PACE Solver Description: tdULL

Ruben Brokkelkamp
Centrum Wiskunde & Informatica (CWI), The Netherlands
ruben.brokkelkamp@cwi.nl

Raymond van Venetië
Korteweg–de Vries Institute, University of Amsterdam, The Netherlands
r.vanvenetie@uva.nl

Mees de Vries
University of Amsterdam, The Netherlands
meesdevries@protonmail.com

Jan Westerdiep
Korteweg–de Vries Institute, University of Amsterdam, The Netherlands
j.h.westerdiep@uva.nl

──── **Abstract** ────

We describe tdULL, an algorithm for computing treedepth decompositions of minimal depth. An implementation was submitted to the exact track of PACE 2020. tdULL is a branch and bound algorithm branching on inclusion-minimal separators.

1 Introduction

The treedepth of an undirected graph is a measure of the complexity of the graph. Informally, it measures how resistant the graph is to being disconnected by removing vertices. All graphs have a treedepth between 1 and their number of vertices. Star graphs $K_{n,1}$, which can be completely disconnected by removing a single vertex, have treedepth 2. Trees have a treedepth at most logarithmic in their size. Complete graphs K_n have full treedepth of n. In general, computing the treedepth of a graph is NP-complete.

The PACE 2020 challenge consists of implementing an algorithm that is capable of computing treedepth. In the exact track, to which the solver tdULL was submitted, the goal was to compute the exact treedepth of 100 graphs from an unknown set, with a time limit of 30 minutes per graph. In order to test implementations, a set of 100 different but representative graphs was published at the start of the contest.

There are many equivalent definitions of treedepth. The following is useful for our purposes. For S a set of vertices of G, we write $G \setminus S$ for the graph obtained from G by removing the vertices from S and any incident edges. For a graph G, we write $\text{cc}(G)$ for the set of its connected components.

▶ **Definition 1** (Treedepth). *The treedepth* $\mathrm{td}(G)$ *of a connected graph* $G = (V, E)$ *is recursively defined as*

$$\mathrm{td}(G) := \begin{cases} 1 & \textit{if } |V| = 1, \\ \min_{v \in V} \max_{H \in \mathrm{cc}(G \setminus \{v\})} 1 + \mathrm{td}(H) & \textit{else.} \end{cases}$$

The minimizing vertex v in the recursion can be taken as the root of a tree, with the connected components of $G \setminus \{v\}$ its children; this way, the computation of treedepth gives rise to a tree, the *treedepth decomposition* of G. The depth of this tree is the treedepth of G.

If removal of v does not cause the graph to be disconnected, the next (recursive) step is again the removal of a single vertex, until enough vertices have been removed to disconnect the graph. This leads to the following alternative definition. A set S of vertices of a connected graph G is called a *separator* of G if $G \setminus S$ is disconnected. Such a set S is called *inclusion minimal* if there is no $S' \subsetneq S$ which separates G. Write $\mathrm{sep}(G)$ for the inclusion-minimal separators of a graph G.

▶ **Definition 2** (Treedepth). *The treedepth* $\mathrm{td}(G)$ *of a connected graph* $G = (V, E)$ *is recursively defined as*

$$\mathrm{td}(G) := \begin{cases} |V| & \textit{if } G \cong K_{|V|}, \\ \min_{S \in \mathrm{sep}(G)} \max_{H \in \mathrm{cc}(G \setminus S)} |S| + \mathrm{td}(H) & \textit{else.} \end{cases}$$

With this definition, tdULL can be described in one sentence as a branch-and-bound algorithm based on Definition 2, with heuristics, clever data structures and exact special cases used for speed-up. The advantage of recurring on separators rather than on single vertices is avoiding duplicate branches: if we recur on a separator of size n, a vertex-based recursion may have $n!$ branches leading to that same point.

2 Algorithm

2.1 Branch and bound

To prune the search tree we use branch and bound. If we have found a treedepth decomposition of depth d for a particular graph, then any further search for a treedepth decomposition of that graph need not continue with any attempt that will have treedepth at least d. Similarly, if we have picked a separator S, and for $H \in \mathrm{cc}(G \setminus S)$ we have that $\mathrm{td}(H) = d$, then there is no need for us to find a treedepth decomposition for $H' \in \mathrm{cc}(G \setminus S)$ of depth lower than d, since it will not make the depth of the full decomposition any lower.

To this end, the main component of tdULL is a function `treedepth`, whose arguments are a graph G, as well as a *search lower bound* (SLB) and a *search upper bound* (SUB). The interval between these two bounds is the interval of treedepths dat are still relevant. The function `treedepth` returns a lower bound l and an upper bound u on the treedepth of G, for which at least one of the following three things is true:

- $u < \mathrm{SLB}$: the treedepth of G is so low we do not need the exact value;
- $l > \mathrm{SUB}$: the treedepth of G is so high we do not need the exact value;
- $u = l$: the treedepth of G is equal to $u = l$.

The main loop of `treedepth` generates the separators of the graph G, and recursively calls itself on the components left after removing the separator. The search bounds allow us to skip computation that is provably not going to improve the treedepth of the graph.

2.1.1 Lower bounds

Upper bounds on the treedepth of the graph G can be found directly, by completing branches of the recursion. Lower bounds are harder to find: the only sure way to find a lower bound is to recur on all possible separators, and conclude that a smaller treedepth cannot be realized.

To skip as many of the branches as possible, it is therefore important that we obtain good lower bounds as quickly as possible. Our main tool is the fact that the treedepth of a graph is *minor monotone*: if H is a graph that can be obtained from G by removing vertices and edges and contracting edges, then $\text{td}(H) \leq \text{td}(G)$. We apply this principle in a number of ways.

Any lower bound on the treedepth of a subgraph returned by the recursion is also a lower bound on $\text{td}(G)$. In rare cases, this lower bound may actually be equal to the treedepth of G, allowing us to short-circuit the rest of the computation.

We pick specific subgraphs of G of which we compute the treedepth. The most important of these is a *core*: we take the vertex of lowest degree of G, and iteratively remove all vertices from G that have at most that degree. In the resulting graph every vertex has higher degree. Heuristically, this core should be the "toughest subset" of G, and thus provide good lower bounds.

If this core is empty, then we compute a contraction of the graph G instead, and try to find its treedepth to use as a lower bound. Specifically, we contract the edge between a vertex of minimal degree and a neighboring vertex for which the overlap of common neighbors is minimal. We experimented with other contraction strategies, but they proved less effective.

2.2 Separators

In order to use the recursion suggested by Definition 2, we need to be able to generate inclusion-minimal separators of the graph G. Analysis of the runtime of tdULL suggests that most of the runtime is spent on this generation process.

We essentially use the algorithm from [1]. This is an algorithm that produces minimal separators, which are subtly different from inclusion-minimal separators. A separator S is called minimal if there are vertices $v, w \in G \setminus S$ such that v, w are in different components after removing S, and there are no $S' \subsetneq S$ which separate v, w. It is easy to check whether a minimal separator S is also inclusion minimal: S is inclusion minimal if and only if each vertex in S has an edge to each connected component of $G \setminus S$. Thus we can use the algorithm for minimal separators, and filter out those which are not inclusion minimal.

We do not generate all separators of a graph at once, but in batches of 10,000. This occasionally helps us to avoid having to compute all the separators, by finding both an optimal decomposition and an optimal lower bound early. We tried several batch size strategies, both fixed and dynamic, and this one worked best.

A batch of separators is not tried in arbitrary order: we sort the separators by the size of the largest component that remains after removing the separator. This prioritizes separators that appear to be efficient at disconnecting the graph, which are heuristically more likely to lead to optimal solutions. We tried several sorting strategies, this one appeared to work best.

If G has a leaf (degree one vertex), then the one neighbor of that leaf forms an inclusion-minimal separator by itself. If a graph has a leaf attached to (nearly) every other vertex, our inclusion-minimal separators are (almost) the same as the vertices of degree greater than one. Then the recursion turns into the one from Definition 1, which is much less efficient. To avoid this problem, we actually compute separators on the graph with all leaves removed. Since leaves are never useful as the root of a treedepth decomposition, these separators suffice.

2.3   Special cases

There are a few cases in which we can quickly find an optimal treedepth decomposition of a graph G. We can easily recognize these cases and apply the faster direct algorithm.

- $G \cong K_n$ for some n;
- $G \cong C_n$ for some n;
- G is a tree (with the algorithm described in [2]).

2.4   Cache

Removing the same set of vertices in a different order results in the same graph. To avoid double work we store all lower and upper bounds on the treedepth of subgraphs in a cache. To quickly add, update and retrieve items from the cache we make use of a *SetTrie* data structure, as described in [3]. Furthermore, this data structure allows for efficient retrieval of subsets and hence can be used to compute lower bounds: because $\mathrm{td}(H) \leq \mathrm{td}(G)$ if $H \subseteq G$, we can use a lower bound on the treedepth of H as a lower bound on the treedepth of G.

This cache also works with contractions: every vertex created by a contraction has a canonical representation as the set of vertices which are contracted. When a new combination of vertices is used for a contraction, it gets a new global index to use in the SetTrie cache.

3   Discussion

The above algorithm is the result of a trial-and-error process, where a big subset of the public instances was used to compare versions. Comparing the number of cases solved by us and by the submissions above us on both the public and private test sets, it seems we may have suffered from some overfitting: we made algorithmic choices that performed better on the public instances of the problem than on the hidden instances.

Many ideas did not make it to the final algorithm. One notable omission is using the symmetry of a graph. Neither using graph automorphisms to reduce the number of separators nor trying to use a cache that works up to isomorphism yielded any improvement.

tdULL spends a lot of time on generating separators. If a graph has many – in the worst case, there may be exponentially many – this pushes us quickly over the time limit. Small improvements such as only using separators of the graph where all leaves are removed certainly helped, but we we have not found a way to substantially reduce the number of separators. Switching from a single vertex recursion to a separator-based recursion has been the biggest performance improvement, though.

Another improvement which yielded big performance improvements was finding better subgraphs with which to compute lower bounds. Even when a graph has a lot of separators, if for some separator the upper bound from the recursion matches the lower bound found before, the algorithm finishes fast because of an early exit.

References

1   Anne Berry, Jean-Paul Bordat, and Olivier Cogis. Generating all the minimal separators of a graph. In *Graph-Theoretic Concepts in Computer Science*, pages 167–172, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
2   Ananth V. Iyer, H. Donald Ratliff, and G. Vijayan. Optimal node ranking of trees. *Inf. Process. Lett.*, 28(5):225–229, August 1988. `doi:10.1016/0020-0190(88)90194-9`.
3   Iztok Savnik. Index data structure for fast subset and superset queries. In *International Conference on Availability, Reliability, and Security*, pages 134–148. Springer, 2013.