


# Exact (GUTHMI) and Heuristic (GUTHM) Computation of Contraction Sequences with Minimum Twin-Width

Alexander Leonhardt ✉  
Goethe University Frankfurt, Germany

Anselm Haak ✉  
Goethe University Frankfurt, Germany

Johannes Meintrup ✉   
THM, University of Applied Sciences  
Mittelhessen, Giessen, Germany

Manuel Penschuck ✉   
Goethe University Frankfurt, Germany

Holger Dell ✉   
Goethe University Frankfurt, Germany

Frank Kammer ✉  
THM, University of Applied Sciences  
Mittelhessen, Giessen, Germany

Ulrich Meyer ✉   
Goethe University Frankfurt, Germany

---

## Abstract

Twin-width ( $\text{tw}$ ) [1] is a parameter measuring the similarity of an undirected graph to a co-graph. It is useful to analyze the parameterized complexity of various graph problems. We submit two algorithms to the PACE 2023 challenge, one to the exact track and one to the heuristic track, to compute the twin-width and provide a contraction sequence as witness. In the exact track the solver has to find the minimal twin-width admissible by the graph while in the heuristic track the goal is to produce a possibly tight upper bound on the twin-width under computational resource constraints.

Our heuristic algorithm relies on several greedy approaches with different performance characteristics to find and improve solutions. For large graphs we use locality sensitive hashing to approximately identify suitable contraction candidates. The exact solver follows a branch-and-bound design, relies on the heuristic algorithm to provide initial upper bounds, and uses lower bounds to show optimality of an heuristic solution found in one branch.

**2012 ACM Subject Classification** Mathematics of computing → Graph algorithms

**Keywords and phrases** PACE 2023 Challenge, Heuristic, Exact, Twin-Width

**Supplementary Material** *Software (Source Code)*: [https://github.com/manpen/twin\\_width](https://github.com/manpen/twin_width)  
*Software (Archived Source Code)*: <https://zenodo.org/record/7996074>

**Funding** *Johannes Meintrup*: Funded by the Deutsche Forschungsgemeinschaft (DFG) – 379157101.  
*Manuel Penschuck*: Funded by the Deutsche Forschungsgemeinschaft (DFG) — ME 2088/5-1 (FOR 2975 — Algorithms, Dynamics, and Information Flow in Networks).

**Acknowledgements** The order of authors is alphabetical with the exception that we moved Alexander Leonhardt to the front as he contributed a more than a proportional amount to the heuristic solver.

## 1 Preliminaries

A *tri-graph*  $G = (V, B, R)$  is an undirected graph  $(V, B \cup R)$  where the edge set is bi-partitioned into so-called *black* edges  $B$  and *red* edges  $R$ . Let  $N_B(v)$  and  $N_R(v)$  denote the open black and red neighborhood of node  $v$ , respectively. Furthermore, denote the black degree  $\text{blkDeg}(v) = |N_B(v)|$  and red degree  $\text{redDeg}(v) = |N_R(v)|$ .

Given a tri-graph, the *contraction* of node  $u$  into  $v$  connects  $v$  to all nodes in the union of the two neighborhoods while retaining the color black of an edge only for edges connected to nodes which appear in the intersection of the two black neighborhoods while coloring

all other edges red. In the following all self-loops are removed and thereby conclude the contraction. Given a simple and undirected graph  $G = (V, E)$ , a *contraction sequence* is a list of  $n - 1$  node contractions transforming the all black tri-graph  $(V, E, \emptyset)$  into a tri-graph with a single node. A graph  $G$  is said to have twin-width  $d$  if  $d$  is the smallest integer such that  $G$  admits a contraction sequence where after each contraction for all nodes  $v$  in  $G'$  (the intermediate tri-graph) it holds that  $\text{redDeg}(v) \leq d$ .

## 2 GUTHM: Greedily Unifying Twins with Hashing and More

Our heuristic solver GUTHM is greedy in nature as it repeatedly selects the locally best contractions to carry out. Based on various heuristics locally suboptimal contractions may be selected, though. Since there are  $\Theta(|V|^2)$  possible merges to consider at each step, a naive greedy approach is prohibitively slow on large graphs. Thus, we use three strategies based on the information available to derive a contraction sequence. A strategy is only changed after the input graph has been fully contracted. Based on the information gathered either a new strategy is employed or the same strategy is used again with a different seed.

- **(P) Priority based:** *Quickly* constructs a somewhat good initial contraction sequence.
- **(S) Sweeping based:** Primarily used as second stage for low twin-width graphs since its runtime characteristics depend on easily reducible graphs.
- **(P+LSH) Priority with support for locality sensitive hashing:** Primarily used as second stage for high twin-width graphs approximating all-pair nearest-neighbor search.

If the input is disconnected, each connected component (CC) can be processed in isolation. Then, node contractions within a CC preserve connectivity. We start by processing each CC using the solver **P**. After establishing a first crude contraction sequence, we repeatedly attempt to improve the partial solution for a CC with the currently largest twin-width. In the following subsections unless otherwise stated a “best” contraction is always defined by a contraction which minimizes the score given by Section 2.4.

### 2.1 P: Priority based solver

The priority based solver **P** always selects a node  $v$  with the smallest red degree and tries to find the best contraction partner for this node (see Section 2.4 for the scoring function). For this evaluation only contraction partners in the 2-neighborhood are considered. If the twin-width is increased due to a contraction involving  $v$  the solver might postpone contracting this node to a later point in time, and, instead, selects the next best candidate.

After a successful merge involving  $v$  the solver contracts the newly created leaves in the direct vicinity of  $v$  (if there are multiple). If there are further “good” contractions involving node  $v$ , they are executed as well. We continue until the intermediate tri-graph is sufficiently small to run an exhaustive final stage solver. The final stage solver considers *all* possible contractions and greedily selects the best contraction at any time.

### 2.2 S: Sweeping based solver

The solver **S** sweeps over all nodes in the graph and carries out contractions that do not increase the twin-width above a certain threshold. On one hand, the threshold helps to guide the solver. On the other hand, it also speeds up the computation as it reduces the number of contraction candidates to consider. As such, we only use this strategy on graphs with a sufficiently low upper bound on the twin-width (previously established by **P** or

**P+LSH**). On top, we employ random sampling to establish an estimate of the new threshold at the beginning of every round further curbing execution speed at the cost of accuracy. In subsequent calls to this solver the threshold and the random samples are continuously tweaked to improve accuracy at the cost of execution speed.

### 2.3 P+LSH: Priority with support for locality sensitive hashing

If the solver **P** found a contraction sequence witnessing a high twin-width graph, it is unlikely that the sweeping solver **S** can process this graph within the time budget. Therefore, we attempt to improve the solution quality of the fast solver **P** by adding global information collected via MinHashing [2].

- **Local information:** Just as solver **P**, we initially select the next vertex  $v$  based on the smallest red degree. The local information is now derived from the possible contractions involving  $v$  and node  $u$  selected from the 2-neighborhood of  $v$ .
- **Global information:** The local perspective, however, fails to identify near-twins with large red degrees. To overcome this restriction, we use a scheme based on MinHashing to identify almost twins by approximating all-pair nearest-neighbor search. From all similar pairs, we order the pairs by their number of collisions in all hash tables and the maximum red degree of the nodes in the pair.

Using MinHashing, we approximately find near twins even in large graphs. Despite the obvious benefits of using MinHashing, it is still important to retain the initial approach of selecting a vertex with the lowest red degree and considering contractions involving this vertex. This is due to the fact that the performance of MinHashing strongly depends on the choice of tuning-parameters, which we cannot efficiently adapt during execution due to time and memory constraints. From the ordered similar pairs we only consider the top- $k$  pairs and select the pair which minimizes the score given in Section 2.4. When the graph becomes sufficiently small, we again switch to an exhaustive final stage solver.

#### 2.3.1 Updates of MinHash-based all-pair nearest-neighbor approximation

Apart from the two nodes  $u$  and  $v$  involved in the current contraction, only neighbors of the survivor  $u$  need to update their hash values in the MinHash table. Since the number of edges changed for any node  $w$  in  $N(u) = N_B(u) \cup N_R(u)$  is at most 2, the probability of needing an update  $P(\text{Update needed for } w)$  is bounded by  $2/\max(|N(w)|, 2)$ . Thus, we can preserve the data structure even for large graphs since the probability of needing to update the neighbors is inversely proportional to their degree.

### 2.4 Move selection

By default, the heuristic solver selects contractions which minimize the following score

$$\text{score}(u, v) = \sum_{(v,x) \in R_{\text{new}}} (\text{redDeg}(x) + 1) - \sum_{(v,y) \in R_{\text{rem}}} \text{redDeg}(y),$$

where  $R_{\text{new}}$  and  $R_{\text{rem}}$  denote the sets of red edges the contraction of  $(u, v)$  introduces and removes, respectively. As long as the contractions do not increase the twin-width of the current intermediate tri-graph this heuristic is employed, otherwise the next contraction is selected greedily by choosing the contraction which induces the smallest increase in the twin-width of the resulting tri-graph.

### 3 GUTHMI: Germanely Unifying Twins with Hashing and Meticulous Inspection

Our exact solver GUTHMI follows the branch-and-bound algorithmic design paradigm. At each level of the recursion, the algorithm potentially follows  $\Theta(|V|^2)$  branches which is prohibitively expensive even for rather small graphs. We use several heuristics to reduce the search space:

- *Safe contraction of twins*: Before processing an (intermediate) graph, we search for exact twins and contract them. For performance reasons, several rules (e.g., for multiple leafs on the same node, general twins, etc.) are dedicated to this idea.
- *Upper bounds*: Before engaging the exact algorithm, we obtain an upper bound from a heuristic solution. This bound may be repeatedly improved during the runtime of the exact solver. It allows us to prune branches that cannot improve the current best solution.
- *Branching order*: We use scoring methods similar to Section 2 to descend into most promising branches first. Thereby, we often discover improvements quite early in the process. These improved upper bounds then translate into even more aggressive pruning.
- *Lower bounds*: Given a graph  $G = (V, E)$  and an induced subgraph  $G'$  of  $G$ , the twin-width of  $G$  is bounded from below by the twin-width of  $G'$ . Based on this observation, we (non-uniformly) randomly sample subgraphs and attempt to solve them exactly in the first 20 seconds of the execution. In many cases (esp. for small graphs with low twin-width), this suffices to prove the heuristic solution optimal. Otherwise, we compare any improved solution to the lower bound, which, upon matching allows us to terminate early.
- *“Conditional lower bounds”*: We pass the maximum red degree of the current contraction sequence candidate down the recursion. Amongst others, this allows us to quickly prune subtrees if an improved solution is found.
- *Infeasibility caching*: Since the upper bound is non-increasing during an execution, a subproblem that cannot improve the upper bound at one point in time, cannot do it later. For this reason we cache small infeasible subproblems to avoid recomputing them later.

Several implementation details helped to shave-off constant factors.

- While descending into the recursion tree, we attempt to reuse as much of the meta-information (e.g., branch scoring) from the parent as possible.
- We heavily rely on meta-programming to compile dedicated solvers for different graph size ranges. For instance, small graphs are kept in bit matrices on stack, while larger graphs are escalated on to the heap. This way, most set operations can fully exploit bit-parallel instructions.

*Final note*: The algorithms are implemented in the relatively new programming language zig[3] which facilitates an explicit management of memory and therefore seemed like a good fit for high performance applications.

---

#### References

- 1 Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. Twin-width i: Tractable fo model checking. *J. ACM*, 69(1), nov 2021. doi:10.1145/3486655.
- 2 A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, page 21, USA, 1997. IEEE Computer Society.

- 3 Zig Software Foundation. Zig a general-purpose programming language and toolchain for maintaining robust, optimal and reusable software, 2023. URL: <https://ziglang.org/>.

