A Simple Twin-width Searcher

Alex Meiburg
University of California Santa Barbara

Abstract

A simple exact twin-width search was implemented for the PACE 2023 Challenge. The depth-first searcher relies on symmetry breaking and pruning where possible. This document describes the algorithm in detail.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Human-centered computing → Open source software

Keywords and phrases Twin-width

Supplementary Material Zenodo record: doi:10.5281/zenodo.8045233

1 Background

Twin-width is a new and popular parameter in parameterized complexity. They are defined using red-black "trigraphs", in a way that is often described as surprisingly complicated, although parameter is has been shown to be powerful. Computing the twin-width, exactly or heuristically, was the focus of the 2023 PACE Challenge. For the exact track, programs were given 30 minutes to output an optimal twin contraction sequence on a given graph, and 200 graphs were used as a test set.

2 Outline of Algorithm

The search was implemented in Java. Graphs were stored as an adjacency matrix packed into 64-bit longs; the red-black trigraphs needed for exploring contraction sequences are implemented using (mutually exclusive) red and black adjacency matrices. The graph data structure also maintains a degree of each vertex, or in the case of the trigraph, the red and black degrees separately. This was found to perform better than array-backed adjacency lists or hashmap-backed adjacency lists.

2.1 Preprocessing

Initial preprocessing consisted of removing all twins from the graph. Then, the graph was split into components, and the twin-width computed on each component respectively. Solutions from components can be combined by concatenating the solutions and merging all components, and the resulting twin-width is the maximum of the componentwise twin-widths. Each graph was complemented and reduced into components again. As further complementation could not create new components at this point, this process was not repeated; also, no more twin vertices could have been created, so no more were checked for.

An approach was developed for finding bridges or articulation points in the graph and trying to use this to further reduce: an optimal contraction sequence *typically* exists that contracts everything on one side of a bridge first, prior to contracting the rest. That is, by analyzing two appropriately created subgraphs and appropriately combining, an optimal solution could usually be fine. Unfortunately, the word "appropriately" did a lot of heavy lifting. The exact nature of when a sequence could be guaranteed optimal eluded the author, and so the software would create a candidate solution and subsequently attempt to get a matching lower bound on the twin-width. This ended up being slower than the simple search, largely because almost no graphs in the dataset had this "two-lobed" structure best suited to this approach. In the final release, after splitting into components, the search would begin. Brides and articulation points were found using Tarjan's algorithm.

2.2 Searching

Search was depth-first through all contraction sequences, progressively exploring contracting any given pair together. Search maintained the current trigraph, a lower bound and upper bound, and a note of the last two vertices merged (i and j). The lower bound informed the lowest twin-width that any contraction sequence in this branch of search could achieve, that is, the greatest red-degree of any vertex in any parent node in the search tree. The upper bound was the greatest twin-width that we would have any use to find, that is, one less than the current best solution found.

A given node consisted of considering all pairs that can be contracted, recursively searching each one, and updating the current score. After each recursive call, the upper bound was updated to one less than the current score. If the upper bound was ever below the lower bound, then this indicated that no more useful searching could be done, and the remaining children were not searched. Note that this means that when the remaining graph size $|V|$ is below the lower bound, it always returns the lexicographically first contraction sequence.

The largest factor in performance was intelligently excluding certain children based on symmetry: in a large graph, most pairs of contractions (u, v) then (x, y) can be freely interchanged in the contraction sequence without affecting the overall red-degree. So, we could require that our contraction sequences are all lexicographically sorted. (Each pair (u, v) is already sorted, $u < v$.) Before contracting a pair (x, y) and searching a child node, it was compared with the most recent contraction (u, v). If (x, y) was lexicographically before (u, v), and the two could be interchanged, then this child was skipped, as $(x, y)(u, v)$ had already been searched and $(u, v)(x, y)$ was unnecessary. As contractions are confluent (they lead to the same trigraph regardless of order), the only necessary check for interchangeability was whether the intermediate graph after (u, v) but before (x, y) would have a higher red degree than the lower bound, potentially altering the score.

During the search, twins were checked for at each node in search. To accelerate copying of graphs, contracted vertices were allowed to remain in the graph with degree zero, and all logic ignored degree-zero vertices (except for the final postprocessing). This allowed the packed adjacency matrices to be copied directly in memory, without renumbering or shifting to account for vertex deletion.

At the start of the search, one simple greedy heuristic was used to generate an initial contraction sequence and a speculative upper bound. This would often be close to optimal, especially on small graphs. The initial lower bound was zero. The greedy heuristic chose the contraction that would minimize the red degree of the contracted vertex, then as a tiebreaker the one that minimized total red edge count in the graph, then as a tiebreaker the one that would minimize total black edge count in the graph (after an initial possible complementation if the black density was over 1/2), then as a tiebreaker the lexicographically first. After each contraction the heuristic search also checked for twins and deleted them.

That specifies the full implemented algorithm.

3 Room for Improvement

Major improvements can be grouped as implementation details, search techniques, or pruning techniques. Implementation details would generally give a constant factor of improvement in time, but this could be significant:

- Different data structures for large graphs vs small graphs, or dense vs sparse, where performance might favor one data structure over another. The adjacency matrix approach would likely benefit from occasionally doing a "slow copy" and removing all degree-zero vertices, when the graph density is sufficiently low.
- A way to record and rewind graph edits could potentially save time by reducing time to copy the graph, where each node in the search tree can rewind to its parent.
- Special cases to handle small graphs (say, less than size 9) could reduce the time spent on the bottom nodes.

Search techniques improve how successful we are guessing good sequences, or using them:

- Sorting all children by some heuristic.
- More generally, combining breadth- and depth-first search with a queue, delaying nodes that will likely take a long time and that we hope could be pruned.
- Better heuristics for initial upper bound, or applying it multiple times with some randomness.
- Reapplying heuristics at some points in the search tree, to try to cut down on the upper bound. This is known in the optimization context as "plunging": occasional deep dives in the tree. In the current pure depth-first search, this is completely subsumed by the "sort all children" improvement, as the initial depth-first probe is already a good heuristic. It only becomes relevant when the breadth search is incorporated as well.

Pruning techniques allow us to eliminate large swathes of the search tree at once:

- Apply reduction rules, if any exist, that contracting a given pair together is never optimal. For instance, if the graph is sparse and the two vertices have no neighbors in common.
- The lexicographic sorting misses some cases where it cannot confidently prove what can be exchanged, because it has to operate using only the $(u, v)-$contracted graph and the numbers u and v. More information would allow more thorough sorting, and pruning more options.
- Recognizing forbidden subgraphs, or particular structures, can be used to lower bound the twin-width. It is easy to exclude a twin-width of zero (does detwinning remove the whole graph?), but even a minimal set of obstructions to a twin-width of one is not known. A set of obstructions does not need to be complete, however. For example, checking for any large induced cycles or anti-cycles of size five or more (holes and anti-holes) would immediately give a lower bound or two.
- Reducing based on articulation points, bridges, or generally small cut sets could almost certainly be useful, but requires a careful choice of when to apply it. It is most useful when the graph has balanced small cuts, and the resulting solution can be proven optimal given the optimality of the subproblems.
- Using a graph automorphism library, such as NAUTY, to find symmetries of the graph and exclude some options. This could be applied at the start, or at various stages throughout (with appropriate trigraph edge coloring). Most test graphs seemed to have no symmetry, so this feature should disable itself after a number of unsuccessful probes.