PACE Solver Description: 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

Digital Object Identifier 10.4230/LIPIcs.IPEC.2020.28

Supplementary Material
Repository `github.com/maxbannach/PID-Star`
Release pace-2020
doi `10.5281/zenodo.3871800`

1 Introduction to Positive-Instance Driven Dynamic Programming

Many graph decompositions have game theoretic characterizations in the form of *vertex pursuit-evasion games.* Such games, which are also known as *graph searching* or *cops and robber,* are played by two players on an undirected graph $G = (V, E)$. In the version of the game that corresponds to treedepth, the first player places a team of k *searchers* on the vertices of the graph, while the second player controls a single *fugitive* that hides in a connected component of the graph. The game is played in rounds as follows [3]: Initially, the fugitive picks one connected component C of G. The game is continued only on $G[C]$ and we say that C is *contaminated.* In each round, both players perform one action:

1. The searchers pick a vertex $v \in C$ on which they want to place the next searcher. We say they *clean* the vertex v.
2. The fugitive responds by picking a component C' of $G[C \setminus \{v\}]$. The contaminated area is reduced to C' and the game proceeds only on this subgraph.

The game ends when the contaminated area shrinks to the empty set, or if the searchers have placed all k members of their team and C is still non-empty. In the first case the graph was *cleaned* and the fugitive was *caught,* in the second case the fugitive *escaped.* The searchers

15th International Symposium on Parameterized and Exact Computation (IPEC 2020).
Editors: Yixin Cao and Marcin Pilipczuk; Article No. 28; pp. 28:1–28:4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

win if they catch the fugitive, otherwise she wins. Note that in this version of the game, the searchers are not allowed to remove an already placed searcher from the graph. The game is therefore monotone and always ends after at most k rounds. Further observe that the fugitive is *visible* in the sense that the searchers know in which connected component she hides – in contrast, an invisible fugitive could hide in subgraphs that are not connected.

We call the configurations of this game *blocks,* which are tuple (C, ρ) with $\rho \in \mathbb{N}$ and $C \subseteq V$ being a connected subgraph with $|N(C)| + \rho \leq k$. Informally, C is the contaminated area (which is connected), and ρ is the number of remaining searchers. We require $|N(C)| + \rho \leq k$ as the neighborhood of C has to be cleaned in order to have C as contaminated area. Let us denote the set of all blocks of the game played on a graph G with a team of k searchers by $\mathcal{B}(G, k)$. Two blocks (C_1, ρ_1) and (C_2, ρ_2) *intersect* if $N[C_1] \cap C_2 \neq \emptyset$. The *start configuration* of the game is the block (V, k) and the *winning configurations* for the searchers are $(\emptyset, \rho \geq 0)$. We say the searchers have a *winning strategy* on a block (C, ρ) if they can ensure to reach a winning configuration no matter how the fugitive acts. The set of such blocks is the *winning region* of the searchers, which we denote by $\mathcal{R}(G, k) \subseteq \mathcal{B}(G, k)$. Every block in $\mathcal{R}(G, k)$ is called *positive.*

It is known that a graph has treedepth at most k if, and only if, k searchers have a winning strategy in the game defined above. In our notation we can express this fact as:

▶ **Fact 1** ([3]). *Let $G = (V, E)$ be a graph and $k \in \mathbb{N}$. Then $(V, k) \in \mathcal{R}(G, k) \iff \mathrm{td}(G) \leq k$.*

Fact 1 tells us that, in order to check whether the treedepth of a graph G is at most k, it is sufficient to compute the set $\mathcal{R}(G, k)$. One way of doing so would be to first compute $\mathcal{B}(G, k)$, then build an auxiliary graph on top of this set, and finally compute $\mathcal{R}(G, k)$ by solving reachability queries on this auxiliary graph. We can estimate the number of configurations with $|\mathcal{B}(G, k)| \leq (k + 1) \cdot n^{k+1}$, as there are n^k possible ways of placing k searchers on an n-vertex graph; at most n connected components adjacent to a separator; and since $\rho \in \{0, \dots, k\}$. Therefore, the sketched algorithm achieves a run time of $O\big(n^{c \cdot k}\big)$ for a constant c, which is not feasible in practice for even moderate values of k.

In order to make the game theoretic approach feasible, we present an *output-sensitive* algorithm that computes just $\mathcal{R}(G, k)$ – without "touching" the rest of $\mathcal{B}(G, k)$. Such an algorithm is called *positive-instance driven.* This algorithmic technique was invented by Hisao Tamaki in the context of treewidth computations [5] and was recently shown to be able to solve a general class of graph searching games [1] – PID* is based on this version.

2 Description of the Core Algorithm

Before we describe the algorithm formally, let us build some intuition about how to compute the set $\mathcal{R}(G, k)$. Surely, we can not start at some block, say (V, k), and just simulate the game – we might touch a lot of blocks in $\mathcal{B}(G, k) \setminus \mathcal{R}(G, k)$ without even noticing it. After all, we do not know whether $(V, k) \in \mathcal{R}(G, k)$. We do know, however, that $(\emptyset, 0)$ is a winning configuration. So let us start with the set $\mathcal{R} = \big\{ (\emptyset, 0) \big\}$ and then try to grow it to $\mathcal{R}(G, k)$. We can first ask which configurations of the game lead to $(\emptyset, 0)$, i. e., what are configurations in which the searchers immediately win in the next round? These are the configurations $(\{v\}, 1)$ with $|N(v)| < k$, as in these the searchers can surround the fugitive and have a searcher left to place it on top of her in the next round. Now assume that we currently have a set $\mathcal{R} \subseteq \mathcal{R}(G, k)$ that did already grow a little. How does a configuration $(C, \rho) \in \mathcal{R}(G, k) \setminus \mathcal{R}$ that is "close to" $\mathcal{R}$ look like? The set C is connected by definition, and since the searchers have a winning strategy from (C, ρ), there is a vertex $v \in C$ such that $G[C \setminus \{v\}]$ has connected components $C_1, \ldots, C_q$ ($q = 1$ is possible) with $(C_i, \rho - 1) \in \mathcal{R}$ for all $i \in \{1, \ldots, q\}$. To find these configurations, we scan through the blocks (C, ρ) in $\mathcal{R}$, guess a neighbor $v \in N(C)$ (the last cleaned vertex), and guess a set $X \subseteq \{\, (C', \rho') \in \mathcal{R} \mid v \in N(C') \wedge N[C] \cap C' = \emptyset \wedge \rho' \leq \rho \,\}$ of pairwise non-intersecting blocks – the other configurations the fugitive could choose. Then the new block $(C \cup \bigcup_{(C',\rho') \in X} C' \cup \{v\}, \rho + 1)$ is positive and added to $\mathcal{R}$ if it has at most $k - \rho - 1$ neighbors. The complete algorithm is presented in Listing 1.

Listing 1 The core positive-instance driven algorithm tailored towards treedepth. We assume that the set $\mathcal{R}'$, the priority queue, and some data structure to mark already explored subgraphs C (for instance a hash set) are available in global memory.

```
INPUT:  graph $G = (V, E)$ and number $k \in \mathbb{N}$
OUTPUT: a set $\mathcal{R} = \mathcal{R}(G, k)$

// in global memory
$\mathcal{R}'$ ← empty set of blocks
queue ←  priority queue of blocks $(C, \rho)$ ordered by $\rho$

function pid()
  // configurations leading to $(\emptyset, 0)$
  for $v$ in $V$ do
    if $|N(v)| < k$ then
      insert $(\{v\}, 1)$ into queue
    end
  end
  // compute the set $\mathcal{R}' \subseteq \mathcal{R}(G, k)$
  while queue is not empty do
    $(C, \rho)$ ← extract a block from the queue
    if $C$ was already visited then
      skip $(C, \rho)$ and continue the while-loop
    end
    mark $C$ as visited
    // compute predecessor configurations
    for $v$ in $N(C)$ do
      for $X \subseteq \{\, (C', \rho') \in \mathcal{R}' \mid v \in N(C') \wedge N[C] \cap C' = \emptyset \wedge \rho' \leq \rho \,\}$ do
        // assert: blocks in $X$ are pairwise non-intersecting
        if $|N(C \cup \bigcup_{(C',\rho') \in X} C' \cup \{v\})| \leq k - \rho - 1$ then
          insert $(C \cup \bigcup_{(C',\rho') \in X} C' \cup \{v\}, \rho + 1)$ into queue
        end
      end
    end
    $\mathcal{R}'$ ← $\mathcal{R}' \cup \{\, (C, \rho), \}$
  end
  // compute $\mathcal{R}$ from $\mathcal{R}'$
  $\mathcal{R}$ ← $\bigcup_{(C,\rho) \in \mathcal{R}'} \{\, (C, \rho') \mid \rho' \geq \rho \wedge |N(C)| + \rho' \leq k \,\}$
end
```

▶ **Theorem 2.** *Let $\mathcal{R}$ be the output of the algorithm in Listing 1 on input of a graph $G = (V, E)$ and a number $k \in \mathbb{N}$. Then $\mathcal{R} = \mathcal{R}(G, k)$.*

3 Preprocessing and Pruning Rules

To compute the treedepth of a graph $G = (V, E)$, we use the algorithm from the previous section for $k = 1, 2, \ldots, \text{opt}$, i. e., we increase a lower bound until we reach the first positive instance. To each such instance (G, k), we apply the following reduction rules in advance:

▶ **Rule 1** (Leaf Rule [2]). *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 [4]). *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** (Simplical Rule [4]). *Let $u \in V$ be simplical such that $|N(v)| > k$ for all $v \in N(u)$, then delete u.*

To increase the performance of the algorithm from Listing 1, we apply the following pruning rules. We say a winning strategy of the searchers has a *conflict* if there are two vertices $u, v \in V$ with $N(u) \setminus \{v\} \subsetneq N(v) \setminus \{u\}$ such that the searchers clean u before v.

▶ **Lemma 3.** *If k searchers have a winning strategy on a graph $G = (V, E)$, then they also have a conflict free winning strategy on G.*

We can adapt the rules of our game with the lemma, without losing Fact 1. The new game simply forbids that the searchers clean a vertex u as long as there is a contaminated vertex v with $N(u) \setminus \{v\} \subsetneq N(v) \setminus \{u\}$. We define the following sets for every vertex $v \in V$:

$$\begin{aligned} \text{descendants}(v) &= \{\, u \mid \{u, v\} \in E \wedge N[u] \subsetneq N[v] \,\}, \\ \text{non-ancestors}(v) &= \{\, u \mid \{u, v\} \notin E \wedge N(u) \subsetneq N(v) \,\}. \end{aligned}$$

Assume the algorithm generates a new block (C, ρ) by gluing previously discovered blocks $(C_1, \rho_1), \ldots, (C_q, \rho_q)$ at some vertex $x \in V$, i. e., $C = \{x\} \cup \bigcup_{i=1}^{q} C_i$ (see line 27 in Listing 1). We check whether we have $\text{descendants}(x) \subseteq C$ and $x \notin \bigcup_{y \in C \setminus \{x\}} \text{non-ancestors}(y)$. If this is not the case, we discard the block.

Our second pruning rule avoids the expensive glue operation in line 24. Let (C, ρ) be a block and $v \in N(C)$. We say v is *covered* if $N(v) \subseteq N[C]$ and we call v an *attachment* if $\text{td}(G[C]) = \text{td}(G[C \cup \{v\}])$ and $|N(C)| = |N(C \cup \{v\})|$. One can show that we can, in both cases, greedily add v to C and proceed with $(C \cup \{v\}, \rho + 1)$ without further handling (C, ρ).

References

1. Max Bannach and Sebastian Berndt. Positive-instance driven dynamic programming for graph searching. In *Proceedings of the 16th International Symposium on Algorithms and Data Structures*, 2019. doi:10.1007/978-3-030-24766-9_4.
2. 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*, 2019. doi:10.1137/1.9781611975499.10.
3. Archontia C. Giannopoulou, Paul Hunter, and Dimitrios M. Thilikos. Lifo-search: A min-max theorem and a searching game for cycle-rank and tree-depth. *Discret. Appl. Math.*, 160(15):2089–2097, 2012. doi:10.1016/j.dam.2012.03.015.
4. Yasuaki Kobayashi and Hisao Tamaki. Treedepth Parameterized by Vertex Cover Number. In *Proceedings of the 11th International Symposium on Parameterized and Exact Computation*, 2016. doi:10.4230/LIPIcs.IPEC.2016.18.
5. Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *J. Comb. Optim.*, 37(4):1283–1311, 2019.