


# An Algorithm for the Exact Treedepth Problem

James Trimble 

School of Computing Science, University of Glasgow, UK  
j.trimble.1@research.gla.ac.uk

---

## Abstract

We present a novel algorithm for the minimum-depth elimination tree problem, which is equivalent to the optimal treedepth decomposition problem. Our algorithm makes use of two cheaply-computed lower bound functions to prune the search tree, along with symmetry-breaking and domination rules. We present an empirical study showing that the algorithm outperforms the current state-of-the-art solver (which is based on a SAT encoding) by orders of magnitude on a range of graph classes.

**2012 ACM Subject Classification** Theory of computation → Graph algorithms analysis; Theory of computation → Algorithm design techniques

**Keywords and phrases** Treedepth, Elimination Tree, Graph Algorithms

**Digital Object Identifier** 10.4230/LIPIcs.SEA.2020.19

**Related Version** A full version of the paper is available at <https://arxiv.org/abs/2004.08959>.

**Supplementary Material** Source code: <https://github.com/jamestrimble/treedepth-solver>

**Funding** *James Trimble*: This work was supported by the Engineering and Physical Sciences Research Council (grant number EP/R513222/1).

**Acknowledgements** Thanks to Ciaran McCreesh, David Manlove, Patrick Prosser and the anonymous referees for their helpful feedback, and to Robert Ganian, Neha Lodha, and Vaidyanathan Peruvemba Ramaswamy for providing software for the SAT encoding.

## 1 Introduction

This paper presents a practical algorithm for finding an optimal treedepth decomposition of a graph. A *treedepth decomposition* of graph  $G = (V, E)$  is a rooted forest  $F$  with node set  $V$ , such that for each edge  $\{u, v\} \in E$ , we have either that  $u$  is an ancestor of  $v$  or  $v$  is an ancestor of  $u$  in  $F$ . The *treedepth* of  $G$  is the minimum depth of a treedepth decomposition of  $G$ , where depth is defined as the maximum number of vertices along a path from the root of the tree to a leaf.

Treedepth is closely related to a number of other problems. The treedepth of a connected graph  $G$  equals the minimum height of an elimination tree for  $G$  ([12], chapter 6), which equals the graph's vertex ranking number [3]. The treedepth of a graph  $G$  is also equal to the minimum number of colours in a centred colouring of  $G$  [12].

Finding an elimination tree of small height is applicable to the parallel Cholesky factorisation of sparse matrices [17]. Treedepth also has relevance to the design of fixed-parameter tractable (FPT) algorithms. For example, the Mixed Chinese Postman Problem is FPT when parameterised by treedepth, but W[1]-hard when parameterised by treewidth or pathwidth [9].

The decision variant of the treedepth problem is NP-complete [13]. However, it can be solved in linear time if the input graph is a tree [16], and in polynomial time for interval graphs [1], trapezoid graphs, permutation graphs and circular arc graphs [4]. There is a polynomial-time approximation algorithm for the problem that gives a result within  $O(\log^2 n)$  of the optimal value. The problem is fixed parameter tractable with respect to both treewidth and treedepth [2, 14].



© James Trimble;

licensed under Creative Commons License CC-BY

18th International Symposium on Experimental Algorithms (SEA 2020).

Editors: Simone Faro and Domenico Cantone; Article No. 19; pp. 19:1–19:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Although numerous heuristics for finding good elimination trees have been designed and implemented [8], we are aware of only two existing implementations of *exact* algorithms for minimum treedepth decomposition; both of these are introduced in [6, 7]. In each case, the optimisation problem is solved as a sequence of decision problems, with each decision problem encoded as an instance of the boolean satisfiability problem and solved using a general-purpose SAT solver.

**This paper’s contribution.** This paper introduces a new algorithm for computing an optimal treedepth decomposition. The algorithm is self-contained and does not require an external solver, although it can optionally use the graph-automorphism library Nauty to break symmetries. The basic structure of the algorithm is very simple. To improve performance, three symmetry breaking and domination features and two lower-bounding functions are added to the algorithm. In a set of experiments, we show that our algorithm is typically orders of magnitude faster than the current (SAT-based) state of the art.

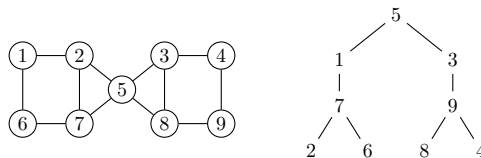
**Structure of the paper.** Section 2 introduces concepts and notation. Section 3 presents the core parts of our algorithm. Section 4 describes enhancements to the basic algorithm. Section 5 provides details of our implementation. Section 6 presents an experimental comparison with the existing SAT encoding for treedepth. Section 7 concludes.

## 2 Preliminaries

Let  $G = (V, E)$  be a graph, where we assume that the elements of  $V$  are integers. We write  $V(G)$  and  $E(G)$  to denote the vertex and edge sets of  $G$ . The neighbourhood  $N_G(v)$  of a vertex  $v$  is the set of vertices that are adjacent to  $v$ . For  $S \subseteq V$ , we denote by  $G[S]$  the subgraph of  $G$  induced by  $S$ ; that is,  $(S, \{\{u, v\} \in E \mid u, v \in S\})$ . We use the notation  $G - v$  for the removal of one vertex and its incident edges; that is,  $G - v = G[V(G) \setminus \{v\}]$ .

The concepts *elimination forest* and *elimination tree* are defined recursively in terms of one another. An elimination forest of graph  $G$  is a rooted forest with vertex set  $V(G)$  composed of elimination trees of each of  $G$ ’s connected components. An elimination tree of a non-empty connected graph  $C$  is a rooted tree  $T$  with vertex set  $V(C)$ . If  $C$  has only one vertex  $v$ , then  $T$  is a tree containing only  $v$ . Otherwise,  $T$  is formed by choosing a vertex  $v \in V(C)$  as the root, finding an elimination forest of  $C - v$ , and making the trees in that forest the child subtrees of  $v$ .

We will use the graph  $G$  in Figure 1 to provide an example of an elimination tree, and as our running example throughout the paper. The second part of the figure shows an optimal elimination tree of  $G$ , which has depth 4. Observe that removing the root vertex of the tree (5) from  $G$  splits the graph into two connected components, and each of these components corresponds to one of the child subtrees of 5 in the elimination tree.



■ **Figure 1** An example graph  $G$  (left) and an optimal elimination tree of  $G$  (right).

Every elimination forest is a treedepth decomposition, and for a given graph  $G$  there is at least one elimination forest whose depth equals the treedepth of  $G$  [12, 11]. Therefore, in order to find an optimal treedepth decomposition of  $G$  it is sufficient to search for a minimum-depth elimination tree of  $G$ . This is the approach taken in this paper.

### 3 The Algorithm

Our algorithm is shown as pseudocode in Algorithm 1. The shaded parts, and the third parameter of each of the first two functions, will be introduced in later sections and can be disregarded for now.

**Algorithm 1** An algorithm to find an elimination forest of minimum depth. To read the basic algorithm (without optimisations), disregard the shaded sections and the third parameter of each of the first two functions.

---

```

1  elimination_forest( $G, k, w$ )
2  Data: Graph  $G$ , maximum depth  $k$ , and parent vertex  $w$ 
3  Result: true if and only if an elimination forest of  $G$  with depth  $\leq k$  exists
4  begin
5      if  $k = 0$  and  $|V(G)| > 0$  then return false
6       $\mathcal{C} \leftarrow$  the connected components of  $G$ 
7      for  $C \in \mathcal{C}$  do
8          if simple_lower_bound( $|V(C)|$ )  $> k$  then return false
9      for  $C \in \mathcal{C}$  do
10         if can_prune_by_path_lower_bound( $C, k$ ) then return false
11     for  $C \in \mathcal{C}$  do
12         if not elimination_tree( $C, k, w$ ) then return false
13     return true

14 elimination_tree( $G, k, w$ )
15 Data: Connected, nonempty graph  $G$ , maximum depth  $k \geq 1$ , and parent vertex  $w$ 
16 Result: true if and only if an elimination tree of  $G$  with depth  $\leq k$  exists
17 begin
18     if  $|V(G)| = 1$  then
19          $v \leftarrow$  the unique element of  $V(G)$ 
20         parent[ $v$ ]  $\leftarrow w$ 
21         return true
22     for  $v \in V(G)$  do
23         if  $v$  is ruled out by a symmetry or domination rule then continue
24         parent[ $v$ ]  $\leftarrow w$ 
25         if elimination_forest( $G - v, k - 1, v$ ) then return true
26     return false

27 optimise( $G$ )
28 Data: A graph  $G$ 
29 Result: The treedepth of  $G$ 
30 begin
31      $k \leftarrow 0$ 
32     while elimination_forest( $G, k, 0$ ) = false do  $k \leftarrow k + 1$ 
33     return  $k$ 

```

---

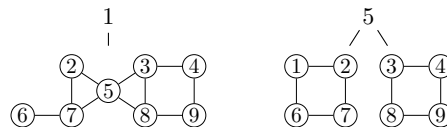
**Inputs and outputs.** The algorithm's first function, `elimination_forest()`, takes a graph  $G$  and an integer  $k \geq 0$ , and returns *true* if and only if there exists an elimination forest of  $G$  of depth  $k$  or less. The function `elimination_tree()` takes a connected, non-empty graph  $G$  and an integer  $k > 0$  and returns *true* if and only if there exists an elimination tree of depth  $k$  or less. The algorithm is run by calling `optimise()`, which takes a graph  $G$  and returns the treedepth of  $G$ .

**Details of the functions.** The first two functions are mutually recursive, and closely follow the definitions of elimination tree and forest. The function `elimination_forest()` begins by returning *false* – indicating infeasibility – if an elimination tree of depth zero is sought for a non-empty graph. The function then returns *true* if and only if an elimination tree of depth no greater than  $k$  exists for each connected component of the graph.

The function `elimination_tree( $G, k$ )` returns *true* if  $G$  has a single vertex (lines 18 to 21). Otherwise, it tries each vertex  $v \in V(G)$  in turn (line 22), and returns *true* if and only if one of these  $v$  values can be the root of an elimination tree of depth  $k$  – which is the case if and only if an elimination forest of depth no greater than  $k - 1$  exists for  $G - v$ .

The main function, `optimise()`, carries out repeated calls to `elimination_forest()` with ascending values of  $k$  until a feasible depth is reached. It would be possible to implement a branch-and-bound variant of the algorithm with a little additional effort, and it is likely that this would be somewhat faster than the approach we have taken. We decided not to do so for two reasons. First, having a sequence of decision problems simplifies the exposition of the algorithm. Second, we have observed in practice that two of the decision problems – the final unsatisfiable problem and the satisfiable problem after which the algorithm terminates – take up most of the run time. This suggests that a branch-and-bound approach would be of limited benefit.

**Example.** Returning to our example graph  $G$  from Figure 1, suppose  $G$  is passed to `elimination_tree()`. Figure 2 illustrates two of the nine subproblems explored at line 22. In the left part of the figure,  $v = 1$ . Choosing this vertex as the root of the elimination tree leaves a connected graph, which is passed to `elimination_forest()` at line 25. The right part of the figure corresponds to the decision  $v = 5$ . Choosing this vertex as the root leaves a disconnected graph, with two four-vertex components. This disconnected graph is passed to `elimination_forest()` at line 25, and subsequently `elimination_tree()` is called for each of the two components.



■ **Figure 2** Two of the nine subproblems visited by the first call to `elimination_tree` on our example graph.

### 3.1 Generating an Elimination Forest

The algorithm described so far returns only a single integer: the treedepth of the input graph. We can easily modify the algorithm to also produce an elimination forest of that depth.

We assume that vertices of the input graph  $G$  are numbered from 1 to  $|V(G)|$ . A global array with  $|V(G)|$  elements, *parent*, is used to record the parent of each vertex in the elimination forest. A value *parent*[ $v$ ] = 0 indicates that vertex  $v$  is a root. Lines 20 and 24 of Algorithm 1 record the parent of  $v$ .

The functions `elimination_forest()` and `elimination_tree()` both have  $w$  as an extra parameter. Vertex  $w$  will be parent to the first vertex chosen from  $G$ . At the first level of recursion for either of these functions,  $w = 0$ , indicating that the next vertex to be selected will be a root.

When either `elimination_forest()` and `elimination_tree()` returns *true*, this indicates not only that an elimination tree of depth  $k$  of the subgraph  $G$  exists, but also that such a decomposition has been recorded in the *parent* array (with any *parent* value not in  $V(G)$  indicating a root of the subproblem's decomposition).

## 4 Enhancements to the Algorithm: Symmetry Breaking and Domination Rules, Pruning, and Sorting

In this section we describe five improvements that can be made to the basic algorithm. The first three of these are symmetry breaking and domination rules that allow us to avoid choosing some values of  $v$  at line 22 of `elimination_tree()`. The fourth technique allows us to prune subproblems by quickly computing a lower bound on treedepth; we introduce two such bounds. The final technique is a re-ordering of vertices before solving.

### 4.1 Symmetry Breaking and Domination Rules

Recall that the loop at line 22 of `elimination_tree()` tries each vertex  $v$  as a potential root of the elimination tree, attempting to find an elimination tree of depth  $k$  or less. For some graphs, there are several values of  $v$  that may be chosen as the root of such a tree; let  $S$  be the set of such vertices. We can omit some choices of  $v$  at line 22 without affecting the algorithm's correctness, provided at least one member of  $S$  is chosen. The three symmetry-breaking and domination rules that follow make use of this fact to avoid visiting some vertices. They ensure that the least-numbered vertex in  $S$  is visited, if  $S$  is non-empty.

**Symmetry breaking using vertex orbits.** Our first symmetry breaking technique is applied only to connected graphs. Before beginning the algorithm, we use Nauty [10] to compute the orbits of the vertices. (Recall that vertices  $v$  and  $v'$  are in the same orbit if and only if there is an automorphism that maps  $v$  to  $v'$ .) In the first call to `elimination_tree()` – that is, the call where  $G$  is the full input graph – we can avoid choosing any value of  $v$  in the loop if there exists a vertex  $v' < v$  that is in the same orbit as  $v$ . This symmetry-breaking technique is valid because the subgraph created by the removal of  $v$  is isomorphic to the subgraph created by the removal of  $v'$ , and thus has the same treedepth.

As an example, consider the graph in Figure 1. Since vertices 1, 4, 6, and 9 are in the same orbit, we can avoid choosing vertices 4, 6, and 9 in the first call to `elimination_tree()`.

This technique is useful on highly symmetrical graphs, but of course cannot be expected to achieve a speedup of more than  $|V(G)|$ . A number of avenues for improved symmetry breaking could be explored in future work. It would be straightforward to extend the symmetry breaking to disconnected graphs by computing the orbits of vertices in each connected component separately. Our technique for symmetry breaking could also be used for subproblems; this may be useful for very symmetrical graphs, but we suspect that the cost of additional calls to Nauty would outweigh the benefit in many cases. We could move beyond finding the orbits of single vertices; it is possible to find the orbits of pairs (or larger sets) of vertices with a single call to Nauty and a small amount of extra work. Lastly, we could use symmetry-breaking techniques from constraint programming such as GE-trees [15].

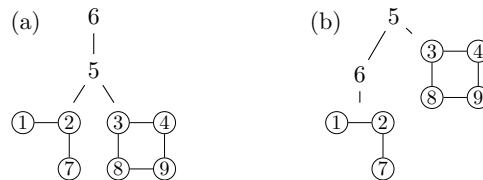
**Vertex domination.** Suppose we have  $v, v' \in V(G)$  (not necessarily adjacent) such that  $v' < v$  and  $N_G(v') \setminus \{v\} \supseteq N_G(v) \setminus \{v'\}$ . We say that  $v'$  *dominates*  $v$ . Clearly,  $G - v'$  is isomorphic to a subgraph of  $G - v$ . Thus, the minimum depth of a treedepth decomposition of

$G$  rooted at  $v$  is no smaller than the minimum depth of a decomposition rooted at  $v'$ . We can use this fact in `elimination_tree()` (at any depth of recursion) by rejecting at line 22 any value of  $v$  such that there exists a lower-numbered  $v'$  such that  $N_G(v') \setminus \{v\} \supseteq N_G(v) \setminus \{v'\}$ .

As an example, suppose the input graph is a clique  $K_n$ , with the vertices numbered  $\{1, \dots, n\}$ . At each call to `elimination_tree()`, the lowest-numbered vertex in  $G$  dominates the other vertices; thus only one vertex needs to be explored in the loop at line 22.

Our use of vertex domination is based on [6, 7], where the technique is used in a preprocessing step to generate constraints for the input graph, rather than applied to each subproblem.

**Only-child vertices.** Consider again our example graph in Figure 1. Suppose that the symmetry-breaking and domination rules described so far in this section are disabled, and that a decomposition of depth 3 is being sought. Figure 3(a) shows the program state after selecting vertex 6 as the root vertex of the tree and vertex 5 as its child. There are two subproblems: the subgraphs induced by  $\{1, 2, 7\}$  and  $\{3, 4, 8, 9\}$ . We call 5 an *only child* in the tree because it has a parent (vertex 6) but has no siblings.



■ **Figure 3** The only child rule allows us to avoid the effort of exploring the subproblem in the left part of the figure, since at least as good a decomposition can be achieved by choosing 5 as the root vertex.

Figure 3(b) shows the tree if vertex 5 is chosen before, rather than after, vertex 6. Observe that the same two subproblems appear, but one of them has been lifted to a higher level in the tree. It is clear that the optimal elimination tree based on Figure 3(a) will have depth no less than that of the optimal elimination tree based on Figure 3(b).

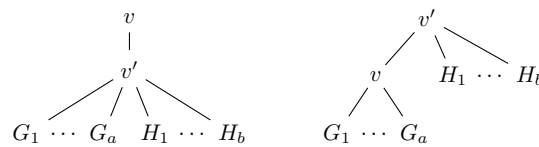
This is an example of a rule that holds in general.

► **Proposition 1.** (*Only-child rule*) *Let  $T$  be a depth- $k$  treedepth decomposition of a connected graph  $G$ . If the root vertex of  $T$  has an only child  $v'$ , then there exists a treedepth decomposition of  $G$  of depth no greater than  $k$  with  $v'$  as its root.*

**Proof.** Let  $v$  be the root of  $T$ , and  $v'$  its only child. Let  $\{G_1, \dots, G_a\}$  be the child subproblems that contain a vertex adjacent to  $v$  in  $G$ , and let  $\{H_1, \dots, H_b\}$  be the child subproblems that do not, as illustrated in Figure 4 (either of these sets of subproblems may be empty). For each graph in  $\{G_1, \dots, G_a\} \cup \{H_1, \dots, H_b\}$ , there must exist a decomposition of depth at most  $k - 2$ .

If we reverse the order of  $v$  and  $v'$ , then the tree and its subproblems will be as shown in the second part of Figure 4 (where  $\{H_1, \dots, H_b\}$  are moved up a level). Clearly, it is possible to construct a decomposition with depth no greater than  $k$  by using the same decompositions of the subproblems  $\{G_1, \dots, G_a\} \cup \{H_1, \dots, H_b\}$  as in  $T$ . ◀

This can be used for a simple domination breaking rule. If the removal of a vertex  $v$  chosen at line 22 of Algorithm 1 leaves a non-empty, connected graph, then the child vertex of  $v$  in the treedepth decomposition,  $v'$ , must be an only child. The only-child rule allows us to omit any vertex  $v'$  that has a lower number than  $v$ .



■ **Figure 4** The general case of the only child rule. If a vertex  $v$  has an only child  $v'$  in the elimination tree, then an elimination tree of the no greater depth can be found by reversing the positions of  $v$  and  $v'$ .

## 4.2 Computing Lower Bounds

At lines 7 to 10 of Algorithm 1, lower bounds are computed with the goal of quickly proving that one of the subproblems is infeasible. For each connected component  $C$ , two functions are called to find lower bounds on the treedepth of  $C$ . If either of these bounds is greater than  $k$ , the current subproblem is unsatisfiable and *false* is returned. The first lower-bounding function uses a simple and very fast algorithm to compute a bound based on the number of vertices in the subproblem and an upper bound on the maximum degree. The second function greedily constructs a path in the graph, and uses as a lower bound a well-known formula for the treedepth of a path graph.

**Simple lower bound.** Let  $b > 0$  be an upper bound on the maximum degree of a graph  $G$ . If  $G$  has no vertices, then its treedepth is zero. Otherwise, if we remove a single vertex and its incident edges from  $G$ , it is clear that the resulting graph can have at most  $b$  connected components and at least one of these components must have  $\lceil (|V(G)| - 1)/b \rceil$  or more vertices. Algorithm 2 is a recursive algorithm that makes use of this fact to give a lower bound on the treedepth of a graph.

■ **Algorithm 2** The simple lower bound function.

---

```

1 simple_lower_bound( $n$ )
2 begin
3   if  $n = 0$  then return 0
4   return  $1 + \text{simple\_lower\_bound}(\lceil (n - 1)/b \rceil)$ 

```

---

In our implementation,  $b$  is a global variable equal to the maximum degree of the input graph. If  $b = 0$ , we do not use this lower bounding technique.

This bounding algorithm runs in time  $O(\log n)$ , where  $n$  is the argument passed to the function. As an optimisation, our implementation pre-computes `simple_lower_bound( $n$ )` for  $n \in \{0, \dots, |V(G)|\}$ , and saves these values in an array before calling `td_optimise()`, thus allowing the bounding function to run in constant time. However, we use a bitset popcount (number of set bits) operation to calculate the value of  $n$ , and therefore the overall time complexity of calculating this bound is  $O(n)$ , where  $n$  is the size of the input graph.

Our example graph in Figure 1 has 9 vertices and maximum degree 4. The bounding function therefore gives a lower bound of 3 on the graph's treedepth.

**Path lower bound.** A graph containing a path of  $k$  vertices has treedepth at least  $\lceil \log_2(k + 1) \rceil$  [12]. We can thus cheaply compute a lower bound on the treedepth of a graph  $G$  by greedily finding a path in  $G$ , as shown in Algorithm 3. We choose the lowest-numbered vertex  $v$  in  $G$  as our starting point, and attempt to grow the path from  $v$  in two directions (line 5). In each of these two growing phases, the algorithm extends the path by one vertex

## 19:8 An Algorithm for the Exact Treedepth Problem

at a time until the most-recently-visited vertex has no neighbours that are not on the path (lines 7 to 9). If  $\lceil \log_2(k + 1) \rceil$ , where  $k$  is the length of the constructed path, exceeds the target treedepth, the algorithm returns *true*.

■ **Algorithm 3** The path lower bound function.

---

```

1 can_prune_by_path_lower_bound( $G, targetDepth$ )
2 begin
3    $v \leftarrow \min(V(G))$ 
4    $P \leftarrow \{v\}$   $\triangleright P$  is the set of vertices on the path
5   repeat 2 times
6      $u \leftarrow v$ 
7     while  $u$  has a neighbour that is not contained in  $P$  do
8        $u \leftarrow$  the least such neighbour
9        $P \leftarrow P \cup \{u\}$ 
10  return  $\lceil \log_2(|P| + 1) \rceil > targetDepth$ 

```

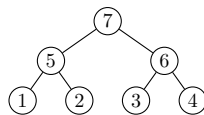
---

As an example, consider again the graph in Figure 1. The algorithm `path_lower_bound()` begins with  $v = 1$ , then greedily extends the path with vertices 2, 5, 3, 4, 9, and 8. As it is not possible to extend the path further from vertex 8, the algorithm returns to vertex 1 and prepends vertices 6 and 7 to the path. The path found is thus 7–6–1–2–5–3–4–9–8, which contains all nine of the graph’s vertices and gives a lower bound of  $\lceil \log_2(9 + 1) \rceil = 4$ . (In this case, this equals the treedepth of the graph, so our optimisation algorithm is able to determine that the graph has treedepth greater than 3 without any recursive calls on subgraphs).

Since we use bitset operations to iterate over the neighbours of  $u$  at line 7 of Algorithm 3, the algorithm runs in  $O(|V(G)|^2)$  time.

Our implementation has an additional small improvement to this lower bounding function. If the found path has three or more vertices, the program looks for a pair of vertices  $(v, w)$  that appear in the path, such that  $v$  and  $w$  are neighbours in  $G$  but do not appear side-by-side in the path. The section of the path from  $v$  to  $w$  thus forms a cycle, and it is possible to use the bound  $1 + \lceil \log_2(k) \rceil$ , where  $k$  is the number of vertices in the cycle [12]. The algorithm continues to visit all such pairs of vertices, stopping early if the calculated bound is greater than *targetDepth*. This cycle-finding extension of the bounding algorithm runs in  $O(|V(G)|^2)$  time, and thus does not increase the algorithm’s time complexity.

**Comparison of the two bounds.** Neither the simple lower bound nor the path lower bound dominates the other. We have already seen that for our example graph, the path lower bound is greater than the simple lower bound. Figure 5 is a graph for which the reverse is true. The graph has order 7 and maximum degree 3; therefore the simple lower bound is 3. The path lower-bounding function finds the path 1–5–2 which has length 3, giving a bound of 2.



■ **Figure 5** A graph for which the simple lower bound is greater than the path lower bound.



### 4.3 Initial Vertex Ordering by Degree

Before running our algorithm, the vertices of the input graph are reordered by non-increasing degree. We have observed that this leads to speed-ups on graph classes including binary trees and random graphs. We give two speculative reasons for this. First, it seems likely that for satisfiable values of the treedepth parameter  $k$ , we are most likely to find an elimination forest of depth  $k$  quickly by choosing high-degree vertices first, as these are most likely to split the remainder of the graph into small components. Second, the path-finding algorithm in our path lower bound function chooses low-numbered vertices first, and by preferring high-degree vertices it is less likely to run into “dead ends”.

## 5 Bitset Implementation

We use bitsets to represent sets, including rows of adjacency matrices. Induced subgraphs are not stored explicitly in memory; our program simply passes a pointer to the full graph along with a pointer to the set of vertices that induce the subgraph.

The space complexity of our algorithm compares favourably to that of the partitioning-based SAT encoding, which uses  $O(n^3)k$  clauses, where  $n = |V(G)|$ .

► **Proposition 2.** *Using a bitset implementation, the algorithm requires  $O(n^2)$  space.*

**Proof.** The bit-matrix representation of the input graph  $G$  requires  $O(n^2)$  space.

At most  $n + 1$  calls are made to `elimination_forest()`, and at most  $n$  calls are made to `elimination_tree()`, since line 25 of Algorithm 1 passes a graph with one vertex fewer than the graph passed to `elimination_tree()`. Each recursive call to these functions requires  $O(n)$  space, with the exception of the connected components found at line 6. Since each connected component is stored in an  $n$ -element bitset, it remains to show that at most  $O(n)$  connected components are stored at once.

We prove by induction that no more than  $n$  components are stored at one time. If `elimination_forest()` is called with a 1-vertex graph, then only one component is found, and no further components are found in recursive calls to the function. Now, let an  $n$ -vertex graph  $G$  be given, and assume that `elimination_forest()` and its recursive calls create no more than  $i$  components when called with any  $i$ -vertex graph ( $1 \leq i \leq n$ ). Let  $c$  be the number of components created by the initial call `elimination_forest(G, ...)`. Each of these components has at most  $n - c + 1$  vertices, since otherwise the components would have more than  $n$  vertices in total. If a component with  $m$  vertices is passed to `elimination_tree()`, then the recursive call to `elimination_forest()` at line 25 passes a graph with at most  $m - 1 \leq n - c$  vertices. By our inductive assumption, this call requires space for at most  $n - c$  components, and therefore at most  $c + n - c = n$  components in total are needed for the call to `elimination_forest(G, ...)`. ◀

## 6 Experiments

This section presents an experimental comparison with the partition-based SAT encoding [6, 7] which is the existing state of the art for the exact treedepth problem. We also investigate the effect of switching off individual features of our algorithm.

The experiments were performed on a cluster of five machines with dual Intel Xeon E5-2697A v4 CPUs and 512 GBytes of RAM, running Ubuntu 18.04. We implemented our algorithm in C, using Nauty version 2.6r11 for vertex-orbit symmetry breaking. The program

## 19:10 An Algorithm for the Exact Treedepth Problem

for the SAT encoding<sup>1</sup> is written in Python and calls an external SAT solver; we made the same choice as the authors of the encoding – the sequential version of Glucose 4.0 (which is based on MiniSAT [5]). Our program and Glucose were compiled with GCC at optimisation level -O3. Both our algorithm and the program for SAT encoding are single-threaded. A time limit of 1000 seconds per instance was used. We verified that all solvers that solved an instance within this time limit returned the same treedepth.

Following Ganian et al. [6, 7], we used three classes of instances – famous named graphs (many of which are regular and highly symmetrical), standard graphs (binary trees, cliques, complete bipartite graphs, cycle graphs, path graphs, and square grids), and random graphs.

■ **Table 1** Solving times in seconds for famous graphs. An asterisk indicates timeout at 1000 s.

Instance	$n$	$m$	$td$	All	–LB	–Sym	–Dom	SAT
Diamond	4	5	3	0.002	0.002	0.002	0.002	0.002
Bull	5	5	3	0.003	0.003	0.002	0.002	0.041
Butterfly	5	6	3	0.003	0.003	0.002	0.003	0.045
Prism	6	9	5	0.002	0.002	0.003	0.002	0.037
Moser	7	11	5	0.002	0.002	0.002	0.003	0.055
Wagner	8	12	6	0.002	0.002	0.002	0.003	0.067
Pmin	9	12	5	0.002	0.002	0.002	0.002	0.100
Petersen	10	15	6	0.002	0.002	0.002	0.002	0.129
Goldner	11	27	5	0.002	0.003	0.002	0.002	0.195
Grotzsch	11	20	7	0.003	0.003	0.003	0.002	0.187
Herschel	11	18	5	0.002	0.002	0.002	0.002	0.194
Chvatal	12	24	8	0.003	0.004	0.009	0.003	0.402
Durer	12	18	7	0.002	0.003	0.003	0.002	0.262
Franklin	12	18	7	0.002	0.002	0.004	0.003	0.273
Frucht	12	18	6	0.003	0.003	0.003	0.002	0.248
Tietze	12	18	7	0.003	0.003	0.003	0.002	0.266
Paley13	13	39	10	0.003	0.005	0.428	0.003	2.931
Poussin	15	39	9	0.005	0.009	0.101	0.005	2.478
Clebsch	16	40	10	0.005	0.020	0.974	0.004	14.147
Hoffman	16	32	8	0.003	0.010	0.013	0.003	1.500
Shrikhande	16	48	11	0.010	0.027	13.471	0.010	51.579
Sousselier	16	27	8	0.004	0.017	0.017	0.004	1.263
Errera	17	45	10	0.007	0.022	2.486	0.006	16.225
Paley17	17	68	14	0.056	0.072	*	0.052	*
Pappus	18	27	8	0.003	0.029	0.019	0.003	2.363
Robertson	19	38	10	0.018	0.365	3.441	0.021	43.926
Desargues	20	30	9	0.004	0.097	0.323	0.005	15.208
Dodecahedron	20	30	9	0.005	0.104	0.329	0.004	11.871
FlowerSnark	20	30	9	0.008	0.298	0.311	0.007	13.415
Folkman	20	40	9	0.004	0.056	0.118	0.007	10.071
Brinkmann	21	42	11	0.195	3.637	*	0.183	*
Kittell	23	63	12	0.405	3.094	*	0.559	*
McGee	24	36	11	0.219	24.042	344.762	0.175	*
Nauru	24	36	10	0.056	4.914	15.565	0.048	179.968
Holt	27	54	13	6.680	441.213	*	5.623	*
WatkinsSnark	50	75	13	*	*	*	870.345	*
B10Cage	70	105		*	*	*	*	*
Ellingham	78	117		*	*	*	*	*

**Famous graphs.** Table 1 shows run times in seconds for famous graphs. The second and third columns show the number of vertices and edges in each graph, and the fourth column shows the treedepth if it is known. The next four columns show run times for our algorithm;

<sup>1</sup> [https://github.com/nehali73/TCW\\_TD\\_to\\_SAT](https://github.com/nehali73/TCW_TD_to_SAT)

“All” has all features enabled, while “–LB”, “–Sym”, and “–Dom” have lower bounding, symmetry breaking, and domination rules turned off respectively. The final column shows run times for the partition-based SAT encoding [6, 7]. With all features on, our algorithm typically performs orders of magnitude faster than the SAT encoding. Both the symmetry breaking and lower bound features contribute to the algorithm’s performance, but on this set of benchmark instances, the domination rule slows the algorithm down slightly. With the rule switched off, the algorithm closes two open instances – the Holt and Watkins Snark graphs – both of which have treedepth 13.

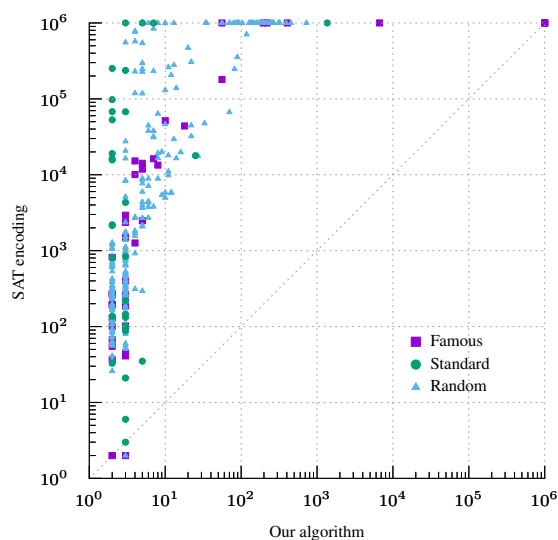
**Standard graphs.** Table 2 shows run times in seconds for standard instances. Again, our algorithm is typically much faster than the SAT encoding. The usefulness of the domination rule is demonstrated on these instances; with it switched off, the larger clique and bipartite instances could not be solved within the time limit. The lower bounding and symmetry breaking features also improve the run time on square grid graphs.

■ **Table 2** Solving times in seconds for standard graphs. An asterisk indicates timeout at 1000 s.

Instance	$n$	$m$	$td$	All	–LB	–Sym	–Dom	SAT
Binary tree 10	10	9	3	0.003	0.003	0.002	0.002	0.088
Binary tree 20	20	19	4	0.002	0.003	0.002	0.002	0.804
Binary tree 30	30	29	5	0.003	0.004	0.002	0.003	4.319
Binary tree 40	40	39	5	0.002	0.008	0.002	0.002	19.021
Binary tree 50	50	49	5	0.002	0.040	0.002	0.003	52.950
Clique10	10	45	10	0.003	0.002	0.003	0.003	0.003
Clique20	20	190	20	0.003	0.003	0.002	0.879	0.006
Clique30	30	435	30	0.003	0.003	0.003	*	0.220
Clique40	40	780	40	0.003	0.004	0.003	*	0.021
Clique50	50	1225	50	0.005	0.004	0.004	*	0.035
Complete bipartite 10	10	25	6	0.003	0.003	0.003	0.002	0.132
Complete bipartite 20	20	100	11	0.002	0.003	0.003	0.012	97.689
Complete bipartite 30	30	225	16	0.003	0.004	0.008	20.394	*
Complete bipartite 40	40	400	21	0.005	0.005	0.194	*	*
Complete bipartite 50	50	625	26	0.007	0.010	7.565	*	*
Cycle 10	10	10	5	0.002	0.003	0.003	0.003	0.137
Cycle 20	20	20	6	0.002	0.004	0.002	0.003	2.194
Cycle 30	30	30	6	0.002	0.007	0.002	0.002	16.151
Cycle 40	40	40	7	0.002	0.409	0.002	0.002	67.870
Cycle 50	50	50	7	0.003	0.762	0.002	0.002	236.847
Path 10	10	9	4	0.003	0.003	0.003	0.003	0.146
Path 20	20	19	5	0.002	0.003	0.003	0.002	2.146
Path 30	30	29	5	0.002	0.010	0.003	0.003	15.688
Path 40	40	39	6	0.003	0.181	0.003	0.003	67.505
Path 50	50	49	6	0.002	0.405	0.002	0.002	251.987
Square grid $2 \times 2$	4	4	3	0.002	0.003	0.003	0.003	0.033
Square grid $3 \times 3$	9	12	5	0.003	0.003	0.003	0.002	0.100
Square grid $4 \times 4$	16	24	7	0.003	0.004	0.003	0.002	0.841
Square grid $5 \times 5$	25	40	9	0.025	2.219	0.797	0.026	17.869
Square grid $6 \times 6$	36	60	11	1.363	*	*	1.619	*

**Random graphs.** We generated random graphs using the Erdős-Rényi  $G(n, p)$  model, with  $n \in \{12, 16, 20\}$  vertices and edge probabilities  $p \in \{0.1, 0.2, \dots, 0.9\}$ . Ten instances were generated for each  $n, p$  pair. Our algorithm solved each of the 270 instances in less than 0.3 seconds per instance; the SAT encoding exceeded the time limit of 1000 seconds on 53 of the 90 instances with 20 vertices.

**Summary of experimental results.** Figure 6 summarises, for all instances, the run times of our algorithm and the SAT encoding. Each point shows the two algorithms' run times for a single instance. Timeouts are shown as 1000 seconds. For the harder instances that take more than ten seconds to solve with the SAT encoding, our algorithm is typically around three orders of magnitude faster.



■ **Figure 6** Run times in milliseconds. Each point represents one instance.

We end this section by noting that the SAT-encoding program writes each SAT instance it generates to disk, whereas our program performs no disk I/O while solving. For the famous graphs, the cost of this disk I/O does not meaningfully affect our comparison of run times; on the three most difficult famous instances that can be solved by the SAT-based program (Nauru, Shrikhande and Robertson), over 95% of the total run time is spent within the SAT solver on a single unsatisfiable instance of the SAT problem. For some of the standard and random graphs, such as complete bipartite graphs, a similar pattern holds. But for other graphs in these classes, such as binary trees, the SAT-based program spends most of its time creating encodings and writing them to disk, and for these instances it is likely that the program could be improved significantly by avoiding writing to disk.

## 7 Conclusion

We have introduced an algorithm for computing the exact treedepth of a graph, and shown experimentally that it runs orders of magnitude faster than the current state of the art on a varied set of benchmark instances. The core of the algorithm is a simple pair of mutually-recursive functions. To this basic algorithm, we have added symmetry breaking, domination, lower bounding, and vertex ordering rules. There is room for further improvement to each of these extensions of the algorithm.

There is also scope for improvement in finding a good upper bound on the treedepth quickly. SAT encodings of the problem have advantages in this regard: modern SAT solvers implement restarts and good heuristics, both of which help to find good solutions quickly. Future research could combine the benefits of SAT solvers with the techniques introduced in this paper, either by including some of the techniques in a SAT model or by adding features such as periodic restarts to our algorithm.

## References

- 1 Bengt Aspvall and Pinar Heggernes. Finding minimum height elimination trees for interval graphs in polynomial time. *BIT Numerical Mathematics*, 34(4):484–509, December 1994. doi:10.1007/BF01934264.
- 2 Hans L. Bodlaender, Jitender S. Deogun, Klaus Jansen, Ton Kloks, Dieter Kratsch, Haiko Müller, and Zsolt Tuza. Rankings of graphs. *SIAM J. Discrete Math.*, 11(1):168–181, 1998. doi:10.1137/S0895480195282550.
- 3 Jitender S. Deogun, Ton Kloks, Dieter Kratsch, and Haiko Müller. On vertex ranking for permutations and other graphs. In Patrice Enjalbert, Ernst W. Mayr, and Klaus W. Wagner, editors, *STACS 94, 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen, France, February 24-26, 1994, Proceedings*, volume 775 of *Lecture Notes in Computer Science*, pages 747–758. Springer, 1994. doi:10.1007/3-540-57785-8\_187.
- 4 Jitender S. Deogun, Ton Kloks, Dieter Kratsch, and Haiko Müller. On the vertex ranking problem for trapezoid, circular-arc and other graphs. *Discrete Applied Mathematics*, 98(1-2):39–63, 1999. doi:10.1016/S0166-218X(99)00179-1.
- 5 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3\_37.
- 6 Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for treecut width and treedepth. In Stephen G. Kobourov and Henning Meyerhenke, editors, *Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019.*, pages 117–129. SIAM, 2019. doi:10.1137/1.9781611975499.10.
- 7 Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-encodings for treecut width and treedepth. *CoRR*, abs/1911.12995, 2019. arXiv:1911.12995.
- 8 Chris Groër, Blair D Sullivan, and Dinesh Weerapurage. Inddgo: Integrated network decomposition & dynamic programming for graph optimization. *ORNL/TM-2012*, 176, 2012.
- 9 Gregory Z. Gutin, Mark Jones, and Magnus Wahlström. The mixed Chinese postman problem parameterized by pathwidth and treedepth. *SIAM J. Discrete Math.*, 30(4):2177–2205, 2016. doi:10.1137/15M1034337.
- 10 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60(0):94–112, 2014. doi:10.1016/j.jsc.2013.09.003.
- 11 Asier Mujika. About tree-depth. Bachelor’s thesis, Universidad del País Vasco, 2015.
- 12 Jaroslav Nesetril and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012. doi:10.1007/978-3-642-27875-4.
- 13 A. Pothén. The complexity of optimal elimination trees. Technical Report CS 88-16, Department of Computer Science, Penn State, 1988.
- 14 Felix Reidl, Peter Rossmanith, Fernando Sánchez Villaamil, and Somnath Sikdar. A faster parameterized algorithm for treedepth. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 931–942. Springer, 2014. doi:10.1007/978-3-662-43948-7\_77.
- 15 Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve Linton. Tractable symmetry breaking using restricted search trees. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI’2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 211–215. IOS Press, 2004.

## 19:14 An Algorithm for the Exact Treedepth Problem

- 16 Alejandro A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, 1989. doi:10.1016/0020-0190(89)90161-0.
- 17 Earl Zmijewski and John R Gilbert. A parallel algorithm for large sparse Cholesky factorization on a multiprocessor. Technical Report TR 86-733, Cornell University, Ithaca, NY, 1986.