


Solver Description of PID[★]

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 provides a short overview of our treedepth solver PID[★] in the version that we submitted to the exact track of the PACE challenge 2020. The solver relies on the positive-instance driven dynamic programming (PID) paradigm that was discovered in the light of earlier iterations of the PACE in the context of treewidth. It was recently shown that PID can be used to solve a general class of vertex pursuit-evasion games – which include the game theoretic characterization of treedepth. Our solver PID[★] is build on top of this characterization.

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

Keywords and phrases treedepth, positive-instance driven

Supplementary Material

<i>Repository</i>	github.com/maxbannach/PID-Star
<i>Release</i>	pace-2020
<i>doi</i>	DOI 10.5281/zenodo.3871800

1 Introduction

We say a dynamic program runs in *positive-instance driven* mode (PID) if it only enumerates positive subproblems. Running an algorithm in this mode has two effects: First, the algorithm will enumerate *less* subproblems and, second, the algorithm will invest *more* time to compute new subproblems. Hisao Tamaki was the first to apply this mode to algorithms that compute tree decompositions [7]. He observed experimentally that, for treewidth, the advantage of the first effect compensates for the disadvantage of the second effect [8] – making PID based algorithms the current state-of-the-art for finding optimal tree decompositions [4].

Treewidth, as well as treedepth and many other graph invariants, have elegant game theoretic characterizations in the form of vertex pursuit-evasion games [3]. In this light, an algorithm computing a graph invariant can be seen as a procedure that computes a winning strategy in the corresponding game; and an algorithm running in PID mode as one that computes exactly the *winning region* in that game – without visiting other configurations.

2 Core Routine of the Solver

With this game theoretic characterization in mind, we recently presented a general PID algorithm that works for a variety of graph parameters, including treedepth [1]. Our

submission is based on this algorithm, but differs in three regards in order to make it fast in practice: First, we only consider connected positive subproblems. For treedepth, as for treewidth, this is sufficient if the input graph is connected – and there can be exponentially less connected positive subproblems than unconnected ones [8].

Secondly, the original algorithm works in two phases: It first computes the set of positive instances and, then, solves distance queries on this set [1]. By replacing the reverse breadth-first search by a reverse version of Dijkstra’s algorithm we are able to compute these distances on-the-fly. This means the algorithm maintains at any point in time a locally optimal treedepth decomposition for every positive instance (and not just one that can be extended to a global optimal one). If this reverse search has multiple candidates of equal quality, we use a heuristic to determine which candidate is most likely to be good for an optimal global solution and pick this one first, leading to some sort of reverse A* – therefore the name.

Finally, we added a potpourri of preprocessing and pruning rules to the algorithm. This is important in order to make PID based algorithms fast in practice, as such algorithms perform worse and worse with increasing k (the larger k , the more positive subproblems). We solve an instance by increasing a lower bound until we reach the first positive instance, i. e., we set $k = 1, 2, \dots, \text{opt}$. In contrast to other algorithmic paradigms like branch-and-bound, where often $\text{opt} - 1$ is the toughest instance, PID based algorithms struggle the most with $k = \text{opt}$ [8]. We run a heuristic beforehand and hope to get this instance out of the way [2].

3 Preprocessing

As mentioned before, the more positive subproblems we have, the worse the performance of PID*. This insight gives the algorithm the strange property that sometimes “easy instances are hard,” because “easy” instances often have many positive subproblems. In order to avoid this phenomenon, we remove “easy parts” of the input with the following preprocessing rules:

- **Rule 1** (Leaf Rule [5]). *Let $v, w, w' \in V$ with $w, w' \in N(v)$ and $|N(w)| = |N(w')| = 1$, then delete w' .*
- **Rule 2** (Improvement Rule [6]). *Let $u, v \in V$ with $\{u, v\} \notin E$ and $|N(u) \cap N(v)| \geq k$, then add the edge $\{u, v\}$.*
- **Rule 3** (Simplicial Rule [6]). *Let $u \in V$ be simplicial such that $|N(v)| > k$ for all $v \in N(u)$, then delete u .*

We remark that in addition to Rule 1, one can also remove apexes. However, there were none in the provided test set and we expect them to be rare in general, so we did not implement this rule. Rule 2 can also be generalized in the sense that we can add edges between vertices that are connected by at least k vertex-disjoint paths. However, this adds a large computational overhead and did not provide any advantage in our experiments. It is also worth mentioning that the first two rules provide a nice little speed-up in general, while Rule 3 has no effect on most instances, but is crucial for solving some dense instances.

Finally, we would like to remark that in contrast to treewidth, we have no good preprocessing rules for treedepth that deal with degree-2 vertices. These vertices, especially in longer chains, seem to be very harmful for the performance of PID*. We therefore expect a notable speed-up through a preprocessing rule that deals with degree-2 vertices and suggest further research in this direction.

References

- 1 Max Bannach and Sebastian Berndt. Positive-instance driven dynamic programming for graph searching. In *WADS*, volume 11646 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 2019.
- 2 Max Bannach, Sebastian Berndt, Martin Schuster, and Marcel Wienöbst. Fluid. <http://www.github.com/maxbannach/Fluid>. Accessed: 12.06.2020; Commit: 75e1176. doi:10.5281/zenodo.3871709.
- 3 Daniel Bienstock. Graph searching, path-width, tree-width and related problems (A survey). In *Reliability Of Computer And Communication Networks*, volume 5 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 33–50. DIMACS/AMS, 1989.
- 4 Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In *IPEC*, volume 89 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- 5 Robert Ganian, Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. SAT-Encodings for Treecut Width and Treedepth. In *Proceedings of the 21th Workshop on Algorithm Engineering and Experiments, ALENEX 2019, San Diego, CA, USA, January 7-8, 2019.*, pages 117–129, 2019. doi:10.1137/1.9781611975499.10.
- 6 Yasuaki Kobayashi and Hisao Tamaki. Treedepth Parameterized by Vertex Cover Number. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation, IPEC 2016, August 24-26, 2016, Aarhus, Denmark*, pages 18:1–18:11, 2016. doi:10.4230/LIPIcs.IPEC.2016.18.
- 7 Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *J. Comb. Optim.*, 37(4):1283–1311, 2019.
- 8 Hisao Tamaki. Experimental analysis of treewidth. In *Treewidth, Kernels, and Algorithms*, volume 12160 of *Lecture Notes in Computer Science*, pages 214–221. Springer, 2020.