PACE Solver Description: Finding Elimination Trees Using ExTREEm - a Heuristic Solver for the Treedepth Decomposition Problem

Sylwester Swat
Institute of Computing Science, Poznan University of Technology, Poland
sylwester.swat@put.poznan.pl

Abstract

This article briefly describes the most important algorithms and techniques used in the treedepth decomposition heuristic solver called "ExTREEm", submitted to the 5th Parameterized Algorithms and Computational Experiments Challenge (PACE 2020) co-organized with the 15th International Symposium on Parameterized and Exact Computation (IPEC 2020).

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases Treedepth decomposition, elimination tree, separator, PACE 2020

Digital Object Identifier 10.4230/LIPIcs.IPEC.2020.33

Supplementary Material The source code of ExTREEm solver is available at `https://doi.org/10.5281/zenodo.3873126`.

1 Problem description

A treedepth decomposition of a connected graph $G = (V, E)$ is a rooted tree $T = (V, E_T)$ such that every edge of G connects a pair of nodes that have an ancestor-descendant relationship in T. The solver briefly described here is a heuristic approach to the treedepth decomposition problem, where the goal is to find a treedepth decomposition for a given graph with as small height as possible.

2 Solver description

In this paper, we provide a short description of the most important algorithms implemented in solver ExTREEm. Due to many parameters used in the implementation and a lot of edge-cases that need to be taken into account, this description may not contain full information about the algorithms behavior in every possible situation.

Before we proceed to the actual description, let us fix some natural notations. For a given graph G we denote by $T(G)$ its treedepth decomposition. For a subset $S \subseteq V$ we denote by $C(G, S)$ the set of connected components of $G \setminus S$. For $a \in V$ we define $N(a) = \{v \in V : \{a, v\} \in E\}$ and for $A \subseteq V$ we take $N(A) = \bigcup_{v \in A} N(v)$.

Given a graph G, we find a separator S of G, then recursively obtain treedepth decompositions for components in $C(G, S)$, and finally merge separator S and found decompositions into an elimination tree of G. At the end, we apply some additional improvements to $T(G)$.

3 Separator evaluation

To assess the quality of a separator S we need to store values $mn(G, S)$ and $me(G, S)$ denoting, respectively, the maximum number of nodes and the maximum number of edges of a graph from $C(G, S)$. Now let us define

© Sylwester Swat;
licensed under Creative Commons License CC-BY

$$score_n(S) = |S| \cdot \frac{1-\beta^{-\frac{\log|V|}{\log\beta}}}{1-\beta}, \quad \text{where } \beta = \frac{mn(G,S)}{|V|}$$
$$score_e(S) = |S| \cdot \frac{1-\gamma^{-\frac{\log|E|}{\log\gamma}}}{1-\gamma}, \quad \text{where } \gamma = \frac{me(G,S)}{|E|}$$
$$score(S) = \alpha \cdot score_n(S) + (1-\alpha)\cdot score_e(S), \quad \text{where } \alpha \in [0,1] \text{ is a parameter.}$$

A separator S is balanced if $mn(G,S) < b \cdot |V|$ or $me(G,S) < b \cdot |E|$, where $b \in (0,1)$ is a parameter. For given two balanced or two unbalanced separators S_1 and S_2, we say that S_1 is better than S_2 if $score(S_1) < score(S_2)$. A balanced separator is always better than an unbalanced one.

4 Preprocessing

The preprocessing phase consists of two steps. The first step consists in detecting some induced subgraphs of a given graph G that are isomorphic to a cactus graph. We do that by repeatedly choosing a node of degree at most 2, removing it from the graph and adding edges connecting its neighbors (unless already present). Considering the initial graph G induced by the set of removed nodes, we observe that all its connected components are cacti. For each such cactus component we find a treedepth decomposition using recursively the Articulation Point Separator Creator method (see 5.1). Roots of those decompositions are attached to a proper node in a decomposition of the remaining graph.

The second step of the preprocessing is based on finding a maximum independent set in a subgraph of G induced by a set containing all nodes that are "center nodes" of induced claw-subgraphs as well as all nodes of degree four whose neighborhood induces a connected graph in G. Each node from the found independent set is removed from G and all pairs of its neighbors are connected by an edge. After finding a decomposition of the obtained graph, nodes from the found independent set are attached to the deepest of their neighbors.

5 Separator creation

After preprocessing, we try to find a good separator. After generating candidates, we select five best ones (with respect to their scores) that are further subjected to a refinement process, called minimization. As a final separator we take the best one after the minimization.

5.1 Articulation Point Separator Creator

In this method, we find all articulation points (cut vertices) of a given graph. Then, for each articulation point v, we find values $mn(G,\{v\})$ and $me(G,\{v\})$. This is done in $O(|E|)$ time using an algorithm similar to Tarjan's algorithm for finding bridges and articulation points.

5.2 BFS Separator Creator

Given a set $B \subseteq V$, we run a standard breadth-first-search with source-nodes set B. By L_i we denote the i-th BFS layer, that is a set containing all vertices that are at distance i from the set B. For each of those layers we consider a graph $G_i = G[V \setminus (L_0 \cup \ldots \cup L_{i-1})]$. Then, we divide nodes in L_i into blocks, two nodes belong to the same block if they belong to the same connected component of G_i. For each such block X we find a minimum vertex cover of a bipartite graph induced by the edges between sets X and $(N(X) \cap L_{i+1})$. All blocks and vertex covers are treated as different separator candidates and are found in time $O(|E||V|^{\frac{1}{2}})$.

5.3 Component Expansion Separator Creator

We take some subset $B \subseteq V$ and then iteratively expand B by adding to it one node from $N(B)\backslash B$. In the first variant, we select a node with the tightest connection to B. In the second one, we select a node from B that has the least number of neighbors outside B, then add these neighbors to B in an arbitrary order. We obtain a sequence $(v_1, \dots v_n)$ of added nodes called an *expansion order*. We now consider separators $S_i = \{v_j : j \leq i, N(v_j) \cap (V \setminus \{v_1, \dots, v_i\}) \neq \emptyset\}$ and try to improve the given expansion order by taking smaller connected components of the partitioned graph before the larger ones. If we have already constructed separator S_i and we add v_{i+1} to that separator (and probably do some more changes) to obtain S_{i+1}, we afterwards rearrange all remaining nodes $v_{i+2}, \dots, v_n$ in such a way that nodes belonging to smaller connected components of $G[\{v_{i+2}, \dots, v_n\}]$ occur before all nodes that belong to larger connected components. All separators S_i are found in time $O(|E| \log |V|)$.

5.4 Flow Separator Creator

At the beginning, we select two sets of nodes that are possibly far from each other. This is done by creating a “landmark set”, that is a set L obtained by first selecting a random node and then iteratively adding to L any node that is farthest from L. Then, we select two random nodes u and v from L and consider sets of the form $B = N(N(N(u)))$ and $E = N(N(N(v)))$. Afterwards, we find a maximal set of node-disjoint paths that begin in B and end in E. As a separator candidate we take the union of those paths, then minimize it with Greedy Minimizer. This method of creating separators works best in the context of Flow Minimizer.

5.5 FlowCutter Separator Creator

In this method, we use our own implementation of a slightly modified version of the FlowCutter algorithm (see [1]). The main idea remains the same - to expand the set of sources or targets and to avoid augmenting paths. As initial source nodes and target nodes we consider, as in 5.4, pairs of nodes from the “landmark set”. Additionally, we admit certain imbalance in the source-reachable and target-reachable node sets only after the size of sources and targets reaches some fixed fraction of $|V|$. After a last node is marked as a source or a target, we consider four different expansion orders based on the order of adding graph nodes to sources and targets, and for each order we find separators using the Component Expansion Separator Creator method (5.3). We also consider as a separator a vertex cover of a graph induced by edges between the final sets of sources and targets.

6 Separator minimization

After creating separator candidates, we proceed to the refinement step - for each candidate S we exhaustively try to minimize the value of $score(S)$ using following methods:

1. Vertex Cover Minimizer – we find a vertex cover of a bipartite graph induced by edges between sets S and $N(S) \cap C_d$, where C_d is some subset of $C(G, S)$.
2. BFS Minimizer and Component Expansion Minimizer – we find separators using the methods from 5.2 and 5.3, respectively, with the initial source-set S.
3. Greedy Minimizer – we iteratively remove nodes from S, each time selecting a node which removal results in the minimal total size of the connected components adjacent to that node.

4. Flow Minimizer and FlowCutter Minimizer – we find separators using the methods from 5.4 and 5.5, respectively, with the initial sets of sources and targets set to some subsets of nodes that lie at a fixed distance from S.

7 Subtrees merging

After recursively finding decompositions for all components in $C(G, S) = \{c_1, \dots, c_k\}$, we need to merge the results to obtain $T(G)$. To do that, we sort all components according to the nonincreasing depths of their decompositions. Then, we create a sequence S' by iteratively adding to S' nodes from $S \cap N(C_i)$ that were not added to S' earlier. As the tree $T(G)$ we initially take the path represented by sequence S', then we attach each tree $T(G[C_i])$ to the last node from sequence S' that occurs in $N(C_i)$. The whole procedure takes time $O(|E| \log |V|)$.

8 Tree improvements

When the tree $T(G)$ is created, we try to improve it by performing some structure-based changes. Those improvements are based on pivot-like operations.

8.1 Block pivots

By the block of $v \in V$ in a tree $T(G)$ we mean a maximal path in $T(G)$ which contains v, such that each node on that path, apart from the deepest one, has at most one son. We construct a separator S of G by taking all nodes from the root-block of T and recursively doing the same for the highest tree in the decomposition of $G \setminus S$, until $|S| > d \cdot H$, where $d \in [0, 1]$ is a parameter and H is the height of $T(G)$. Then, we merge separator S and all trees just as described in Section 7.

8.2 Hall-set pivots

Let us fix any block in $T(G)$ that lies on a longest leaf-root path P and let v be the topmost node in that block. Let $U(v)$ be the set of nodes on that path from the root to the parent of v and $D(v)$ be the remaining nodes on path P. By T_v we denote the subtree of T with root in v. We now consider a maximum matching M in a bipartite graph induced by edges between sets $U(v)$ and $N(U(v)) \cap R$, where R is either T_v or $T_v \setminus D(v)$.

If possible, we take a set $H_M \subseteq U(v)$ with the property $|H_M| > |N(H_M) \cap R|$ and check a treedepth decomposition obtained from T by removing nodes that belong to $N(H_M) \cap R$, setting those nodes as the root-block and performing some other necessary structural changes.

9 Availability

The source code of ExTREEm solver is available at `https://doi.org/10.5281/zenodo.3873126`.

References

1 Michael Hamann and Ben Strasser. Graph bisection with pareto-optimization. *CoRR*, abs/1504.03812, 2015. `arXiv:1504.03812`.