


PACE Solver Description: Exact (GUTHMI) and Heuristic (GUTHM)

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 (tw) is a parameter measuring the similarity of an undirected graph to a co-graph [3]. It is useful to analyze the parameterized complexity of various graph problems. This paper presents two algorithms to compute the twin-width and to provide a contraction sequence as witness. The two algorithms are motivated by the PACE 2023 challenge, one for the exact track and one for the heuristic track. Each algorithm produces a contraction sequence witnessing (i) the minimal twin-width admissible by the graph in the exact track (ii) an upper bound on the twin-width as tight as possible in the heuristic track.

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. It relies on the heuristic algorithm to provide initial upper bounds, and uses lower bounds via contraction sequences to show the optimality of a heuristic solution found in some branch.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

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

Digital Object Identifier 10.4230/LIPIcs.IPEC.2023.37

Supplementary Material *Software (Source Code)*: https://github.com/manpen/twin_width
archived at `swh:1:dir:b01c88158d6a76db60eade64dbb8b9f8121127`
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 more than a proportional amount to the heuristic solver.

1 Introduction

Twin-width has been a recent focus of researchers in the field of parameterized complexity. It was first introduced by Bonnet et al. [3] in the context of model-checking, and further results by Bonnet et al. followed in a series of publications. Previously, there has been only one work on exactly computing twin-width in practice, which is based on a SAT-formulation [5].



© Alexander Leonhardt, Holger Dell, Anselm Haak, Frank Kammer, Johannes Meintrup, Ulrich Meyer, and Manuel Penschuck;

licensed under Creative Commons License CC-BY 4.0

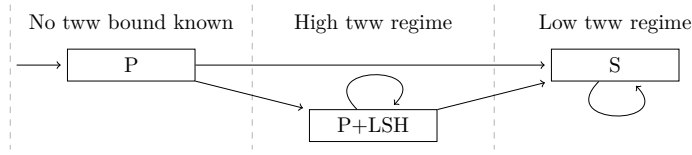
18th International Symposium on Parameterized and Exact Computation (IPEC 2023).

Editors: Neeldhara Misra and Magnus Wahlström; Article No. 37; pp. 37:1–37:7

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Heuristic strategies. The solver always starts with **P** to *quickly* upper bound a connected component's tww. It then attempts to find better solutions with **S** or **P+LSH** based on the best known tww bound. Hence, the solver may only move to the right in the figure.

To our knowledge, there has been no work on heuristics for obtaining contraction sequences of low width. It is currently unknown if there exists an FPT-approximation algorithm for computing the twin-width of a graph, and it is NP-complete to decide whether the twin-width of a graph is at most four [1]. Graphs of twin-width one can be recognized in polynomial time [2], for graphs of twin-width two or three the question remains open.¹

2 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 $N_B(v)$, $N_R(v)$ and $N(v)$, denote the open neighborhood of v in the graphs (V, B) , (V, R) , and $(V, B \cup R)$, respectively, and $N[v]$ denote its closed neighborhood in the graph $(V, B \cup R)$. Furthermore, denote by $\text{blkDeg}(v)$ its black degree $|N_B(v)|$ and by $\text{redDeg}(v)$ its red degree $|N_R(v)|$. The *2-neighborhood* of v is defined as the set of v 's neighbors and their neighbors (excluding v itself).

Given a tri-graph, the *contraction* of node v into u removes v and replaces all edges incident to u by the following new edges: A black edge to any node that had a black edge to both u and v . A red edge to any node that had a red edge to either u or v or was only adjacent to either u or v , but not both. Here, self-loops on u are not added. Intuitively, this can be seen as merging u and v and coloring all differences in their incident edges red.

Given a simple and undirected graph $G = (V, E)$, a *contraction sequence* for G is a list of $n - 1$ contractions transforming the all black tri-graph (V, E, \emptyset) into a tri-graph with a single node. The *width* of such a contraction sequence is the maximal red degree of any node in any of the intermediate graphs obtained by only applying a prefix of the sequence. The *twin-width* of a graph G is the minimal width of any contraction sequence for G .

3 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.

As summarized in Figure 1, 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.

¹ Open problems for twin-width: <http://perso.ens-lyon.fr/edouard.bonnet/openQuestions.html>

- **(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 depends on characteristics exhibited by this kind of graphs.
- **(P+LSH) Priority with support for locality sensitive hashing:** Primarily used as second stage for high twin-width graphs extending the greedy approach with locality sensitive hashing.

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 trivial contraction sequence, we repeatedly attempt to improve the partial solution for a CC with the currently largest twin-width bound. In the following subsections unless otherwise stated a “best” contraction always refers to a contraction minimizing the score given in Section 3.2.

3.1 P: Priority based solver

The priority based solver **P** always selects a node v with the smallest red degree. It then identifies the best contraction partner for this node (see Section 3.2 for the scoring function). To accelerate the second step, we devised a fast method to find all nodes in the 2-neighborhood of v and rank each by the similarity between its neighbors and v 's neighbors:

► **Observation 1.** *A two-level BFS can be adjusted to calculate the symmetric difference between v 's neighborhood and the neighborhood of the nodes in its 2-neighborhood.*

After calculating the score for all nodes in the 2-neighborhood of v as depicted in Figure 2, one can calculate the cardinality of the symmetric difference of the neighborhoods of u and v

$$\text{SD}(u, v) = \begin{cases} |N[v] \oplus N[u]| = |\deg(v)| + |\deg(u)| - 2 \cdot \alpha_u, & \text{if } u \text{ and } v \text{ are adjacent} \\ |N(v) \oplus N(u)| = |\deg(v)| + |\deg(u)| - 2 \cdot \alpha_u, & \text{otherwise,} \end{cases}$$

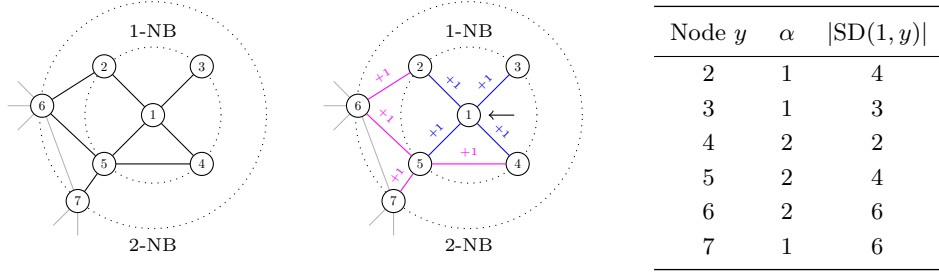
where α_u is the number of visited incoming edges of u during a two-level BFS from v .

► **Lemma 2.** *The calculation of all symmetric neighbor set differences in a graph in the 2-neighborhood of an arbitrary node v is possible in time $\mathcal{O}(|E_{2\text{-NB}}(v)|)$. Here, $|E_{2\text{-NB}}(v)|$ is the total number of edges in the 2-neighborhood.*

Proof. Every visited incoming edge of a node u during a two-level BFS traversal directly corresponds to a shared neighbor with v . The number α_u therefore measures the number of shared neighbors between u and v , which is $|N[u] \cap N[v]|$ for direct neighbors and $|N(u) \cap N(v)|$ otherwise. Since a shared neighbor is present in both u 's and v 's neighborhood, a factor of two is necessary to calculate the symmetric difference of the neighborhoods. The time to traverse all edges in a 2-neighborhood is bounded by $\mathcal{O}(|E_{2\text{-NB}}(v)|)$. ◀

► **Observation 3.** *This approach can be extended to a tri-graph by executing it twice; once on the black induced subgraph and once on the red one. Now a single two-level BFS which only traverses the paths which consist of different colored edges, red-black and black-red, suffices to correct the overestimation provided by the sum of the first two BFS applications.*

The above technique is used to find and rank all possible contraction partners by their neighbor set similarity. After finding and scoring the top- k partners of v a best one is selected as the next contraction partner. 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.



■ **Figure 2** Application of BFS to order all neighbors in the 2-neighborhood by their symmetric neighbor set difference with the source node. Grey edges depict edges from BFS level ≥ 3 while blue and magenta edges depict edges from BFS level 1 and 2 respectively.

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 before selecting a new v . 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.

3.2 Move selection

If the contractions do not increase the twin-width of the current intermediate tri-graph this heuristic is employed, otherwise the next contraction is greedily chosen as a contraction with the smallest increase in twin-width. We say the *best* contractions are those which minimize the following scoring function:

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

where R_{new} and R_{rem} denote the sets of red edges the contraction of (u, v) introduces and removes, respectively.

3.3 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.

3.4 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 [4].

- **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 approximately finding a solution for the closest pair problem. From all similar pairs obtained, 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, we empirically found 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 because 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 a pair that minimizes the score given in Section 3.2. When the graph becomes sufficiently small, we again switch to an exhaustive final stage solver.

Updates of MinHash-based all-pair nearest-neighbor

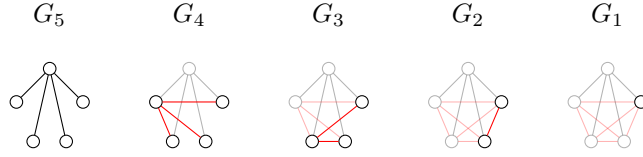
After a contraction of v into u , only neighbors of the survivor u need to be updated in the MinHash table. Observe that at most 2 adjacent edges are changed for any node w in $N(u) = N_B(u) \cup N_R(u)$. Thus, we can preserve the data structure even for large graphs, since the probability of needing to update a neighbor is inversely proportional to its degree.

3.5 Graph reconstruction

Since the heuristic track requires processing of large graphs with only limited memory, we devised a data structure allowing fast reversals of previous contractions. It uses $\mathcal{O}(|V| + |E|)$ memory and has a runtime for reverting contractions proportional to the combined degree of the involved nodes for all practical purposes. As illustrated in Figure 3, any data structure keeping track of all previously existing edges requires $\Omega(|V|^2)$ memory in the worst case. Therefore, any algorithm trying to achieve better memory bounds has to delete previously existing edges at some point, making the reconstruction considerably harder. Other algorithms (*Contraction tree based reconstruction*), which trivially achieve these memory bounds, struggle to reconstruct the previous state with a good time complexity. We present a data structure achieving an acceptable memory consumption in addition to a fast reversal time.

Our data structure keeps track of any deleted node using a union-find data structure. Every red edge keeps track of the node whose removal induced the color switch to red. This value is transferred alongside the edge when it is transferred to a new node. In case of a conflict, that is, when the contracted nodes both have a red edge to the same neighbor, the red edge from the survivor is taken. Furthermore, for every contraction, we use a stack to store certain tokens that allow us to reconstruct the previous state of the graph. Upon a contraction of v into u , v and all incident edges are deleted. For every neighbor $w \in N(v) \cup N(u)$, let $(w_u, w_v) \in \{\emptyset, r, b\}^2$ describe the relationship of u, v with the corresponding node w with $w_x = \emptyset$ if $w \notin N(x)$, $w_x = r$ if $w \in N_R(x)$, and $w_x = b$ otherwise.

From now on the special relationships $w^+ = (w_u, w_v) = (\emptyset, r)$ and $w^- = (w_u, w_v) = (r, \emptyset)$ are distinguished from all other possible relationships. For every neighbor in $N(u) \cup N(v)$ the following case distinction is made alongside the rules to be able to revert a contraction in case of a particular relationship:



■ **Figure 3** Depicts the $\Theta(|V|^2)$ different edges present during the whole lifetime of the graph even though it never exceeds $|E| = \Theta(|V|)$ edges at any point in time.

- $(w_u, w_v) = (\emptyset, b)$ - Put (τ_-, w) on the stack. *Reverse:* $N_R(u) = N_R(u) \setminus \{w\}$.
- $(w_u, w_v) = (b, \emptyset)$ - Put (τ_+, w) on the stack. *Reverse:* $N_R(u) = N_R(u) \setminus \{w\}$ and $N_B(u) = N_B(u) \cup \{w\}$.
- $(w_u, w_v) = (r, b)$ - Put $(\tau_<, w)$ on the stack. *Reverse:* $N_B(v) = N_B(v) \cup w$.
- $(w_u, w_v) = (b, r)$ - Put $(\tau_>, w)$ on the stack. *Reverse:* $N_B(v) = N_B(v) \setminus \{w\}$ and $N_R(v) = N_R(v) \cup \{w\}$.
- $(w_u, w_v) = (r, r)$ - Put $(\tau_=, w)$ on the stack. *Reverse:* $N_R(v) = N_R(v) \cup \{w\}$.

Without going further into detail, every token on the stack either corresponds to a black edge turning red (happens at most **once** for every black edge), or to two edges having a conflict, which necessarily deletes one edge and can therefore only happen $\mathcal{O}(|E|)$ times.

► **Observation 4.** *The cases w^+, w^- correspond to cases where exactly one of the nodes is connected by a red edge to a neighbor w while the other one is not. Determining the source of the red edge is equivalent to looking up the node responsible for the edge turning red in the union-find data structure, which is in the same set as one of the last contraction's nodes.*

By Observation 4 the only potentially memory-wise unbounded cases left can be handled by querying the union-find data structure. Combining the case distinction and the observation above, a contraction can be reverted with the stated memory and time bounds.

► **Lemma 5.** *There exists a data structure using $\mathcal{O}(|E| + |V|)$ memory and supporting the reversal of the last contraction up to the initial graph in $\mathcal{O}(\alpha(|V|) \cdot (\deg(v) + \deg(u)))$ expected time for the reversal of any contraction (u, v) where $\alpha(\cdot)$ is the inverse Ackermann function.*

Proof. The union-find data structure needs at most $\mathcal{O}(|V|)$ memory for the currently deleted nodes. Since all edges are removed from the graph upon a contraction and the number of edges cannot increase by contracting two nodes, the memory needed to store the graph never exceeds $\mathcal{O}(|V| + |E|)$. We only add an item to the stack used to distinguish the different removals from the graph, if two edges are reduced to a single edge or a black edge turns red. Therefore, this can happen at most $|E|$ times. This leads to a total memory consumption dominated by $\mathcal{O}(|V| + |E|)$. Since any reinsertion of edges can be done in linear time, the worst case is having to look up every edge in the union-find data structure. In this case, the running time is bounded by $\mathcal{O}(\alpha(|V|) \cdot (\deg(v) + \deg(u)))$. ◀

The implementation of the heuristic solver often uses approximations of the described techniques to stay within the imposed time bounds.

4 GUTHMI: Germanely Unifying Twins with Hashing and Meticulous Inspection

Our exact solver GUTHMI follows the branch-and-bound paradigm. At each level of the recursion, the algorithm potentially follows $\Theta(|V|^2)$ branches which is prohibitively expensive even for 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 leaves 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 3 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, 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 optimality of the heuristic solution. 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 compile dedicated solvers for different graph size ranges using meta-programming. For instance, small graphs are kept in bit matrices on stack, while larger graphs are escalated on to the heap. This way, most set operation implementations are bit-parallel.

5 Conclusion

We presented two solvers for approximately and exactly finding contraction sequences of small width. The solvers are able to handle arbitrary graph classes with varying success, they are generic in that sense. Further research directions might involve using the weighted Jaccard similarity such that one can directly approximate the score in Section 3.2.

References

- 1 Pierre Bergé, Édouard Bonnet, and Hugues Déprés. Deciding twin-width at most 4 is np-complete. In *ICALP 2022*, volume 229 of *LIPICs*, pages 18:1–18:20, 2022. doi:10.4230/LIPICs.ICALP.2022.18.
- 2 Édouard Bonnet, Eun Jung Kim, Amadeus Reinald, Stéphan Thomassé, and Rémi Watrigant. Twin-width and polynomial kernels. *Algorithmica*, 2022. doi:10.1007/s00453-022-00965-5.
- 3 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.
- 4 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. doi:10.1109/SEQUEN.1997.666900.
- 5 André Schidler and Stefan Szeider. A SAT approach to twin-width. In *ALENEX 2022*, pages 67–77. SIAM, 2022. doi:10.1137/1.9781611977042.6.