HeiTwin - Heuristic Twin Width Solver

Thomas Möller
Heidelberg University, Heidelberg, Germany

Nikita-Nick Funk
Heidelberg University, Heidelberg, Germany

Dennis Jakob
Heidelberg University, Heidelberg, Germany

Ernestine Großmann
Heidelberg University, Heidelberg, Germany

Abstract

We present a novel heuristic approach to efficiently find the approximate twin-width of a given graph. A contraction sequence is computed using a greedy sliding-window approach. The algorithm checks a fixed amount of vertices in every step and greedily performs the best sequence for these vertices. The result is improved until a time limit is reached by increasing the number of vertices looked at in each step of the algorithm and by running the heuristic with different vertex orderings.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases twin-width, heuristic, greedy, sliding window

1 Introduction

The concept of *twin-width* was introduced by Bonnet et al. in 2021 [2]. Intuitively, it describes how much a graph differs from a cograph. Not much is known regarding twin-width other than it being NP-complete to determine whether a given graph has a bounded twin-width of at most four [1]. It is also known to be NP-hard to approximate the twin-width with an approximation ratio better than 5/4 [1]. This work proposes a heuristic approach to quickly find a contraction sequence resulting in a low twin-width for a given graph by using a greedy sliding-window technique.

2 Preliminaries

A *trigraph* is an *undirected, simple* graph $G = (V, E)$, the edge set $E(G)$ is partitioned into the sets $B(G)$ of black edges and $R(G)$ of red edges. A regular graph can be considered a trigraph for which $E(G) = B(G)$ and $R(G) = \emptyset$. The set $N_G(v)$ of neighbours of a vertex $v \in V(G)$ are all vertices adjacent to v by a black or red edge. A neighbour $u \in N_G(v)$ is called a *black neighbour* of v if $uv \in B(G)$ and a *red neighbour* if $uv \in R(G)$. The *red degree* of a vertex $v \in V(G)$ is the number of red neighbours of v. A *d-trigraph* is a trigraph where each vertex has a red degree of at most d.

A trigraph G' is obtained from a trigraph G by *contracting* two, not necessarily adjacent vertices. Two vertices u and v are contracted by merging them into a new vertex w and updating the edges as follows: Every vertex in the symmetric difference $N_G(u)\Delta N_G(v)$ is made a red neighbour of w. If a vertex $x \in N_G(u) \cap N_G(v)$ is both a black neighbour of u and of v it is made a black neighbour of w, otherwise x is made a red neighbour of w. All other edges remain unchanged. A *d-sequence* or *sequence of d-contractions* of a graph G is a sequence of d-trigraphs $G_0, G_1, \cdots, G_{n-1}$ such that $G_0 = G$ and G_{n-1} is the graph on a single vertex. For $i \geq 1$ G_i is obtained from G_{i-1} by contraction. The twin-width *tww(G)* is the smallest integer d for which G admits to such a d-sequence.

3 Data Structure

The graph data structure uses a vector of lists of edge pointers. Edges are structs consisting of the edge target, edge color, a field describing whether it is actually an edge or not and pointers to the next, previous and reverse edge. These edge lists are ordered. Vertices are also structs consisting of the vertex id, whether it is actually a vertex or not, a field describing whether the vertex is active or not, the vertex degree, the vertex red degree as well as pointers to the first and last edge and the next and previous vertices. This vertex struct is structured in a way that it serves as a sentinel for the edge list. By storing vertices in a vector and in a linked list we are able to access arbitrary vertices as well as only iterate over active vertices. Contractions are done by iterating over the edge lists of the vertices to be contracted simultaneously. This approach allows us to more easily see whether two edges connect to the same vertex which results in the correct action being taken for each scenario. Anytime the twin-widht changes the current twin-width of the graph is updated accordingly. Edges which change color during contraction are kept track of to facilitate uncontractions. All other edges do not need to be saved explicitly as they remain unchanged.

4 Solver Overview

Our solver can be split into two major steps: a reduction step and a local search step.

4.1 Reductions

Only two simple and exact reductions are used. First a *DegreeZeroReduction* is performed which finds and contracts vertices of degree zero. These contractions do not contribute to the twin-width as no red edges can be created. Afterwards a *DegreeOneReduction* is performed which searches twins of degree one. This is done by going through all vertices and storing vertices of degree one in an array at the position of its neighbours id. If a vertex is already stored at this position two twins are found and therefore contracted. These contractions also do not contribute to the final twin-width. Both of these reductions run in linear time.

4.2 Local Search

If the input graph is sufficiently small a strategy called *RedDegreeLimit* is performed. The algorithm is started with a limit of 0. All vertex pairs are checked and contracted if the contraction yields a new red degree which is smaller or equal to the current limit. If no such vertex pair is found the limit is increased. The amount by which the limit is increased depends on the size and order of the graph and the number of vertices which were contracted in the last iteration. This is done until the whole graph is contracted. This strategy is expensive but computes good results.

The main part of the algorithm is the *TreeContract* strategy. The idea is that contracting the same vertex many times leads to both a high twin-width and slow running time because the degree of such a vertex increases dramatically. To circumvent this, the vertices are contracted in tree-like manner, i.e. for $|V| = 4$, first the vertex pairs (1, 2) and (3, 4) are contracted followed by (1, 3) in the next iteration.

To possibly find a better solution, the algorithm inspects k consecutive vertices with respect to a given vertex order in each step (instead of just 2). Among these k vertices the pair which yields the best result is contracted. In the next step, the next k vertices in the given order are considered. In summary this algorithm inspects a sliding window which moves over the graphs vertices. This is done until $|V(G)| < k$ after which we sequentially contract the remaining vertices.

This is done twice for two different orderings which turned out to yield good results. In the first iteration the *natural* order, i.e. the order with respect to the vertex ids, is used. This very much depends on the input graph, but turned out to be a very good order for many graphs. In the second iteration a *BFS ordering* is used instead. This way, vertices which are close to each other in the graph also end up close to each other in the vertex order. Between any two iterations the graph is uncontracted to the point after the initial reductions were applied. We always store the best solution found so far.

If there is time to spare after both iterations conclude the value for k is incremented and the strategies are performed again until the time limit is reached. Depending on the order of the input graph the starting value for k is $k \in \{3, 4, 5\}$. It turned out that higher k generally lead to better solutions, which means that the algorithm computes better solutions the more time is given.

References

1 Pierre Bergé, Édouard Bonnet, and Hugues Déprés. Deciding twin-width at most 4 is np-complete. *CoRR*, abs/2112.08953, 2021. URL: `https://arxiv.org/abs/2112.08953`, `arXiv:2112.08953`.

2 Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width I: tractable FO model checking. *J. ACM*, 69(1):3:1–3:46, 2022. `doi:10.1145/3486655`.