PACE Solver Description: Tree Depth with FlowCutter

Ben Strasser
Independent Researcher, Germany
acedemia@ben-strasser.net

Abstract

We describe the FlowCutter submission to the PACE 2020 heuristic tree-depth challenge. The task of the challenge consists of computing an elimination tree of small height for a given graph. At its core our submission uses a nested dissection approach, with FlowCutter as graph bisection algorithm.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases tree depth, graph algorithm, partitioning

Digital Object Identifier 10.4230/LIPIcs.IPEC.2020.32

Supplementary Material Our source code is available at `https://github.com/ben-strasser/flow-cutter-pace20` and `https://doi.org/10.5281/zenodo.3870928`.

Acknowledgements We thank the organizers for their work in making the 2020 PACE challenge a success.

1 Introduction

Using the original FlowCutter code base, we implemented a program to compute elimination trees of small height. We submitted it to the PACE 2020 heuristic tree-depth implementation challenge. In this paper, we describe our submission.

The objective of the challenge is to implement an algorithm that, given an undirected graph, computes an elimination tree of small height within a fixed amount of time. The height of the computed tree is used to score the competing implementations. Our source code is available at [13, 14]. The core of our program is very similar to the FlowCutter submissions to the PACE 2016 and 2017 heuristic tree decomposition [8, 12].

The FlowCutter algorithm was introduced in [6, 7]. Our submission is based on the original code. Two other independent PACE 2020 competitors make use of reimplementations of the FlowCutter algorithm. These are ExTREEm [16] and Sallow [17]. ExTREEm won the first place and Sallow the third. We won the second place. The FlowCutter algorithm is thus used in all top three submissions.

FlowCutter is a balanced graph bisection algorithm. The base version computes balanced edge cuts. There exists an extension that computes node separators. For a detailed description of FlowCutter, we refer to [6, 7]. The idea of FlowCutter is to compute a maximum flow and then to derive a minimum cut from it. If the derived cut is balanced, we are finished. Otherwise, additional source or target nodes are added and the flow intensity is increased. The minimum cut derived from the new flow has a better balance due to the additional source and target nodes. This is repeated until the desired balance is reached.

FlowCutter does not only compute a single cut. It computes a sequence of cuts that optimize balance and cut size in the Pareto-sense. Using this sequence, solutions to more complex problems can be found. For example, the cut expansion can be maximized. The expansion of a cut is its size divided by the number nodes on the smaller side.

FlowCutter has been developed to analyze road networks with the goal of accelerating shortest path queries. FlowCutter is used to compute an elimination tree, which is used as input to an algorithm called Customizable Contraction Hierarchy (CCH) [1, 2, 15], which can quickly compute shortest paths. In the CCH context, elimination orders are called contraction orders.

2 Pace 2020 Submission

The core of our submission is a combination of nested dissection [4] and FlowCutter. We use it to compute an elimination order, from which we derive an elimination tree. Nested dissection is a recursive scheme to compute an elimination order. First, a node separator is computed. This separator splits the graph into parts. Next, elimination orders are recursively computed for every part. The final elimination order is the concatenation of the parts' elimination orders followed by the nodes of the separator. The base case for our recursion are clique and tree graphs. For cliques, any order is optimal and thus the problem is trivial. For tree graphs, we use the optimal algorithm of [11]. If the graph is not a tree nor a clique, we compute a separator with a large expansion and a bounded imbalance using FlowCutter.

FlowCutter usually finds good separators regardless of the graph structure. However, it has a running time proportional to the product of the number of edges and the size of the separator. If there are small separators that FlowCutter can find, then FlowCutter has a nearly linear running time. Unfortunately, if the separator sizes found by FlowCutter are linear in the input graph size, then FlowCutter has quadratic running time. As a result, the time limit is reached before FlowCutter finishes.

To mitigate these problems, we implemented two additional approaches. One is based on a scheme inspired by the minimum degree heuristic [9, 5] and the Contraction Hierarchy node ordering heuristic [3]. The details are described in Section 3.

The other is very loosely based on label propagation [10], which tends to work well on graphs with large separators. It computes edge cuts. From these, we derive nodes separators by picking the nodes incident to the cut on one side. Together with nested dissection, an elimination order can be computed with this scheme. The edge cut algorithm is described in Section 4. We only use it, when we suspect that the graph has large separators.

In multiple steps, we try finding solutions using various approaches. First, we run the minimum degree heuristic variant to assure that we find some solution. If the achieved depth is above a threshold, we run the label propagation approach. Next, we run the edge cut variant of FlowCutter from which we derive node separators and apply nested dissection. The rational is that the edge cut FlowCutter variant is faster than the node variant.

Finally, we run the node separator variant of FlowCutter combined with nested dissection. This step is repeated until the timeout is reached. In every step, the random seed and tuning parameters are changed. For details about the tuning parameters, we refer to the code. We abort a nested dissection if the resulting tree would be higher than the best known tree.

3 Minimum Degree Heuristic

Most minimum degree heuristics use a node rating function $r(x)$. All nodes placed into a minimum priority queue ordered by $r(x)$. Nodes are then iteratively removed from the queue. After removing a node x, it is eliminated from the graph, i.e., edges among x's neighbors are inserted. Afterwards, the rating value $r(y)$ of all neighbors y of x must be recomputed. We update the positions of the neighbors y in the queue. This is repeated until the queue is empty. The order in which the nodes are removed from the queue is the computed elimination order. Usually, the rating function estimates something related to the height of a node.

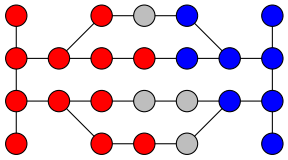

Figure 1 Example of Local Optimum. Initially, red nodes are left and blue and gray nodes are right. Grey nodes are moved by move-to-left round.

Our rating function is $r(x) = d(x) + 8 \cdot \ell(x)$. It has two terms $d(x)$ and $\ell(x)$. $d(x)$ is the degree of x in the graph after eliminating all nodes removed from the queue. $\ell(x)$ is an estimation of the height of x that is induced by the eliminated nodes. Initially, $\ell(x)$ is zero. When a node x is eliminated, we set $\ell(y)$ to $\max\{\ell(y), \ell(x) + 1\}$ for all neighbors y of x.

Inserting edges between all neighbors of a node x is slow if x has a high degree. We therefore abort the queue-based algorithm if all remaining nodes have a degree ≥ 150. If this happens, we sort the remaining nodes x by $r(x)$ and place them into the elimination order.

4 Label Propagation

We implement a graph bisection algorithm that is loosely based on label propagation. First, we describe the high-level label propagation scheme. Next, we explain the details of our algorithm. Finally, we discuss why this setup works on certain graphs.

Label propagation algorithms start with a simple and fast method to find an initial cut. This initial cut can be very large. It is then refined in many rounds. In every round, we iterate over all nodes x incident to the cut in random order. Using local information, we determine whether x should change sides. Rounds are executed until almost convergence is reached, i.e., the number of nodes changing side is small.

We refer to the sides as *left* and *right*. The *imbalance* of a cut is the size of the larger side divided by the node count. We aim to find a cut with few edges and at most an imbalance of 2/3. First, we pick two random seed nodes. From these, two simultaneous breath first searches are run. A node x is on the side whose search first reaches x.

We use three types of rounds named *decrease-cut-size*, *decrease-imbalance*, and *move-to-left*. We repeat all rounds until near convergence. The rounds only differ with respect to when a node moves between sides. In a decrease-cut-size round, a node moves if moving it decreases the cut size and the imbalance is at most 2/3. The objective is to only perform moves that immediately improve the cut size. In a decrease-imbalance round, a node additionally moves, if moving decreases the imbalance without increases the cut size. Finally, in a move-to-left round, a node moves to the left side, if the cut size remains the same and the imbalance is at most 2/3. In this round, no node moves to the right. We start by performing decrease-cut-size rounds, then do decrease-imbalance rounds, and finally do move-to-left rounds. This is repeated for a fixed number times. We finish with a decrease-imbalance round.

The decrease-cut-size and improve-balance rounds try to greedily improve our optimization criteria. However, they can run into local optima. One such situation is depicted by Figure 1. Initially, the red nodes form the left side and the gray and blue nodes the right side. The cut goes through path subgraphs. No single move improves balance or decreases cut size. However, a lot of nodes exist that can be moved without causing harm. Moving them randomly will move nodes aimlessly between left and right. If the paths are long enough, no progress is made with sufficient probability. To make progress, we need to tie-break these moves. We do this by moving as many nodes as possible to the left while maintaining

a 2/3-balance. This has a high chance of moving the grey nodes. In the next decrease-cut-size rounds, the cut size shrinks. The next improve-balance rounds find the symmetric structure on the top of the figure example. This concludes the description of our label propagation graph bisection algorithm.

5 Conclusion

We described our FlowCutter submission to the PACE 2020 tree-depth challenge. It is based on the original FlowCutter code. Two other independent competitors make use of reimplementations of the FlowCutter algorithm. These are Sallow and ExTREEm. Together these three submissions managed to win first, second, and third place. This shows that FlowCutter is a powerful tool to compute eliminations trees.

References

1 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In *Symposium Experimental Algorithms SEA 2014*, volume 8504, pages 271–282. Springer, 2014. doi:10.1007/978-3-319-07959-2_23.

2 Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *ACM Journal of Experimental Algorithmics JEA*, 21(1):1.5:1–1.5:49, 2016. doi:10.1145/2886843.

3 Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Workshop on Experimental Algorithms, WEA 2008*, volume 5038, pages 319–333. Springer, 2008. doi:10.1007/978-3-540-68552-4_24.

4 Alan George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

5 Alan George and Joseph WH Liu. The evolution of the minimum degree ordering algorithm. *Siam review*, 31(1):1–19, 1989.

6 Michael Hamann and Ben Strasser. Graph bisection with pareto-optimization. In Michael T. Goodrich and Michael Mitzenmacher, editors, *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 90–102. SIAM, 2016. doi:10.1137/1.9781611974317.8.

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

8 Michael Hamann and Ben Strasser. Correspondence between multilevel graph partitions and tree decompositions. *Algorithms*, 12(9):198, 2019. doi:10.3390/a12090198.

9 Harry M Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

10 Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

11 Alejandro A. Schäffer. Optimal node ranking of trees in linear time. *Inf. Process. Lett.*, 33(2):91–96, 1989. doi:10.1016/0020-0190(89)90161-0.

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

13 Ben Strasser. Flowcutter pace 2020 submission, 2020. URL: https://github.com/ben-strasser/flow-cutter-pace20.

14 Ben Strasser. Flowcutter pace 2020 submission, 2020. doi:10.5281/zenodo.3870928.

15 Ben Strasser and Dorothea Wagner. Graph fill-in, elimination ordering, nested dissection and contraction hierarchies. In *Gems of Combinatorial Optimization and Graph Algorithms*, pages 69–82. Springer, 2015. doi:10.1007/978-3-319-24971-1_7.

16 Sylwester Swat. Extreem source code. doi:10.5281/zenodo.3873126.

17 Marcin Wrochna. Sallow: a heuristic algorithm for treedepth decompositions. *CoRR*, abs/2006.07050, 2020. arXiv:2006.07050.