PACE Solver Description: Fluid

Max Bannach
Institute for Theoretical Computer Science, Universität zu Lübeck, Germany
bannach@tcs.uni-luebeck.de

Sebastian Berndt
Institute for IT Security, Universität zu Lübeck, Germany
s.berndt@uni-luebeck.de

Martin Schuster
Institute for Epidemiology, Kiel University, Germany
martin.schuster@epi.uni-kiel.de

Marcel Wienöbst
Institute for Theoretical Computer Science, Universität zu Lübeck, Germany
wienoebst@tcs.uni-luebeck.de

Abstract

This document describes the heuristic for computing treedepth decompositions of undirected graphs used by our solve fluid. The heuristic runs four different strategies to find a solution and finally outputs the best solution obtained by any of them. Two strategies are score-based and iteratively remove the vertex with the best score. The other two strategies iteratively search for vertex separators and remove them. We also present implementation strategies and data structures that significantly improve the run time complexity and might be interesting on their own.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases treedepth, heuristics

Digital Object Identifier 10.4230/LIPIcs.IPEC.2020.27

Supplementary Material

Repository	`github.com/maxbannach/Fluid`
Release	pace-2020
doi	`10.5281/zenodo.3871709`

1 Introduction

A treedepth decomposition T of an connected, undirected graph $G = (V, E)$ is a rooted tree such that G is a subgraph of the closure of T. Such a decomposition can be obtained iteratively by taking a vertex $v \in V$ as root of T. Its children are then the decompositions of the connected components of $G[V \setminus \{v\}]$. Our heuristic iteratively removes vertices or sets of vertices to obtain a treedepth decomposition in this top-down fashion. Different strategies for choosing these vertices are used and the best solution over all these strategies is presented as output.

2 Score-Based Strategies

Our first two strategies are based on score function on the vertices, i. e., we iteratively choose a vertex with the best score, remove it from G, insert it in T, and update the scores of the other vertices. We use the following two score functions:

Degree-Score: $\text{score}_{\text{d}}(v) = |N(v)|$;
Fill-In-Score: $\text{score}_{\text{fi}}(v) = |\{\, \{x, y\} \mid x, y \in N(v) \wedge x \neq y \wedge \{x, y\} \notin E \,\}|$.

A naive way of implementing a score-based strategy is to recursively take the best vertex, remove it from the graph, and recompute the connected components. However, in this way, computing the connected components alone would require time $O(|V| \cdot |E|)$. For larger graphs, such a non-linear running time is not acceptable.

Instead of modifying the graph, we use the score functions to compute an *elimination ordering* $\pi = (v_1, v_2, \ldots, v_n)$ of G, that is, a permutation of the vertices such that v_i has the highest score in $G[V \setminus \{v_1, \ldots, v_{i-1}\}]$. Updating the scores is comparatively simple – for instance, computing an elimination ordering for score_{d} can be done in time $O(|E|)$.

Given an elimination ordering, we now face the new problem of obtaining a treedepth decomposition from it. We could, of course, compute the tree vertex-by-vertex by iterating over the vertices in the order of π. But then, we would have to recompute the connected components again – yielding a run time of $O(|V| \cdot |E|)$. To overcome this issue, we use the algorithm presented in Listing 1, which avoids the recursive recomputation of connected components by processing π in reversed order using a union-find data structure [1].

Listing 1 An efficient algorithm that computes a treedepth decomposition from a given elimination ordering in time $O(|E| \log^* |E|)$. The algorithm builds the tree in reversed order and maintains a union-find data structure in order to find the roots of subtrees efficiently.

```
1  INPUT:  graph G and elimination ordering π = (v_1, v_2, ..., v_n)
2  OUTPUT: elimination tree T
3  T ← ∅
4  uf ← ∅ // union-find structure with root pointer for each set
5  for v ← v_n, v_{n-1}, ..., v_1 do
6    Insert v as new singleton subtree in T.
7    Insert {v} as new set in uf. // set root pointer to v
8    for each w with (v, w) ∈ G and w ∈ T do
9      if v and w are not in the same subtree in T then
10       Let r be the root of the subtree in T containing w.
11       Insert an edge from r to v.
12       Join v and w in uf; update root pointer.
13 return T.
```

The advantage of implementing the score-based algorithm in two phases is that we can check many elimination orderings efficiently. We utilise this idea by repeatedly adding random perturbations to the score functions and running the algorithm multiple times. This leads to a large collection of treedepth decompositions, from which we output the best one.

3 Separator-Based Strategies

Instead of removing one vertex at a time, we may also remove a whole vertex separator at once and then recur into the new connected components immediately. Our two separator-based strategies iteratively search for such vertex separators, remove them from the graph and, then, proceed on the connected components separately.

3.1 Searching Separators Greedily

The first strategy finds the separator using a greedy algorithm. We maintain three sets A, B, and C such that no vertex in A is connected to any vertex in B. Initially, an arbitrary vertex v is chosen and A is initialized as $\{v\}$. The neighbors of A are inserted into C, all other vertices of V go to B. Now, we iteratively choose the vertex $v \in C$ that has the least number of edges to B, move v to A and put the neighbors of v that are still in B into C.

The algorithm will return a set C of minimal size that separates the graph into A and B within some balanced range, e. g., both at least $1/4$ of $|V|$. We observed the following extension to be helpful: When the subgraph induced by B contains a small connected component, we move the entire component with all its neighbors to A. This operation decreases the size of C while not changing the ratio of $|A|$ and $|B|$ too much.

Finally, we swap the role of A and B whenever $|B|$ is decreased to $1/4$ of $|V|$. In this way, we let B grow and A shrink and are often able to find smaller separators or separators with a better balance.

3.2 Search for Separators using Community Detection

Our second strategy runs the *asynchronous fluid communities algorithm* for community detection [3]. The algorithm computes a partition of V into two sets A and B. These sets are not necessarily of the same size, but are likely to be communities, i. e., it should be easy to separate A from B, but not so easy to separate the graph within A or within B.

In order to find a separator between A and B that we can use for our treedepth decomposition, we construct an auxiliary bipartite graph with one shore being A and the other being B. The edges of this bipartite graph are just the edges of the input graph G with one endpoint in A and one in B. We compute a maximum matching M in the bipartite graph and, using the König-Egervary Theorem [2], transform M into a minimum vertex cover S of the bipartite graph. This set S is then the sought separator in the original graph G.

We remark that, even though our algorithm has its name from the fluid community detection, it turned out that score-based heuristics or the greedy separator strategy is often superior compared to the fluid algorithm – both in quality and speed. However, there were a few instances in the test set on which this strategy was notably better than the others.

References

1 Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

2 Denes König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. *Mathematische Annalen*, 77(4):453–465, 1916. doi:10.1007/BF01456961.

3 Ferran Parés, Dario Garcia Gasulla, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, and Toyotaro Suzumura. Fluid communities: A competitive, scalable and diverse community detection algorithm. In Chantal Cherifi, Hocine Cherifi, Márton Karsai, and Mirco Musolesi, editors, *Complex Networks & Their Applications VI*, pages 229–240, Cham, 2018. Springer International Publishing.