair	Therefore v pairs with
1	5	10	5, 9, 10
4	10	10	10
5	1	1	1
7	10	10	10
9	1	1	1
10	1	7	1, 4, 7

Figure 2: Compressed representation for all TVBs from Figure 1.

We assume $G = (V, E)$ is a directed graph on $n + 1$ vertices; without loss of generality $V = [n+1] = \{0, 1, \ldots, n\}$. We use "path" to mean a directed path containing ≥ 1 vertices such that no vertex is visited twice in the path. For convenience, we further assume that, for every $u \in V$, there is a path from $s = 0$ to u and a path from u to $t = n$. Finally, we assume that two paths from s to t have been identified, called "path 1" and "path 2". These two paths have no vertices in common apart from s and t.

For u and v vertices, we write $u \prec v$ to mean that (a) $u \neq v$, (b) u and v are both on path 1 or both on path 2, and (c) u is before v on their common path. "Before" means closer to s. In the special case where G is acyclic and topologically sorted, then $u \prec v$ holds if and only if u and v both lie on path 1 or both lie on path 2 and $u < v$.

Definition 1. A path π from u to v is said to *avoid* path i (for $i \in \{1, 2\}$), if π and path i have no vertices in common except possibly u and v.

Definition 2. Let $i \in \{1, 2\}$ and u be on path i. Then u is *bypassed* if there are vertices u and v on path i with $v \prec u \prec w$ such that there is a path from v to w that avoids both path 1 and path 2.

For example, in Figure 1, vertex 2 lies on path 1 and is bypassed, and vertex 8 lies on path 2 and is bypassed.

Theorem 3. *Suppose vertex u on path i is bypassed. Then u is not part of any TVB.*

Proof. Suppose $\{u, x\}$ form a TVB. Then x must lie on path $i' := 3-i$, since otherwise path i' is a path from s to t that avoids both u and v, contradicting the hypothesis that $\{u, v\}$ are a TVB. Choose a path π from v to w as in Definition 2 witnessing that u is bypassed. Consider the path that follows the first part of path i from s to v, then path π from v to w, and finally the last part of path i from w to t. This path also avoids both u and v, contradicting $\{u, v\}$ being a TVB. Thus u cannot be part of a TVB. □

Theorem 3 establishes that it is necessary that u not be bypassed for u to be part of a TVB. This is not a sufficient condition, however: Figure 3 shows two scenarios in which a vertex u is not bypassed and also is not part of any TVB. In the graph on the left, the presence of the edge from 1 to 6 means that a TVB is not obtained by pairing vertex 3 with either 2 or 4. This is because of the path from s to t with vertices $s, 1, 6, t$. For similar reasons, the edge from 2 to 5 means that vertex 3 cannot be paired with either 4 (again) or 6.

In the graph on the right in Figure 3, the edges from 1 to 4 and from 4 to 5 means that vertex 3 cannot be paired with either 2 or 6. And, since vertex 4 is bypassed, vertices 3 and 4 also do not give a TVB.

Definition 4. Let u and v be distinct from s and t and suppose u lies on path 1 and vertex v lies on path 2. A path from w to w' is a *crossing separator* for u and v provided that w, w' are distinct from s, t and that either

- w lies on path 1, w' lies on path 2, $w \prec u$, and $v \prec w'$; or
- w lies on path 2, w' lies on path 1, $w \prec v$, and $u \prec w'$.

Theorem 5. *Let u and v be distinct from s and t and suppose u lies on path 1 and vertex v lies on path 2. If there is a crossing separator for u and v, then $\{u, v\}$ is not a TVB.*

Proof. This is immediate from the above discussion. Suppose that a path π from w to w' is a crossing separator for u and v with (say) w on path 1. Then following path 1 from s to w, then path π from w to w' and finally path 2 from w' to t gives a path from s to t that avoids both u and v. □

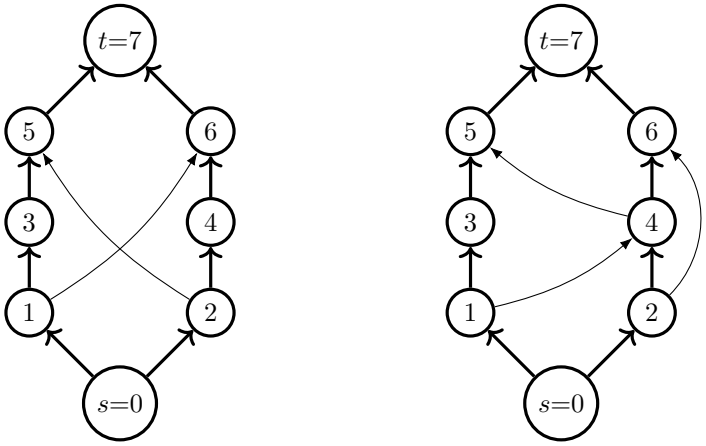


Figure 3: In both graphs, path 1 contains vertices $s, 1, 3, 5, t$ and path 2 contains vertices $s, 2, 4, 6, t$. In both graphs, vertex 3 is not part of any TVB.

Theorem 6. *Let u and v be distinct from s and t and suppose u lies on path 1 and vertex v lies on path 2. Then u and v form a TVB if and only if none of the following hold:*

a. u is bypassed,

b. v is bypassed,

c. u and v have a crossing separator.

Proof. The "only if" direction follows from Theorems 3 and 5. For the converse, suppose that u and v do not form a TVB and thus there is a path π from s to t that avoids both u and v. Consider the first vertex x on π such that $u \prec x$ or $v \prec x$ holds. There is at least one such x since t satisfies this property. Then let y be the last vertex before x on the path π that lies on either path 1 or 2. There must such a y since s satisfies this property. Note that y does not equal u or v since these do not lie on path π. Thus, by choice of x and y, either $y \prec u$ or $y \prec v$. Note that y does not equal u or v since these do not lie on path π. Let π' be the subpath of π from y to x.

If y and x both lie on path 1 or both lie on path 2, then the subpath π' bypasses one of u and v. (This includes the case where $y = s$.) Otherwise, y lies on path 1 and x lies on path 2, or vice-versa. In either case, the subpath π' is a crossing separator for u and v. □

The next observation is that for a fixed vertex u on path i (for $i \in \{1, 2\}$), the restrictions imposed by crossing separators can be coalesced into a minimum value and maximum value for vertices v to pair with u. Specifically, define $csMin(x)$ and $csMax$ (for "crossing separator minimum and maximum"):

Definition 7. Let x be a vertex on path i. Let i' be $3-i$. Then

- $csMin(x)$ is the last vertex w' on path i' such that there is a path π from a vertex w on path i to w' with $w \prec x$.
- $csMax(x)$ is the first vertex w on path i' such that there is a path π from a vertex w to a vertex w' on path i with $x \prec w'$.

For example, in Figure 1, $csMin(4) = csMin(7) = 10$ and $csMax(5) = 2$.

Theorem 8. *Suppose u lies on path i, is distinct from s and t, and is not bypassed. Let i' equal $3-i$ and*

$$Y = \{y : y \text{ is on path } i', y \text{ is not bypassed, and } csMin(u) \leq y \leq csMax(u)\ \}.$$

Then Y also equals $\{y : x$ and y form a TVB$\}$.

Theorem 8 follows immediately from the earlier theorems. It is also the basis for the linear time algorithms presented in the next sections. Section 3 presents the algorithm for directed acyclic graphs in detail, since this is what we needed to implement for our CDCL applications. Section 4 describes how to modify this algorithm to work for graphs that are not necessarily acyclic.

In addition to identifying all TVBs, the algorithm also finds minimum ancestors from the other path. For example, in Figure 1, vertex 1 can be paired with any of the vertices 5, 9, 10 to form a TVB; the minimum ancestor of 1 from the other path is 10 (since there is a path from 1 to 10). Similarly, in the lefthand side of Figure 3, the minimum ancestor of 1 from the other path is 6, and that of 2 is 5. Many applications will not need this minimum ancestor information, and if so the algorithm can be modified to not collect merely by setting or using the `minReach1` and `minReach2` values.

3 The algorithm for directed acyclic graphs

This section presents the algorithm for generating the compressed representation for all possible TVBs in a directed acyclic graph. (A similar algorithm works for non-acyclic directed graphs; this is outlined in Section 4.) As illustrated in Figure 2, the algorithm finds the participating vertices in two paths and, for each participating vertex, finds the corresponding $csMin$ and $csMax$ values. (Note that the two paths are not necessarily unique.) Since we wanted this information for experimenting with finding DIPs in CDCL solvers, the algorithm in addition collects for, each participating vertex u, the identity of the first vertex on the other path which is above (an ancestor of) u.[2]

We present the algorithm in a sequence of five phases, called Phases A, B, C1, C2, and D. A short synopsis of the phases is given first. After that, detailed algorithms are presented. These are presented in pseudocode and follow closely the C++ implementation in the the `TwoVertexBottlenecks` software package available online.[3]

The five phases are:

[2] However, as of the time of writing this, this ancestor information is not used in our CDCL application.

[3] `http://math.ucsd.edu/~sbuss/ResearchWeb/TwoVertBottlenecks`

Phase A: Finds a path from s to t. Each vertex on the path is given a pointer to its successor: relative to the direction of edges in the graph, this is actually a predecessor; that is, in the direction of s.

Phase B: Finds two vertex disjoint paths from s to t, or detects that no such paths exist. If the paths do not exist, the algorithm exits without identifying any TVBs. If two vertex-disjoint paths are found, then vertices labeled as to which path they are on, and they are given pointers to their successors on the path, that is a pointer to the next vertex on the path in the direction of s.

Phase B starts after Phase A has identified a path π from t to t. It searches for two vertex-disjoint paths using a specialized version of an augmenting path algorithm. It alternates between searching forward in the directed graph from s until reaching vertices on π, and searching backward along π towards s. The forward search phases traverse as much of the graph as possible without traversing vertices on π; the reverse search phases only search along π. Each vertex encountered in the forward search phases is given a pointer to the vertex from which it was first reached. Phase B ends once the vertex t is encountered; at that point, the two vertex-disjoint paths are easily identified.

Phase C: This phase gathers information on which vertices on paths 1 and 2 are reachable from each other by paths that avoid both path 1 and 2. Specifically, for each vertex u on path 1 or 2, it determines

- The maximum vertex on path 1 and the maximum vertex on path 2 that are reachable from u by a path that avoids both path 1 and 2. These are called "max-directly-reachable" from u.
- The least vertex on the other path (the one u is not on) that is above u. These are called "min-reachable" from u.

This information is obtained by traversing the graph in reverse topological order, starting at t and ending at s, collecting max-directly-reachable and min-reachable values for *every* vertex u in the graph. If t is found to be max-directly-reachable from s then there are no TVBs, and the algorithm exits.

Phase D: This phase culls the vertices on paths 1 and 2 that are bypassed (as in Definition 2). The non-bypassed vertices are loaded into two arrays P1 and P2. For each (non-bypassed), vertex u on path i, we obtain the maximum "max-directly-reachable" value from all the vertices $u' < u$ that strictly precede u on the path i from s to t. The "min-reachable" information about the least vertex on path $i' = 3-i$ that is reachable from vertex x is preserved.

Phase E: Phase D has collected enough information to determine all the pairs of vertices that can paired to be a TVB. By linearly scanning the two arrays of non-bypassed vertices, it is possible to directly compute, for each x, the minimum value $y = csMin(x)$ and the maximum value $z = csMax(x)$. such that x can be paired with any non-bypassed vertex u from the other path that satisfies $y \leq u \leq z$.

```
1 u := s
2 u.onFirst := true
3 while u ≠ t do
4 |   v := first child of u
5 |   v.succ := u
6 |   v.onFirst := true
7 |   u := v
```

Figure 4: The Phase A algorithm.

Data field	Type	Description
v.succ	int	Successor node to v (towards source s).
v.onFirst	bool	True if on first path.
v.onSecond	bool	True if on second path.
v.handledB	bool	True if already handled in Phase B.
v.direct1	int	Maximum directly reachable vertex on path 1
v.direct2	int	Maximum directly reachable vertex on path 2
v.minReach1	int	Minimum reachable vertex on path 1
v.minReach2	int	Minimum reachable vertex on path 2

Figure 5: The data fields associated with a vertex v. "Maximum" means the closest to t; "Minimum" means the furthest from t. "Directly reachable" means reachable by a path that avoids both path 1 and path 2.

We now give the algorithms in detail, presented as pseudocode. Our pseudocode follows the `C++` implementation `TwoVertBottlenecks` fairly closely, albeit not exactly.

The Phase A algorithm is shown in Figure 4. It works very simply by starting at the source s and following the first outgoing edge of each vertex until reaching the sink node t. This works because we assumed that t is reachable from all vertices. Each vertex on the path is labeled as being on Path 1 and has a pointer named '`succ`' pointing to its predecessor on the path towards s.

A complete list of the data fields for each vertex is shown in Figure 5. The four Boolean values for each vertex are initialized to false before the algorithm starts.

The Phase B algorithm is shown in Figure 6. The while loop starting on line 5 starts with a set (the "DFS stack") of vertices to initiate a depth-first search. The first time this while loop is entered the DFS stack contains just s. When the depth-first search reaches a vertex that does not have its `onFirst` flag set, it sets the .`handledB` flag, sets its successor vertex, and pushes it onto the DFS stack. When it reaches a vertex that does have the `onFirst` flag set, it records the highest such vertex as w and the vertex it was reached from as w', but it does not add any vertex from the first path to the DFS stack. If the maximum reached vertex w on the first path is equal to t, then two disjoint paths have been discovered (line 13). When this happens, all nodes on both of

```
1  Push s onto the DFS stack                    // Starting vertex for a DFS
2  s.handledB := true
3  w := s                                       // Maximum reached vertex on first path
4  while true do
5      while DFS stack is non-empty do
6          Pop v' from the DFS stack
7          foreach each child v of v' do
8              if v.onFirst then
9                  if w ≺ v then w := v and w' := v'
10             else if not v.handledB then
11                 Set v.handledB:=true and v.succ := v'
12                 Push v onto the DFS stack
13     if w = t then break // Found two disjoint paths
14     x := w'
15     while (not x.onFirst) do x.onFirst := true; x = x.succ
16     if w.handledB then abort // Abort if no disjoint paths
17     y := w
18     while (not y.handledB) do
19         y.handledB := true; y.onFirst := false
20         Push y onto the DFS stack
21         y := y.succ
22 z := w'
23 while z ≠ s do
24     z.onFirst = false; z.onSecond = true
25     z := z.succ
26 s.onSecond := true; t.onSecond := true;
```

Figure 6: The Phase B algorithm.

the paths are marked as being on the first path (with `onFirst`). To fix this, lines 22-25 follow the `succ` pointers to follow the new (second) path and to reset their `onFirst` flags and set their `onSecond` flags. The value of w' gives the vertex on the second path that is the (alternate) successor of t.

On the other hand, if the highest reached vertex w on the first path is not equal to t, then it backtracks from w' following `succ` pointers to label vertices as being on the first path (even though these vertices may end up on the second path). After that lines 14-21 backtrack from w along the first path until reaching a vertex with the `reachedB` flag set. Each vertex reached in this way is marked by setting `reachedB` true and added to the DFS stack for the next depth-first search.

To understand how Phase B carries out an augmenting flow algorithm, note the first while loop is following edges that contain no flow, and lines 14-21 are

```
1  Set t.direct1, t.direct2, t.minReach1, t.minReach2 all to t.
2  for  x := t−1, t−2, ..., 2, 1, s do
3      min1 := t; min2 := t; max1 := s;max2 := s
4      foreach child y of x do
5          if not (y.onFirst and y.onSecond) then
6              max1 := max(y.direct1, max1)
7              max2 := max(y.direct2,max2)
8          else
9              if y.onFirst then  max1 := max(y, max1)
10             if y.onSecond then  max2 := max(y, max2)
11         if  y.onFirst then min1 := y
12         else  min1 := min(y.minReach1, min1)
13         if  y.onSecond then min2 := y
14         else  min2 := min(y.minReach2, min2)
15     x.direct1 := max1; x.direct2 := max2
16     x.minReach1 := min1; x.minReach2 := min2;
```

Figure 7: The Phase C algorithm

following reverse edges that do carry a flow by virtue of being on the first path.

Phase C sets the following values for every vertex x:

- For each $i = 1, 2$, the maximum vertex y on path i such that there is a (directed) path from x that avoids both path 1 and 2.
- For each $i = 1, 2$, the minimum vertex y on path i for which there is a (directed) path from x towards t that reaches y. (These values do not enter into the computation of the TVBs, and thus are needed only if the information about these minimum reachable vertices is also wanted.)

This information is collected by iterating the values of x from $t = n-1$ down to $s = 0$. This is completely straightforward as these values for x can computed from these values for the child vertices y of x, and the knowledge of whether y is on path 1 or 2. The pseudocode is shown in Figure 7.

If the minimum-reachable information is not needed, then Phase C can be simplified by removing all computations involving min1 and min2, including lines 11-14 and 16.

Phase D creates arrays P1 and P2 containing the members of paths 1 and 2 that are not bypassed. For $i = 1, 2$, the ℓ-th member of P$i[\ell]$ has two entries:

P$i[\ell]$.`vert`	A vertex x from path i
P$i[\ell]$.`reachOther`	Maximum vertex on the other path directly reachable from a vertex $x' < x$ on path i

One small inconvenience is that the paths are specified with pointers (`succ` values) that point along the path from t to s, but Phase D needs to traverse

```
1  x := t.succ                          // last vertex on path 1 before t
2  while x ≠ s do
3  |  P1[ℓ].vert := x;
4  |  x := x.succ; ℓ := ℓ + 1
5  Reverse the order of members of the array P1.
6  to := 0
7  curBypass := s.direct1
8  curReachOther := s.direct2
9  for x = P1[0].vert, ..., P1[ℓ−1].vert do
10 |  if x ≥ curBypass then
11 |  |  P1[to].vert := x
12 |  |  P1[to].reachOther := curReachOther
13 |  |  to := to + 1
14 |  curBypass := Max(curBypass, x.direct1)
15 |  curReachOther := Max(curReachOther, x.direct2)
```

Figure 8: The Phase D algorithm. This algorithm is carried out twice: once as written for path 1, and once for path 2, for which x is initialized as w' instead of t.succ.

the paths from s to t. Accordingly, the paths are first traversed from t to s to store the path vertices in the arrays P1 and P2. It then traverses the vertices in the array in the direction from s to t. It is easy to recognize which vertices are bypassed by keeping a running maximum value (curBypass) of directly reachable vertices. Bypassed vertices are removed from the arrays.

The code for Phase D's processing of path 1 vertices is shown in Figure 8. The path 2 vertices are processed in exactly the same way except with the roles of the paths 1 and 2 interchanged, and with the vertex x on the path that is adjacent to t initialized to equal w' instead of t.succ. The result is that the arrays P1 and P2 contain the non-bypassed nodes, along with information that will let us identify the TVBs. Note that some vertices in P1 and P2 still might not participate in a TVB (see Figure 3 and the associated discussion).

Upon finishing Phase C, if either list P1 or P2 is empty, then there are no TVBs. Otherwise, Phase E finalizes the information about all the TVBs, if any. The calculation is that for the ℓ-th vertex in P1, the vertex $x :=$ P1[ℓ].vert cannot be paired with any vertex y such that $y <$ P1[ℓ].reachOther. For the same (dual) reason, x cannot be paired with y if $y :=$ P2[d].vert and $x <$ P2[k].reachOther. Since the reachOther values are increasing, the bounds for all x on what y's be paired with x to form a TVB can be calculated easily with linear-time traversals of P1 and P2.

The Phase E algorithm is shown in Figure 9. The algorithm is run twice: once as shown, and once with the roles of paths 1 and 2 reversed. In fact, it is redundant to run it twice, because the full information about all the TBV pairs is obtained by either running. Namely, running it for path 1, makes it

```
   // P1 and P2 have ℓ0 and k0 entries, respectively.
1  k := 0
2  for ℓ = 0, 1, 2, ..., ℓ0−1 do
3  |  x := P1[ℓ].vert
4  |  while k < k0 and P2[k].reachOther ≤ x do k := k + 1
5  |  if k = 0 then continue
6  |  z := P2[k−1].vert                 // Max possible TVB pair for x
7  |  y := P1[ℓ].reachOther          // Lower bound on TVB pair for x
8  |  if y ≤ z and (k < k0 or z ≤ P2[k0−1].vert) then
9  |  |  Report: x has at least one TVB pair, and any u in P2
   |  |  |       with y ≤ u ≤ z forms a valid TVB pair with x.
   |  |  |       x.minReach2 is the least ancestor of x on path 2.
10 if no vertex x reported then abort  // Aborts if no TVB pairs
```

Figure 9: The Phase E algorithm. This algorithm may be carried out twice: once as written for path 1, and again with the roles of the paths interchanged for path 2.

convenient to find all TVB pairs involving a vertex x on path 1; and conversely running it for path 2 makes it convenient to find all TVB pairs involving a vertex x on path 2. The algorithm works by considering every vertex x in P1. Since the x values are increasing, and since the `reachOther` values (in P2) are also increasing, we can find the minimum entry P2[k] with `reachOther` value $> x$ by continuing to traverse the P2 from where the previous time through the loop left off. This gives immediately values y and z bracketing the possible vertices u that can be paired with x. In most cases, there will be at least one such u, namely, z itself. However, line 8 includes an extra test that applies when the members of P2 have been exhausted and y does not actually exist: this check ensures that at least the last vertex in P2 can be paired with x.

4 The algorithm for general directed graphs

We now describe the modifications to the algorithm that are needed to work with general directed graphs that are necessarily acyclic. The same five phases are still used. The primary changes are to Phases A and C. We assume $G = (V, E)$ is a directed graph with vertex set $V = [n+1]$ and the distinguished vertices are $s = 0$ and $t = n$. Without loss of generality, s is a source, and t is a sink. Each vertex u in G has a list of the vertices v such that $\langle u, v\rangle \in E$ and a list of the vertices w such that $\langle w, u\rangle \in E$.

Phase A still must identify a path from s to t. This is done using a depth-first search following directed edges in G. The vertices on path 1 are marked by setting their `onFirst` values. As before, each vertex has a pointer (called `succ`) to the vertex before it on the path (namely, the vertex closer to s). In addition, each vertex x on path 1 is assigned its index on the path, so that x.`indexOnPath`

is equal to i for x the ith vertex on the path. The point of the index values is to allow testing the condition $x \prec y$ for vertices x and y on path 1

Phase B still must find two vertex-disjoint paths from s to t. As always, we assume that no vertex appears twice on a path. This can be done with a standard augmenting path algorithm; indeed, the Phase B algorithm shown earlier in Figure 6 works without modification.

The main change to the algorithm is in Phase C. For a linear time algorithm, we use the fact that each vertex has a list of the incoming edges. Phase C must compute, for each vertex x, the values of x.`direct1` and x.`direct2` giving the maximal directly reachable vertices from paths 1 and 2. To compute the `direct1` values, the algorithm traverses path 1 starting at vertex t. For each vertex on path 1, it follows incoming edges to perform a reverse DFS seeking directly reachable vertices. The reverse DFS stops whenever it reaches a node that has been previously traversed or that lies on path 1 or 2. When a vertex x is reached for the first time, its x.`direct1` value is set and it is marked as handled. Once a vertex is marked as handled, it will not be re-traversed in subsequent reverse DFS's from nodes on path 1.

The .`direct2` values are set similarly, by carrying out reverse DFS's from the nodes on path 2, starting with a reverse DFS from t.

If the information about minimum ancestors, `minReach1` and `minReach2`, is needed it is obtained by a similar algorithm that carries out reverse DFS's from vertices along the paths 1 and 2. These searches, however, start with s instead of t.

Phases D and E use the .`direct1` and .`direct2` information to find all possible TVBs. The algorithms outlined earlier still work with no essential changes needed.

References

[1] A. Atserias, J. K. Fichte, and M. Thurley, *Clause-learning algorithms with many restarts and bounded-width resolution*, Journal of Artificial Intelligence Research, 40 (2011), pp. 353–373.

[2] P. Beame, H. A. Kautz, and A. Sabharwal, *Towards understanding and harnessing the potential of clause learning*, Journal of Artificial Intelligence Research, 22 (2004), pp. 319–351.

[3] S. Buss, J. Chung, V. Ganesh, and A. Oliveras, *Title to be determined*. In preparation, 2024.

[4] S. Buss and J. Nordström, *Proof complexity and SAT solving*, in Handbook of Satisfiability - Second Edition, Frontiers in Artificial Intelligence and Applications 336, IOS Press, 2021, pp. 233–350.

[5] A. Darwiche and K. Pipatsrisawat, *Complete algorithms*, in Handbook of Satisfiability, IOS Press, 2009, pp. 99–130.

[6] A. B. Kahn, *Topological sorting of large networks*, Communications of the ACM, 5 (1962), pp. 558–562.

[7] J. P. Marques-Silva and K. A. Sakallah, *GRASP — A new search algorithm for satisfiability*, IEEE Transactions on Computers, 48 (1999), pp. 506–521.

[8] K. Menger, *Zur algemeinen Kurventheorie*, Fundamenta Mathematicae, 10 (1927), pp. 96–115.

[9] R. E. Tarjan, *Edge-disjoint spanning trees and depth-first search*, Acta Informatica, 6 (1976), pp. 171–185.