PACE Solver Description: Sallow: A Heuristic Algorithm for Treedepth Decompositions

Marcin Wrochna
University of Oxford, UK
mwrochna@gmail.com

Abstract

We describe a heuristic algorithm for computing treedepth decompositions, submitted for the PACE 2020 challenge. It relies on a variety of greedy algorithms computing elimination orderings, as well as a Divide & Conquer approach on balanced cuts obtained using a from-scratch reimplementation of the 2016 FlowCutter algorithm by Hamann & Strasser [2].

2012 ACM Subject Classification Theory of computation → Discrete optimization; Mathematics of computing → Solvers; Mathematics of computing → Graph theory

Keywords and phrases treedepth, decomposition, heuristic, weak colouring numbers

Digital Object Identifier 10.4230/LIPIcs.IPEC.2020.36

Related Version A full version of the paper is available at `https://arxiv.org/abs/2006.07050`.

Supplementary Material Code: `http://doi.org/10.5281/zenodo.3870565`.

Funding This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 714532, PI: Stanislav Živný).

Acknowledgements The author is very grateful to the PACE 2020 organizers at the University of Warsaw, organizers of past editions, and the OPTIL.io team at the Poznań University of Technology for making this challenge possible.

1 Orderings and elimination

We start by recalling a few useful notions and facts (experts will recognize we are essentially describing the well-known statement that $\mathrm{td}(G) = \mathrm{wcol}_\infty(G)$, see e.g. [3, Lemma 6.5]).

Treedepth has many equivalent definitions. Small treedepth can be certified as usual by a treedepth decomposition (also known as a Trémaux tree) – it suffices to specify the `parent` of each vertex in the tree. A corresponding `ordering` is any linear ordering of vertices such that parents come before children – it can be obtained from a `parent` vector by any topological sorting algorithm, for example. In turn, any linear `ordering` of vertices can be turned into a treedepth decomposition by an elimination process: repeatedly remove the last vertex in the ordering and turn its neighbourhood into a clique. The `parent` of the removed vertex is set to the latest vertex in the neighbourhood.

It is easy to check this results in a new valid treedepth decomposition. Moreover, turning a decomposition into an ordering and back cannot increase the depth. To see this, observe that by induction, at any point in the elimination process, the neighbourhood of the removed vertex consists only of its ancestors (in the original decomposition), because all later vertices were removed, hence all introduced edges are still in the ancestor-descendant relationship. In particular the new parent of each vertex is an ancestor in the original decomposition.

A more static look at the elimination process is through *strongly* and *weakly* reachable vertices. Fix a vertex v. A vertex x is in the neighbourhood of v at the moment v is eliminated if and only if x is earlier in the ordering and can be reached, in the original graph, via a path whose internal vertices are later than v in the ordering (= have been eliminated). We say x is *strongly reachable* from v (in the given graph and ordering). So this neighbourhood is the set of strongly reachable vertices (in the graph G with ordering L), usually denoted $\mathrm{SReach}_\infty[G, L, v]$. Similarly x is *weakly reachable* from v if x is earlier and can be reached via a path whose internal vertices are later than x (instead of "later than v"). Equivalently, this relation is the transitive closure of strong reachability. The set of weakly reachable vertices is denoted $\mathrm{WReach}_\infty[G, L, v]$ – it is equal to the set of ancestors of v in the treedepth decomposition obtained by elimination.

To give some context, the maximum size of $\mathrm{SReach}_\infty[G, L, v]$ over vertices v is the *strong ∞-colouring number* of (G, L) and its minimum over all orderings L is equal to $\mathrm{tw}(G) + 1$ [1, Theorem 3.1]. The maximum size of $\mathrm{WReach}_\infty[G, L, v]$ over vertices v (hence the depth of the resulting treedepth decomposition) is the *weak ∞-colouring number* of (G, L) and its minimum over all orderings L is equal to $\mathrm{td}(G)$. The ∞ here is customary because using paths of length at most $k < \infty$ leads to other notions important in sparse graph theory, see e.g. [6].

A third, more efficient look at the elimination process comes from the observation that descendants of v, in the resulting treedepth decomposition, are exactly vertices reachable in the subgraph induced by vertices later than or equal to v (in the ordering). We can thus define a *building process* on an ordering as follows: we process vertices starting from the last, maintaining connected components of the subgraph induced by vertices processed so far. This is sufficient to build the same treedepth decomposition, without changing the graph: we maintain the treedepth decomposition of the subgraph induced by processed vertices (using a `parent` vector) and represent each component by the root of the corresponding tree (equivalently, the earliest vertex of the component). When processing a vertex v, for each neighbour y later than v in the ordering, to update components we only have to merge y's component with v (initially a singleton). To update the decomposition, we find the root of y's component and make it a child of v, which thus becomes the new root. Thus y's parent is the latest weakly reachable vertex, as expected.

In other words, the building process consider vertices in the same order as the elimination process. However, in the elimination process we maintain a graph on unprocessed vertices and when eliminating the next vertex, we replace its neighbourhood by a clique, which is somewhat costly. Instead, in the building process we maintain a graph on more and more processed vertices (hence the name), or rather a structure to represent its connected components.

2 Greedy algorithms

The above processes suggest simple heuristics: we can start with vertices ordered decreasingly by any notion of centrality (e.g., the degree in the original graph), since we expect higher vertices in optimal treedepth decompositions to be more "central". Moreover, we can update this "centrality score" of unprocessed vertices on the fly: we maintain a heap of unprocessed vertices with the minimum score at the top, popping and processing a vertex until the heap is empty.

By elimination

In the elimination process, we update the score of a vertex v based on a linear combination of: 1. its height ($1+\max$ height of neighbours eliminated so far; once we decide to eliminate v this becomes the height of the subtree rooted at v in the resulting decomposition); 2. its degree (in the partially eliminated graph; once we decide to eliminate v this becomes $|\mathrm{SReach}_\infty[G, L, v]|$ in the resulting ordering); 3. some initial, static score.

Consider a graph with n vertices, m edges, and suppose we stop when unable to obtain a decomposition of depth better than d. In the elimination process, we can then assume that the neighbourhood of every vertex at every step is smaller than d. Simulating it then requires $\Theta(nd^2)$ time in many cases: e.g. if most vertices have a neighbourhood of size $\Theta(d)$ at the time they are processed (as in $K_{d,n}$), ensuring that this neighbourhood becomes a clique takes $\Theta(d^2)$ time. This bound is also sufficient; to do the simulation we maintain neighbourhood lists of the partially eliminated graph as `std::vector`s sorted by vertex name (not by the ordering we're about to compute), so that the union of neighbourhoods can be computed by merge-sort (when turning v's neighbourhood into a clique). Note however that we cannot compute parents on the fly, since the ordering of unprocessed vertices is not yet decided; we do this in a second pass, using the faster building approach once the ordering is fixed. See Algorithm 1 in the full version.

The memory requirement is $\Theta(nd)$ in the worst case and this cannot be improved to $\mathcal{O}(m)$, because a random 3-regular graph on d vertices will have, after eliminating half of its vertices, a clique on $\Theta(d)$ vertices and $\Theta(d^2)$ edges (due to its expansion properties). This means memory usage can also be prohibitive for large graphs with expected treedepth d much larger than the average degree.

By building

The $\Theta(nd^2)$ time and $\Theta(nd)$ space bound of the elimination process can be prohibitive for huge graphs of large treedepth. Instead, the building process can be simulated in $\mathcal{O}(\min(m \cdot \alpha(n), nd))$ time and $\mathcal{O}(m)$ space by maintaining components with the classic union-find data structure (where α is the inverse Ackerman function [5] and the latter bound follows from the fact that for each vertex, its pointer in the structure only goes up the tree). We note that this also allows to check the correctness of a treedepth decomposition (by replacing the assignment $\texttt{parent}[y] := v$ with whatever the original parent was and checking it is a descendant of v) in the same running time; this can be significantly faster than the straightforward $\mathcal{O}(md)$ method when d is large.

The details are similar as in Algorithm 1, see Algorithm 2 in the full version. One change is that $\texttt{g}[v]$ does not represent the neighbourhood of a vertex after elimination; instead, it represents the graph after contracting processed components. For a root vertex r of a component C (starting from singleton components), $\texttt{g}[r]$ stores the neighbours of that component. For a non-root vertex $\texttt{g}[v]$ is cleared: this guarantees that the total size never increases. This also means the α part of the score is less meaningful; using $\alpha \cdot |g[x]|$ below would be the same as $\alpha \cdot |N_G(x)|$ (the original degree, since x is not processed yet). Instead we use $\alpha \cdot \max(|g[x]|, |g[v]|)$ as a slightly better heuristic. This does result in noticeably worse results compared to the elimination version.

Moreover, we maintain a union-find structure with pointers $\texttt{ancestor}[v]$. We decided not to balance unions by size or rank, instead of opting for a simpler and more natural choice: $\texttt{ancestor}[v]$ is always some ancestor in the treedepth decomposition computed so far ($\texttt{ancestor}[v]$ is $\perp$ for unprocessed vertices) – we expect these to be shallow anyway.

Fast versions with lookahead

In Algorithm 2 the cost of computing $\texttt{g}[v]$ is still significant (though much lower compared to the elimination version). A super-fast version can be obtained by removing $\texttt{g}[v]$; however the `height` of unprocessed vertices cannot be maintained exactly then. In that case the heap is useless and we can simply do the building process with a fixed ordering by initial score.

However, we can significantly improve this super-fast version with a simple lookahead. Instead of processing the last unprocessed vertex, we check the last ℓ unprocessed vertices, compute what their height would be at this point, and choose the minimum height. For $\ell = 2$ this is almost as fast as a DFS; for $\ell = 64$ this is still faster than other versions and results in significantly better depth than DFS, often giving a reasonable ballpark estimate. Nevertheless we essentially always use the full version of Algorithm 2 as well, unless we know that the super-fast estimates are good enough (e.g. in recursive runs where other branches are already deeper).

A similar idea can be used to get the best of the elimination and building versions. The problem with the building version is that we do not have access to the degree of a vertex after eliminations, for evaluating the heuristic score. A work-around is to do this evaluation exactly (by computing unions of neighbourhoods) for a few vertices close to the top of the heap. In fact, a re-evaluation can only increase the score, pushing a vertex down, so it suffices to re-evaluate and update the top vertex of the heap some constant ℓ number of times (completely forgetting the computed unions of neighbourhoods afterwards). We can stop as soon as the top vertex stays at the top after re-evaluation, so even high constants ℓ turn out to be quite affordable. For $\ell = 1024$, this results in an algorithm that seems just as good as greedy by elimination, yet avoids the heavy memory usage in huge graphs of large treedepth.

3 Divide & Conquer

The other main component of the submitted algorithm is to divide & conquer: find a possibly small balanced cut, remove it, recurse into connected components, and output a treedepth decomposition with the cut arranged in a line above the recursively obtained decompositions. To find balanced cuts we use the FlowCutter algorithm submitted for the PACE 2016 challenge by Ben Strasser. It is a crucial part here as well, but the idea and details are already very well described in a paper by Hamman and Strasser [2] (see also some further details in [4]).

A final feature is that of cutoffs. A bad cutoff d means we abandon any attempt that won't lead to a decomposition of depth strictly smaller than d. A good cutoff d means we return as soon as it can output a decomposition of depth at most d. We use e.g. the best know decomposition's depth as a bad cutoff and the maximum depth in sibling branches computed so far as a good cutoff.

Further ideas and details are deferred to the full version (`arXiv:2006.07050`).

References

1 Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability – a survey. *BIT Numerical Mathematics*, 25(1):2–23, 1985. `doi:10.1007/BF01934985`.

2 Michael Hamann and Ben Strasser. Graph bisection with pareto optimization. *ACM Journal of Experimental Algorithmics*, 23, 2018. `doi:10.1145/3173045`.

3 Jaroslav Nešetřil and Patrice Ossona de Mendez. Bounded height trees and tree-depth. In *Sparsity*, pages 115–144. Springer, 2012. `doi:10.1007/978-3-642-27875-4_6`.

4 Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. `arXiv:1709.08949`.

5 Robert Endre Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, 1984. `doi:10.1145/62.2160`.

6 Jan van den Heuvel, Patrice Ossona de Mendez, Daniel Quiroz, Roman Rabinovich, and Sebastian Siebertz. On the generalised colouring numbers of graphs that exclude a fixed minor. *Eur. J. Comb.*, 66:129–144, 2017. `doi:10.1016/j.ejc.2017.06.019`.