Sallow – a heuristic algorithm for treedepth decompositions

Marcin Wrochna
University of Oxford

June 2020

Code: github.com/marcinwrochna/sallow/releases/tag/v1.0
Code DOI: 10.5281/zenodo.3870565

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 [HS18].

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. [NM12, 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. This implies in particular that 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 $\mathrm{tw}(G) + 1$ [Arn85, 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 considering only paths of length at most k leads to similar definitions of strong and weak k-colouring numbers, which are important in the theory of sparse graphs, see e.g. [Heu+17].

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 process 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.

2 Greedy algorithms

The elimination and building processes suggest heuristics for finding treedepth decomposition. For example, 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 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.

2.1 By elimination

In the elimination process, we update the score of a vertex v based on (a linear combination of)

- its height (one plus the 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);
- its degree (in the partially eliminated graph; once we decide to eliminate v this becomes $|\operatorname{SReach}_\infty[G, L, v]|$ in the resulting ordering);
- 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 and space in the worst case, because the graph can have that many edges in the middle of the process. 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 below.

For the initial score, the final implementation only uses the height of vertices in the best previously obtained decomposition. For the parameters α, β, γ that give the linear combination defining the score, the best choice for a single run seems to be to choose non-zero α, much larger β, while γ can be zero (e.g. $\langle 1, 9, 0\rangle$). On the other hand, repeating the algorithm with a variety of different parameters often finds significantly better solutions.

Algorithm 1: Greedy by elimination on graph G with parameters α, β, γ and initial `score`

```
for v in V(G) do
    g[v] := copy of N_G(v)
ordering := ⟨⟩
height := ⟨1, ..., 1⟩
initialize heap with v ↦ α·|g[v]| + β·height[v] + γ·score[v]
while heap not empty do
    v := pop minimum element from top of heap
    ordering := v, ordering
    for x in g[v] do
        g[x] := (g[x] − v) ∪ (g[v] − x)
        height[x] := max(height[x], height[v] + 1)
        heap.update(x ↦ α·|g[x]| + β·height[x] + γ·score[x])
    clear g[v]
root := first in ordering
depth := height[root]
compute parent from ordering
```

2.2 By building

The $\Theta(nd^2)$ time and 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 by maintaining components with the classic union-find data structure (where α is the inverse Ackerman function [TL84] 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 `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. One change is that `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), `g`$[r]$ stores the neighbours of that component. For a non-root vertex `g`$[v]$ is cleared (it is important to actually free the memory using `std::vector::shrink_to_fit()`; calling `clear()` keeps the capacity unchanged). 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 `ancestor`$[v]$. We decided not to balance unions by size or rank, instead of opting for the more natural choice: `ancestor`$[v]$ is always some ancestor in the treedepth decomposition computed so far (`ancestor`$[v]$ is $\perp$ for unprocessed vertices). This theoretically spoils the $m\alpha(n)$ running time guarantee, but simplifies the implementation, and we expect the trees to be shallow anyway.

2.3 Super-fast version with lookahead

In Algorithm 2 the cost of computing `g`$[v]$ is still significant (though much lower compared to the elimination version). A super-fast version can be obtained by removing `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.

Algorithm 2: Greedy by building on graph G with parameters α, β, γ and initial `score`

```
for v in V(G) do
    g[v] := copy of N_G(v)
ordering := ⟨⟩
parent := ⟨⊥, ..., ⊥⟩
ancestor := ⟨⊥, ..., ⊥⟩
height := ⟨1, ..., 1⟩
initialize heap with v ↦ α·|g[v]| + β·1 + γ·score[v]
while heap not empty do
    v := pop minimum element from top of heap
    ordering := v, ordering
    ancestor[v] := v
    for y in N_G(v) do
        if ancestor[y] ≠ ⊥ then
            r := find(y) in the ancestor union-find structure
            if r ≠ v then
                parent[r] := v
                ancestor[r] := u    (union)
                g[v] := (g[r] − v) ∪ (g[v] − y)
                clear g[r]
            else
                g[v] := g[v] − y
    for x in g[v] do
        height[x] := max(height[x], height[v] + 1)
        heap.update(x ↦ α·max(|g[x]|, |g[v]|) + β·height[x] + γ·score[x])
```

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).

2.4 By building with lookahead

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.

All in all, on the public instances of the PACE 2020 challenge, the total score (sum of $\frac{\text{best known depth}}{\text{algorithm's depth}}$ over all tests) and total running time was roughly (with $\langle \alpha, \beta, \gamma \rangle = \langle 1, 9, 0 \rangle$):

- 37% in 1 minute for the "super-fast greedy" version with $\ell = 64$,
- 63% in 8 minutes for the basic "greedy by building" version (Algorithm 2),
- 72% in 15 minutes for the "building with lookahead $\ell = 1024$" version,
- 86% in 3000 minutes for the "building with lookahead $\ell = 1024$" version running 30 minutes on each instance with various $\langle \alpha, \beta, \gamma \rangle$,
- 96% in 3000 minutes for the final algorithm with Divide & Conquer.

(single-thread on a i7-8550U CPU with `-O3 -march=ivybridge -flto`).

3 Divide & Conquer

The other main component of the submitted algorithm is a simple divide & conquer idea: 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 [HS18] (see also some further details in [Str17]). In this submission it is only perhaps noteworthy that the algorithm has been reimplemented from scratch, in an attempt to optimize it (it is after all the main bottleneck on most large instances). The only semantic difference however is that the new implementation works directly on vertex cuts and vertex flows. Unfortunately, this adds some technical complexity, since for example augmenting paths of vertex flows can revisit a vertex twice. An actual experimental comparison remains to be done, but this seems to matter less than how the cuts are actually used.

Due to time constraints of the author, the final algorithm uses the greedy and FlowCutter components in suboptimal ways, in a rather fragile and unprincipled patchwork. It starts with greedy algorithms and then takes the first cut of at least a given balance yielded by FlowCutter and recurses. Afterwards FlowCutter is ran again, without using the fact that in one run it can output several cuts of increasing balance and size. Recursive results are also never used again, instead the algorithm just runs in a loop with more and more costly recursions.

We attempted modifying FlowCutter to judge balance not based on sizes of the two sides, but on estimates of their treedepth (using callbacks). It appears this did not bring significant improvements to the final result, so this modification is turned off in the submitted version.

To speed things up a final feature is that of cutoffs. A bad cutoff d tells the algorithm to abandon any attempt that won't lead to a decomposition of depth strictly smaller than d. For example we use the best know decomposition's depth as a bad cutoff most of the time. A good cutoff d tells the algorithm to return as soon as it can output a decomposition of depth at most d. For example we use the maximum depth in sibling branches computed so far. The estimates obtained at early phases of the algorithm are also used to cut recursion attempts that are unlikely to yield better decompositions. We also use a lower bound based on the degeneracy of a graph and the longest path found in a DFS to quickly finish in some easy cases.

Acknowledgements

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

References

[Arn85] S. Arnborg. "Efficient algorithms for combinatorial problems on graphs with bounded decomposability – a survey". *BIT Numerical Mathematics* 25.1 (1985), 2–23.

[Heu+17] J. van den Heuvel, P. O. de Mendez, D. Quiroz, R. Rabinovich, and S. Siebertz. "On the generalised colouring numbers of graphs that exclude a fixed minor". *Eur. J. Comb.* 66 (2017), pp. 129–144.

[HS18] M. Hamann and B. Strasser. "Graph Bisection with Pareto Optimization". *ACM Journal of Experimental Algorithmics* 23 (2018).

[NM12] J. Nešetřil and P. O. de Mendez. "Bounded Height Trees and Tree-Depth". *Sparsity*. Springer, 2012, pp. 115–144.

[Str17] B. Strasser. "Computing Tree Decompositions with FlowCutter: PACE 2017 Submission". *CoRR* abs/1709.08949 (2017). arXiv: 1709.08949.

[TL84] R. E. Tarjan and J. van Leeuwen. "Worst-case Analysis of Set Union Algorithms". *J. ACM* 31.2 (1984), pp. 245–281.