PACE 2023: TinyWidth Solver

Gabriel Bathie
DI ENS, PSL Research University, Paris, France, and LaBRI, Université de Bordeaux, France

Jérôme Boillot
DI ENS, PSL Research University, Paris, France

Nicolas Bousquet
Univ. Lyon, Université Lyon 1, CNRS, LIRIS UMR 5205, France

Théo Pierron
Univ. Lyon, Université Lyon 1, CNRS, LIRIS UMR 5205, France

Abstract

The twin-width, introduced in 2020 by Bonnet et al., is a graph parameter that captures the tractability of first-order model checking on ordered graphs.

This paper presents the TinyWidth solver, a program that computes the twin-width of undirected graphs, geared towards graphs of small twin-width. This solver was submitted to the exact track of the 2023 PACE Challenge. Our solver uses a branch-and-bound approach, combined with caching of partial results to speed-up computation and a recursive call for lower bounds. Furthermore, we present pre-processing reduction rules that reduce isolated tree-like parts of the graph.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases twin-width, branch and bound

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Overview

TinyWidth solver uses a branch-and-bound search strategy, augmented with memoization of partial results. Branch-and-bound is an exhaustive search strategy that prunes branches of the search trees that cannot improve on the current best results, for example when we create a node of red degree greater than or equal to the width of the best contraction sequence found so far. Before running the branch-and-bound search, we apply reduction rules (Section 4) to the input graph, compute upper bounds (Section 2) and lower bounds (Section 3), which we use to backtrack in the search.

TinyWidth solver is open source [1]: its source code along with instructions to build and run it can be found at `https://github.com/GBathie/pace2023-tinywidth`. TinyWidth solver was submitted to the exact track of the 2023 PACE Challenge.

1.1 Definitions

The twin-width, introduced in 2020 by Bonnet et al., is a graph parameter defined in terms of *contraction sequences* on *edge-colored graphs*. We consider graphs whose edges are either colored *black* or *red*: the latter corresponds to the presence of errors arising from *contractions*. We use $N(u)$ (resp. $N^r(u)$) to denote the set of vertices connected to u with black (resp. red) edges. Contracting u and v (where $u \neq v$) in the graph G transforms it into a graph G', where u and v are merged into a new vertex w. With the above notation, we set the neighbors of w as follows:

$$N(w) := N(u) \cap N(v) \text{ and } N^r(w) := (N^r(u) \cup N^r(v) \cup (N(u) \Delta N(v))) \setminus \{u, v\}.$$

A contraction sequence for a graph G with n vertices is an ordered list of $n-1$ contractions $(u_i, v_i)_{i=1,\dots n-1}$ such the contraction (u_i, v_i) transforms G_i into G_{i+1}, where $G_1 = G$.

The width of a contraction sequence is the minimum over i of the maximum red degree of G_i, and the twin-width $tww(G)$ of G is the minimum width of a contraction sequence for G.

1.2 Branch-and-bound and memoization

For a given graph G, our solver tries every possible contraction u, v to find the one that starts a contraction sequence of minimum width. However, this leads to redundant computation, as the graph G' obtained by contracting u, v then u', v' in G is the same as the graph obtained when reversing the order of the contractions. In both cases, an optimal contraction sequence for G' will yield the minimum possible width for these starting contraction.

Therefore, we store in a cache the optimal contraction sequence for every graph that we create during the search, and use the result if we re-create the same graph at a later point in time. Deciding whether G appears in the cache requires solving the graph isomorphism problem, which is not known to be solvable in polynomial time. We opt for a simpler solution, which may miss some isomorphic occurrences, using a fingerprint that maps every vertex of G to the index of the smallest vertex in its "contraction class" and can be computed in linear time.

2 Upper bounds

Greedy-merge:

This heuristic repeatedly merges a pair of vertices $(u, v), u \neq v$ that minimizes the red degree of the resulting node, i.e. that minimizes the cardinality of $N^r(u) \cup N^r(v) \cup (N(u) \Delta N(v))$, until there is only a single node left in the graph.

Close-merge:

The above heuristic runs in time $\Theta(n^3)$ (not accounting for the time for comparing neighborhoods), an fails to finish within 5 minutes for larger graphs. The Close-merge aims to reduce this running time by choosing the merged pair (u, v) using random sampling instead of trying every candidate.

Our sampling procedure is based on the following observation: if merging u and v results in a small red degree, then u and v share many neighbors (or they have small degree). To find a candidate pair, we sample a vertex w of degree at least two, and choose u and v among its neighbors. We repeat this process $t = 200$ times and merge the best pair found, and iterate until the graph contains a single vertex.

Tree-merge:

The above two heuristics generally do not perform well on trees (which have twin-width at most 2). We include a specific heuristic to find the exact value for trees, but also for most "tree-like" graphs (in practice, graphs with few cycles and/or large girth).

This heuristic chooses a random a spanning tree T of the graph G and applies to G an optimal contraction sequence for T. We then repeat multiple independent runs of this heuristic and return the best result found.

3 Lower bound

The study of twin-width has only started recently, and no strong lower bounds are known for it. The most frequent approach is to compute the minimum of $N(u) \Delta N(v)$ for $u \neq v$,

but this lower bound is implicitly used in the "early exit" check of our branch-and-bound algorithm, hence it is unnecessary to compute it separately.

However, our solver finds the twin-width of small graph ($n \leq 25$) in a fraction of second. Moreover, if H is an induced subgraph of G, then $tww(H)$ is a lower bound for $tww(G)$. Hence, we are able to obtain a lower bound on the twin-width of G by computing the twin-width of random connected induced subgraphs of size k, sampled by doing a bounded BFS starting from a random node. In practice, we set $k = 25$, and repeat this operation for a fixed amount of time ($60s$).

Finally, we speed-up these lower bound computations by using the value of the current lower bound ℓ for G as a lower bound for the computation in H: if we find a contraction sequence for H that has width less than ℓ, its optimal value will also be smaller than ℓ, hence H will not yield an improved lower bound, and we can backtrack.

4 Pre-processing and reduction rules

As the twin-width of a graph is equal of the maximum of the twin-widths of its connected components (CCs), we start the pre-processing stage by decomposing the input graph into CCs, and solving them separately. Moreover, we propagate the twin-width of the previous components as lower bound for the next one.

We then apply various of *reduction rules*, that aim to merge vertices when we know that it will not affect the width of the solution.

For example, if u and v are twins, merging them does not affect the solution, as it does not create red edges. We can extend this rule to cover all cases that do not create red edges.

▶ **Reduction rule 1** (Generalized twins)**.** If u and v are non-adjacent vertices such that $N(u) \subseteq N(v)$ and $((N(v) \setminus N(u)) \cup N^r(v)) \subseteq N^r(u)$, merge u and v.

Recall that the set of red neighbors of the vertex obtained after merging u and v is $N^r(u) \cup N^r(v) \cup (N(u) \Delta N(v))$: under the above assumption, this set is contained in $N^r(u)$. If u and v are adjacent, this rule can also be applied, but we need to remove u $N(v)$ or $N^r(v)$ (and symmetrically for v) in the above equation.

The following reduction rule only applies to graphs of twin-width at least 2, hence we apply it only when the lower bound is at least 2.

▶ **Reduction rule 2** ("Hanging" trees)**.** If uv is a cut edge in G, such that the connected component $\mathcal{C}$ of v in $G \setminus uv$ is a tree, contract $\mathcal{C}$ into a single (red) edge using an optimal tree contraction sequence.

This reduction rules turns a tree "hanging" from the graph into a path on two edges, the edge connected to rest of the graph remaining black.

The non-local nature of twin-width makes it hard to prove that a reduction rule is correct. Instead of discarding reduction rules for which we could not prove correctness even though they are correct in most cases, we restrict their use to upper bound computation. We call such rules *heuristic reduction rules*.

▶ **Heuristic reduction rule 1** (Paths)**.** If x is an induced path of on at least 6 vertices, contract its 3rd and 4th vertices.

Applying this rule will reduce any induced path of length more than 5 (edges) to a path of 4 edges, with edge colors black-red-red-black.

To compute the upper bound of a graph G, we compute the graph G' obtained by applying heuristic reduction rules to G, run all the heuristics of Section 2 on both G and G', and return the smallest result over these runs.

References

1 Gabriel Bathie, Jérôme Boillot, Nicolas Bousquet, and Théo Pierron. Pace 2023 - TinyWidth solver, May 2023. doi:10.5281/zenodo.7991737.