Technische Universität Berlin
Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Algorithmics and Computational Complexity (AKT)

Heuristics for Twin-Width

Denis Koshelev

Thesis submitted in fulfillment of the requirements for the degree
"Bachelor of Science" (B. Sc.) in the field of Computer Science

November 2023

First reviewer:	Prof. Dr. Mathias Weller
Second reviewer:	Prof. Dr. Stefan Schmid
Supervisors:	Tomohiro Koana

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den **01.11.2023**

Datum

Unterschrift

Zusammenfassung

Diese Arbeit befasst sich mit der Berechnung der Twin-Width von Graphen unter Verwendung einer Reihe von Heuristiken, die darauf abzielen, optimale Kontraktionssequenzen zu erreichen. Die Forschung geht Herausforderungen bei der Bestimmung der Twin-Width an, indem sie Techniken wie zufällige Irrfahrt von ausgewählten Knotenpunkten und die Verwendung von Knotengraden zur Identifizierung von Hauptkandidaten einsetzt. Die Studie führt auch nicht-triviale "fast" Reduktionsregeln ein. Obwohl diese Regeln keine strengen theoretischen Garantien bezüglich der Lösungsgröße bieten, haben sie einen bemerkenswerten Einfluss auf den Verlauf der Kontraktionsbreite. Durch die Kombination dieser heuristischen Ansätze bietet die Studie eine umfassende Methode, die Ergebnisse für verschiedene Grafentypen verbessert. Ein flexibler Solver wird entwickelt, der als solide Grundlage für weitere Untersuchungen zur Twin-Width dient. Diese Arbeit vertieft nicht nur unser Verständnis von Twin-Width, sondern verfeinert auch verschiedene Parameter und geht dabei auf Neuberechnungen von Scores von Knotenpaaren, Funktionsauswahl und mehr ein. Durch die Verschmelzung von theoretischen Erkenntnissen mit heuristischen Methoden erweitert diese These unser Wissen über Twin-Width und ebnet den Weg für die Entwicklung von geschickteren Algorithmen für komplexe Grafenstrukturen.

Abstract

This thesis addresses the computation of the twin-width of graphs using a range of heuristics focused on achieving optimal contraction sequences. The research tackles challenges in twin-width determination by employing techniques such as random walks from selected vertices and using vertex degrees to pinpoint prime candidates. The study also introduces non-trivial "almost" reduction rules. While these rules lack strict theoretical guarantees about solution size, they have a notable impact on the progression of contraction width. By combining these heuristic approaches, the study offers a comprehensive method that improves outcomes across different graph types. A flexible solver is developed, serving as a solid foundation for further exploration into twin-width. This work not only deepens our understanding of twin-width but also refines various parameters, delving into score recalculations, function choices, and beyond. Merging theoretical insights with heuristic methods, this thesis enhances our knowledge of twin-width and sets the stage for the development of more adept algorithms for intricate graph structures.

Contents

Chapter 1

Intro

This chapter provides an overview of the thesis, covering motivation and related work.

1.1 Motivation

The parametrized complexity of algorithms is an evolving area of research. Initially introduced by Gurevich, Stockmeyer, and Vishkin [GSV84] in their work on solving NP-hard problems on graphs that are almost trees and an application to facility location problems, this field has significantly evolved by exploring new classes of structures and estimating their boundaries.

First-order model checking holds notable significance in this area. It is a technique to evaluate if a given first-order formula holds true for a specified structure, aiding in the analysis and understanding of complex graph structures and ensuring the correctness of systems modeled by these graphs. The initial exploration of tractable FO model checking started with bounded-degree graphs, as demonstrated by Seese [See96]. This work set the stage for various generalizations across diverse graph classes. For instance, Grohe, Kreutzer, and Siebertz [GKS14] showcased that FO model checking is tractable on nowhere dense classes of graphs. Meanwhile, explorations into dense graph classes started to grow. Important steps forward include the discovery of Linear Time Solvability in MSO_2 for graphs with bounded clique-width by Courcelle, Makowsky, and Rotics [CMR00], and the FPT algorithm for general FO model checking on posets of bounded width as illustrated by Gajarský et al. [Gaj+15], among other significant contributions.

Finally, in 2020, Bonnet et al. [Bon+20] expanded the fixed-parameter tractability of FO model checking by introducing a new graph parameter called *twin-width*. This theoretical result is important since it builds a bridge between dense and sparse graphs in FO traceability world by generalizing some of the classes. Subsequently, several new insights regarding twin-width have become available: posets of width d have a twin-width of at most $9d$, as stated by Balabán and Hlinený [BH21]. Additionally, Bonnet et al. [Bon+21] established that deciding if a graph has twin-width at most 1 can be done in polynomial time, and Schidler and Szeider [SS21] introduced an efficient SAT-encodings for the twin-width problem.

Intuitively, twin-width is a graph parameter that measures how close a graph is to a cograph. A cograph is a special type of graph that can be transformed into a

single vertex by repeatedly identifying and merging pairs of vertices, known as twins, that share the same neighbors. Finding and merging such twins is a straightforward process in a cograph, leading to a zero twin-width. However, when dealing with graphs that are not cographs, we might encounter vertices that don't share all neighbors in common. This discrepancy is marked with red edges, representing a penalty for the non-common neighbors. The accumulation of these penalties, measured through red edges, helps determine the graph's twin-width, indicating the extent of deviation from the ideal cograph structure.

An optimal sequence of vertex pairs that minimizes this penalty is called a d-sequence, where d represents the twin-width of the graph. Knowing this d-sequence is crucial, as we cannot solve FO model checking without it. This brings us to a crucial question: Are there any reasonable methods to estimate the twin-width for general graphs and derive a corresponding d-sequence? This bachelor thesis explores various approaches to determining such sequences for general graphs as precisely and quickly as possible.

1.2 Our contributions

In this bachelor thesis, our primary goal is to delve into various heuristic methods to estimate the twin-width of graphs, focusing on obtaining close to optimal contraction sequences. The journey begins with a thorough exploration of the inherent challenges associated with twin-width determination. Chapter 2 introduces all necessary definitions and notations for understanding the twin-width problem. Chapter 3 provides a foundational understanding, from examining the limitations of brute force approaches to a preliminary analysis of graph instances. A key takeaway is the non-trivial nature of vertex contraction decisions.

Chapter 4 delves into strategies for identifying vertex pairs. We harness vertex degree information in Subsection 4.1.1 to filter potential vertex pair candidates, followed by implementing multiple random walks in Subsection 4.1.2, which capitalizes on the observation of optimal vertex pairs' neighborhood relations. A hybrid approach, explored in Subsection 4.1.3, leverages the strengths of both methods based on the graph's degree deviation. As we venture deeper, Section 4.2 discusses various score functions and threshold choices to gauge the optimality of vertex pairs, primarily focusing on neighborhood sizes and degree differences.

Chapter 5 offers a shift towards optimization techniques. The "almost" reduction rules in Section 5.1 present both straightforward preprocessing techniques, such as twin elimination, and more ambitious approaches. However, different strategies for the one-degree rule in Subsection 5.1.2, despite their potential, did not significantly improve the solution size. We then transition to the utility of caching scores in Section 5.2, emphasizing the balance between computational efficiency and result accuracy. We also consider bipartite graphs in Ssection 5.3. Interestingly, our heuristics' natural tendency to contract vertices from the same partition means the bipartiteness of the graph wasn't as advantageous as initially expected. Lastly, the density trick outlined in Section 5.4 showcases a simple yet effective method to significantly improve both runtime and solution sizes.

Conclusively, this thesis does not only focus on the theory but also provides a practi-

cal lens for estimating contraction sequences with a low width. Alongside our theoretical contributions, we have developed a solver, which has been crucial in benchmarking different strategies. This tool, available for further exploration and contributions [Kos23], lays the groundwork for future exploration, setting a firm foundation for continued research into the intricacies of the twin-width problem.

1.3 Related work

In 2023, the PACE Challenge designated Twin-Width as the problem of the year for both exact and heuristic categories. This selection highlights the problem's importance and intricacy. The challenge prompted the creation of a diverse range of innovative solvers, tailored to handle exact and heuristic computations of the twin-width of graphs. Since our work and the challenge were conducted in parallel, there are instances where our thesis complements and overlaps with the results of some solvers presented in the challenge.

Among the heuristic solvers, *GUTHM: Greedily Unifying Twins with Hashing and More* [ALP23] stood out with its greedy approach, employing different strategies based on the information gathered from previous contractions. Similarly to our solver, it employs a random-walk approach but in a different variation. *Zygosity* [EAV23] adopts a randomized greedy heuristic, considering multiple random contractions at each step and selecting the best one based on red degree and intersection size. In our work, we consider a similar heuristic that we call the intra-set pairing heuristic and, as a result, extend this work. We explore how the total degree and red degree affect the choice of the vertices of the potential contraction. *RedAlert* [BD23] follows a unique scheme involving estimation, sampling, and selection of candidate pairs for contraction, switching to cruder heuristics as time runs out.

On the exact solvers' front, *Hydra Prime* [YMS23] performs modular decomposition with upper- and lower-bound algorithms, introducing innovative approaches like timeline encoding and hydra decomposition for enhanced performance. *GUTHMI* [ALP23], on the other hand, employs a branch-and-bound paradigm, leveraging heuristics to trim down the search space.

In addition to the challenge results, there are some theoretical findings concerning upper bounds for the twin-width of specific graphs. For example, Hliněný and Jedelský [HJ23] demonstrated that the twin-width of planar graphs is at most 8 and reduces to at most 6 for bipartite planar graphs. These findings suggest alternative approaches that might be employed to estimate a contraction sequence.

Chapter 2

Preliminaries

This chapter is used to define preliminary terms and notation. We define the rudimentary elements such as vertices, edges, and neighborhoods, followed by more specific concepts like contraction sequences, black and red neighborhoods, and twin-width.

Basic definitions

$G = (V, E, R)$ A trigraph G is a triple, comprising a set V of vertices, a set E of all edges and a set R of red edges. Each edge is a 2-element subset of V;

$V(G)$ the *vertex set* of G;

$E(G)$ the *edge set* of G with $E(G) \subseteq \binom{V(G)}{2}$; for an edge $e = \{u, v\} \in E(G)$ the two vertices u and v are called *endpoints* of e;

$R(G)$ the *red edge set* of G, edges colored red;

$B(G)$ the *black edge set* of G; edges colored black, formally $B(G) := E(G) \backslash R(G)$;

$|S|$ the *cardinality* of a set S is the number of elements in S;

V the number of vertices, formally, $V := |V(G)|$;

E the number of edges, formally, $E := |E(G)|$;

$N(v)$ the (open) *neighborhood* of v, formally, $N(v) := \{u \in V; \{u, v\} \in E(G)\}$;

$N[v]$ the (closed) *neighborhood* of v, formally, $N[v] := N(v) \cup \{v\}$;

$\deg(v)$ the *degree* of v, formally, $\deg(v) := |N(v)|$;

$\operatorname{red}(v)$ the *red degree* of v, formally, $\operatorname{red}(v) := |N_R(v)|$;

$G[U]$ the *induced subgraph* of G on $U \subseteq V$, formally, $G[U] := (U, E(G) \cap \binom{U}{2})$;

$G - V'$ the graph obtained from G by deleting the vertices $V' \subseteq V(G)$, formally, $G - V' := G[V(G) \backslash V']$;

$N_R(v)$ the *vertex* set adjacent to the vertex v with a red edge, the red open neighborhood of v, formally, $N_R(v) := \{u \in V; \{u, v\} \in R(G)\}$;

$N_B(v)$ the *vertex* set adjacent to the vertex v with a black edge, the black open neighborhood of v, formally, $N_B(v) := \{u \in V; \{u, v\} \in B(G)\}$;

$N_R[v]$ the *vertex* set adjacent to the vertex v with a black edge including v itself, the black closed neighborhood of v, formally, $N_R[v] := N_R(v) \cup \{v\}$;

$N_B[v]$ the *vertex* set adjacent to the vertex v with a black edge including v itself, the black closed neighborhood of v, formally, $N_B[v] := N_B(v) \cup \{v\}$.

The notion of twin-width is explored through a sequence of operations referred to as *contractions*. Unlike the traditional notion of edge contraction, our definition of contraction operates on a pair of vertices and follows a unique set of rules, as described below:

Contraction of two vertices For a vertex pair $(v, u) \in V(G)$, where either $(v, u) \in E(G)$ or $(v, u) \notin E(G)$, we define a special operation called *contraction* as follows

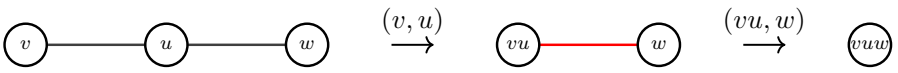
1. If an edge $(v, u) \in E(G)$ exists, it gets deleted;
2. For every vertex w, if $w \in N_B(v) \cap N_B(u)$, an edge (v, w) remains black
3. For every vertex w, if $w \in N(v)$ but $w \notin N(u)$, an edge (v, w) gets colored red, formally, $(v, w) \in N_R(v)$ and $(v, w) \notin N_B(v)$. If an edge was already red, nothing changes.
4. For every vertex w, if $w \notin N(v)$ but $w \in N(u)$, a red edge (v, w) is added to the graph, formally, $(v, w) \in N_R(v)$ and $(v, w) \notin N_B(v)$
5. u gets deleted from the graph, formally, $u \notin V(G)$ and $\forall w \in V(G).(u, w) \notin E(G)$

This unique definition of contraction helps understand a *contraction sequence* and the calculation of the *twin-width* of the graph.

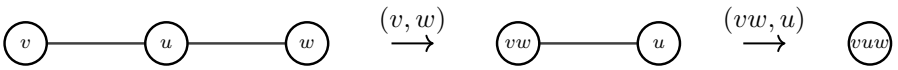
Contraction sequence A *contraction sequence* for a graph $G = G_1$ is defined as an ordered sequence of vertex pairs $((v_1, u_1), (v_2, u_2)...(v_{n-1}, u_{n-1}))$, where $|V(G)| = n$, following the rules:

1. $v_1 \in V(G_1)$ and $u_1 \in V(G_1)$
2. A vertex pair (v_1, u_1) is contracted so that one gets an induced graph $G_2 \subset G$ where $v_2 \in V(G_2)$ and $u_2 \in V(G_2)$
3. It holds inductively for all v_i and u_i in G_i where $i \in [n]$
4. After last contraction of vertex pair (v_{n-1}, u_{n-1}) we get $|V(G)| = 1$

Twin-width The *width* of a contraction sequence S is the maximum red degree of any vertex that appears during the contraction process denoted by width(S), formally width(S) := $\max_i \left\{\max_{v \in V(G_i)} \text{red}(v)\right\}$. Width of the contraction sequence S at i-th step is denoted by width(S_i) where $S_i \subseteq S$ for all $i \in [V-1]$ where V is a number of vertices in the graph. The *twin-width* of the graph $tww(G)$ represents the minimum possible width achievable through a contraction sequence.



(a) Suboptimal contraction sequence.



(b) Optimal contraction sequence.

Figure 2.1: Examples of contraction sequences on a path.

In all our figures, for illustrative purposes, we depict the result of contracted vertices as a combination of their names. Refer to Figure 2.1 for an illustration of two distinct contraction sequences. In subfigure (a), the vertex pair (v, u) is contracted suboptimally. As a result, since vertex u has a unique neighbor w, the edge to w becomes red, elevating the contraction width to 1. In contrast, subfigure (b) showcases a more efficient approach. Here, the twin pair (v, w) is identified and contracted, followed by the contraction of the remaining vertices. This results in a contraction width, and correspondingly a twin-width of the graph, of 0.

Experimental Environment All the experiments in this study were run on a virtual machine (VM) to keep the environment consistent. The VM operated on Ubuntu 22.04 with an aarch64 architecture and was given 4GB of memory to ensure it had enough resources to carry out the tests. The program used for the experiments was not parallelized and was configured to run always with one thread, ensuring a straightforward execution and evaluation. This setup helped in obtaining reliable results throughout the experiments and also makes it easier for others to reproduce the work done in this thesis.

Chapter 3

Problem Insights

Bergé, Bonnet, and Déprés [BBD21] showed that computing twin-width at most 4 is a NP-complete problem. In this chapter, we analyze the twin-width problem to understand better that aids in devising efficient solutions. In Section 3.1, we examine the time complexity of the brute-force approach. In Section 3.2, we delve into the cost of contracting vertices. In Section 3.3, we analyze heuristic instances from the PACE challenge to better understand the problem domain. In Section 3.4, we explore the uncertainty in selecting a vertex pair for contraction, discussing how a locally optimal choice may lead to a worse contraction width globally. Through this examination, we aim to lay the groundwork for more effective algorithm development.

3.1 Time complexity for all possible merges

Let's first consider the brute-force approach for iterating through all possible contraction sequences. We define it as follows:

1. Suppose we order all vertices $V(G) = (v_1, v_2, ...v_n)$ with $|V(G)| = n$. Then, for vertex v_1 there are $n-1$ candidates for a possible contraction. For vertex v_2 there are correspondingly $n-2$ candidates since we consider contraction of vertex pair (v, u) and (u, v) as equivalent, even though it results in different but isomorphic graphs. We define $T(n)$ as a number of all possible vertex pair contractions on a graph with n vertices. Therefore, for the first graph state G_1, where $|V(G_1)| = n$, number of all possible merges is defined as follows:

$$T(n) = (n-1) + (n-2) + ... + 2 + 1 = \frac{n \cdot (n-1)}{2}$$

2. For every subsequent induced graph G_{i-1} after successful contraction in the graph G_i holds that $|V(G_{i-1})| = |V(G_i)| - 1$. As a result, we can inductively define a number of steps for the following graph:

$$T(n-1) = \frac{(n-1) \cdot (n-2)}{2}$$

3. To compute the total time complexity of a brute-force approach we would need to multiply all possibilities in each step:

$$T(n) \cdot T(n-1) \cdot ... \cdot T(2) = \frac{n \cdot (n-1)}{2} \cdot \frac{(n-1) \cdot (n-2)}{2} \cdot ... \cdot \frac{2 \cdot 1}{2} = \frac{n \cdot (n-1)^2 \cdot ... 2^2 \cdot 1^2}{2^{n-1}} = \frac{n! \cdot (n-1)!}{2^{n-1}}$$

Considering a number of all possible merges, we can conclude that the upper bound for a brute-force would be $\mathcal{O}(n!)$ since a factorial function grows faster than an exponential with a constant base.

While it might seem clear that brute-force isn't suitable for our algorithms, the steep time complexity underscores its inefficiency. It emphasizes the necessity for a more intelligent and cost-effective method to identify optimal pairs.

3.2 Time complexity for contracting vertices

In our work, we use an adjacency list as the method for storing graph edges. This choice is primarily driven by the nature of our graph instances, as we discuss in 3.3. These instances are both vast in size and sparse in terms of connections, making the adjacency matrix representation impractical and inefficient. As mentioned before, contracting a vertex pair consists of 5 steps with the following complexities:

1. **Removing the edge** (v, u) **if it exists:** This operation has a time complexity of $\mathcal{O}(E)$, where E is the number of edges.

2. **Updating the red edges for common neighbors of** v **and** u**:** The time complexity is $\mathcal{O}(\deg(v) + \deg(u))$.

3. **Marking the unique neighbors of** v **as red:** This operation takes $\mathcal{O}(\deg(v))$.

4. **Transferring unique red edges from** u **to** v**:** The complexity of this step is $\mathcal{O}(\deg(u))$.

5. **Removing the vertex** u**:** For this operation, the time complexity is $\mathcal{O}(V + E)$, where V is the number of vertices.

Given the above steps, the overall worst-case time complexity of the vertex merging operation using the adjacency list representation is $O(V + E)$. Due to the dominating growth of the factorial function, the $\mathcal{O}(V + E)$ complexity of the vertex pair contraction becomes negligible in the context of the overall complexity.

3.3 Preliminary instances analysis

In developing heuristic algorithms, understanding the dataset's structural properties is paramount. Prior to formulating our heuristic, we undertook an analysis to ascertain key attributes of the graphs in our instances. In our work, we mainly use both public and private instances from the PACE Challenge heuristic track if we don't explicitly mention some other types of graphs. We use the same limits as for the challenge; in particular, run our solver on each instance for 5 minutes.

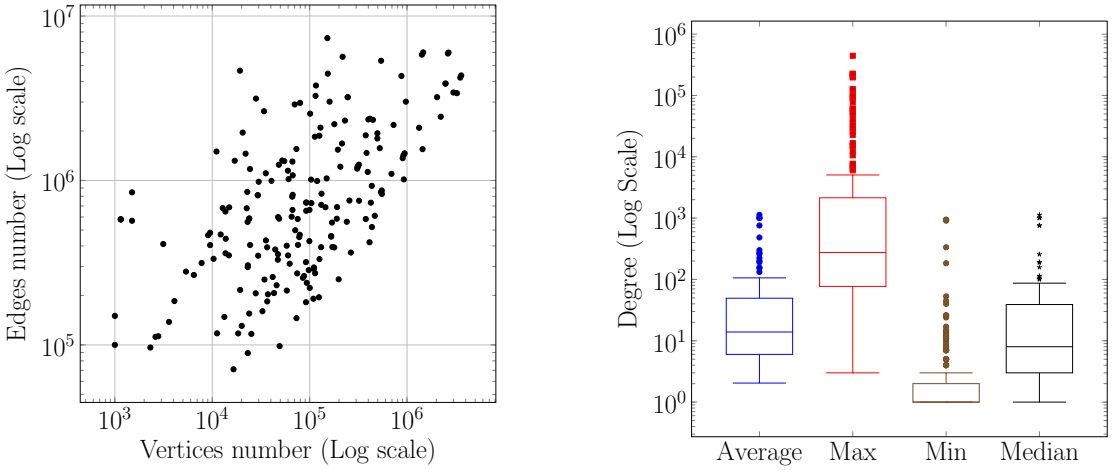


Figure 3.1: On the left plot, logarithmic distribution of vertices against edges, on the right plot boxplot representation of degree metrics on a logarithmic scale

The primary examination focused on graph size by plotting vertex against edge distributions. As seen in the left subplot of Figure3.1, most instances are concentrated around 10^5 vertices and 10^6 edges. Notably, there are outlier instances with a substantial scale of 10^6 vertices and 10^7 edges, highlighting the need for a scalable heuristic approach.

Further structural analysis was conducted on the degree distribution. Degree, a crucial parameter, offers insights into the potential complexity and challenges of heuristic methods. The box plots of degree metrics indicate the presence of both dense and sparse graphs in the dataset. Notably, there exist large graphs with a low average degree, necessitating a heuristic adaptable to both graph size and density variations. Besides, we found that 19 out of 200 graphs were bipartite, and 12 were planar. We try to handle some of such special cases in Section 5.

Connected Components In our analysis of the graphs, one observation stood out clearly: most of these graphs, in particular 135 instances out of 200, consist of just one connected component.

However, some graphs, especially the larger ones, still contain numerous separate components, as shown in Figure 3.2. When a graph breaks down into these separate components, we can approach each as an individual problem. Since the width of the contraction is determined independently for each component, the overall complexity

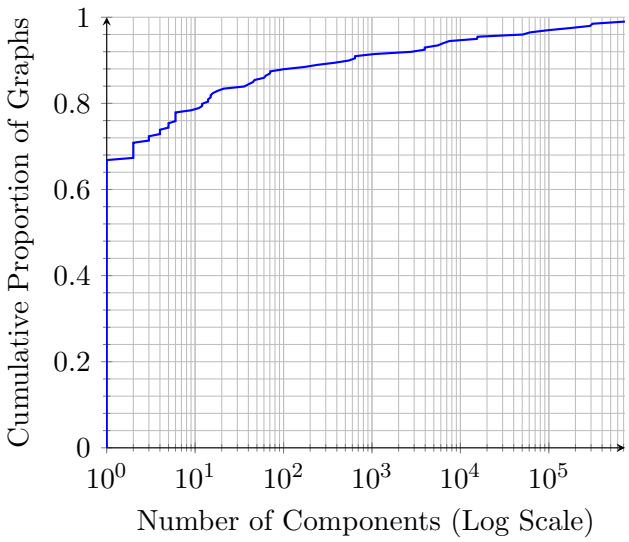


Figure 3.2: Cumulative distribution function (CDF) plotted on a logarithmic scale, illustrating the proportion of graphs with multiple connected components

for the entire graph is essentially defined by its most complex part. In other words, we're essentially paving the way for the entire graph's solution by solving for the most challenging component.

Further refinements and improvements of our default solver would always be performed on each connected component individually since otherwise results might be screwed by merging vertices from different components.

3.4 One merge affects further merges

The main challenge in computing an optimal contraction sequence is that a locally optimal decision—a contraction that minimizes the emergence of new red edges and minimally increases the width of the graph—can potentially lead to less favorable outcomes in the end. As of now, it's unclear how a non-optimal contraction step, judged by the number of new red edges it produces, might contribute to achieving the most optimal contraction sequence, which in turn would define the twin-width of the graph.

Consider the following example in the Figure 3.3. We have two contraction sequences S^* and S that consist of the same vertex pairs for the first 6 contraction steps, so we end up with G_7 as the current state of the graph. In S^* as a next step we contract vertex pair $(6, 3)$ and get a corresponding width of 3. But by contracting $(3, 8)$ in S we still remain width unchanged with width equals 2. However, if we consider all possible contractions in the next step on the graph $G_{8,2}$, we won't be able to find any pair that wouldn't increase the width of the next contraction sequence to 4. On the contrary, we can find such pairs for S^*, which showcases that locally optimal steps might cause a suboptimal contraction sequence as a result.

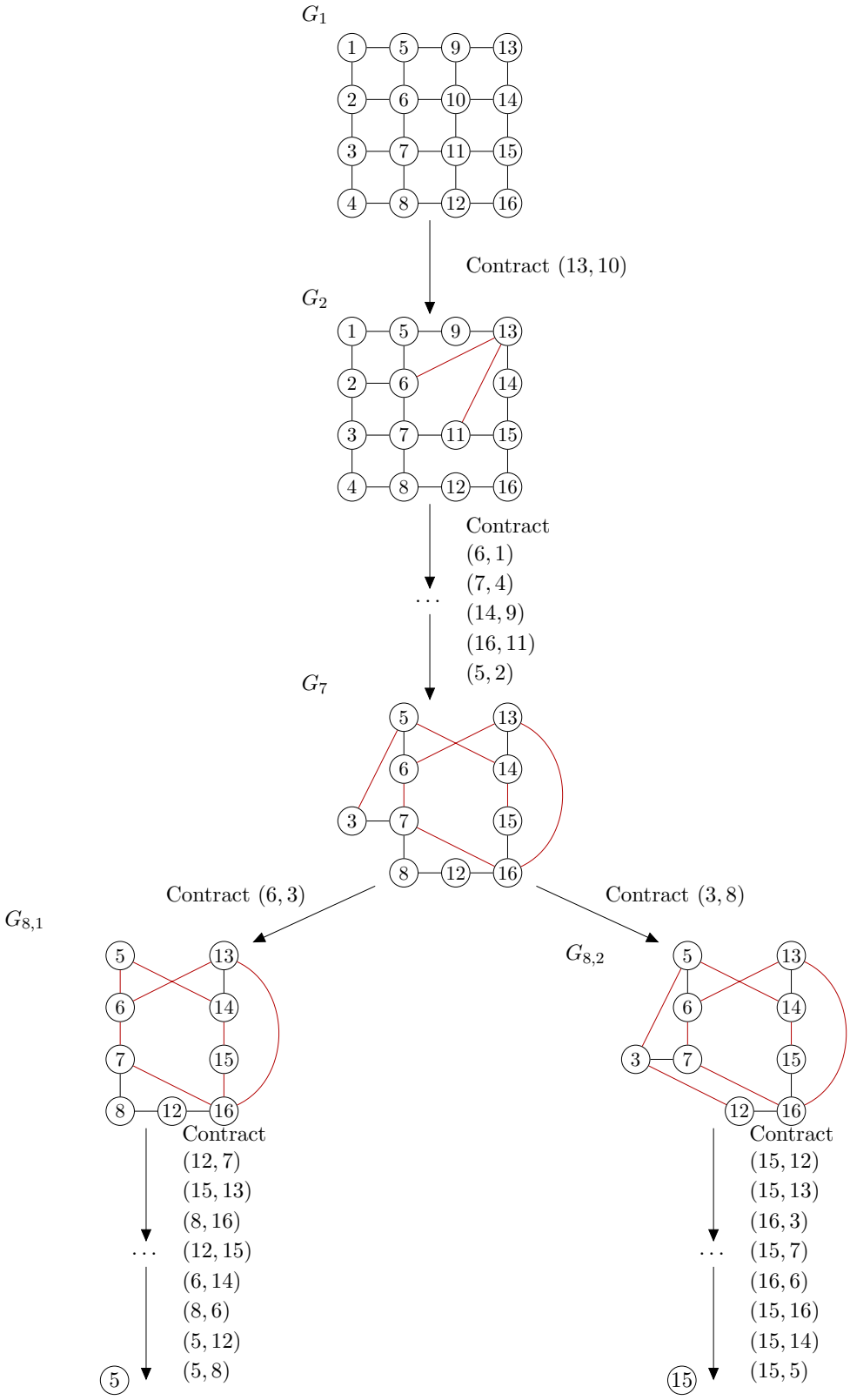


Figure 3.3: Example of optimal contraction sequence S^* on the left and suboptimal S on the right with $\text{width}(S^*) = 3$ and $\text{width}(S) = 4$

Chapter 4

Heuristics for finding Twin-Width

In this chapter, we delve into the development and evaluation of heuristics aimed at efficiently discovering contraction sequences with low width. Initially, in Section 4.1, we explore various strategies to identify promising vertex pairs for contraction based on the subsequent red edges generated post-merging, substantiated with experimental evaluations. Section 4.2 introduces our approach to approximating optimal vertex pairs through different measurement metrics, addressing the inherent challenge in assessing the suitability of a vertex pair for contraction, as mentioned in 3.4, and discusses the decision of threshold selection, particularly in the context of larger graphs, to fine-tune the performance of our heuristics. Through this chapter, we aim to establish a solid foundation of heuristics to navigate the challenges of twin-width computation.

4.1 Strategies of finding vertex pairs

In this section, we explore strategies for finding vertex pairs to contract. In Subsection 4.1.1, we utilize vertex degree information for pairing similar vertices. In Subsection 4.1.2, we design a heuristic based on the random walk from the vertices of the low degrees. Finally, in Subsection 4.1.3, we combine both approaches to enhance the pairing results.

4.1.1 Intra-Set pairing

One of the ideas is to start building a valid contraction sequence through vertices with small neighborhood sizes to avoid triggering such vertex pairs that would cause a drastic red degree increase after a single contraction. The intuition behind that is that the lower the vertex degree, the fewer edges from this vertex pair can become red.

We also implement the same strategy with a slight modification: instead of considering the total degree of vertices, we use only the red degree. Compared to the former version, where we don't account for any information related to the red edges while vertex selection, this heuristic considers the red degree. This objective is important since it might directly influence the final width of the contraction. The idea for both heuristics looks as follows:

1. Begin with an empty contraction sequence S.

2. Until the graph is contracted down to a single vertex, repeat the following steps:
   (a) Select the first η vertices from $V(G)$ with the smallest degree (or red degree in the modified version), denoting them as candidates.
   (b) Among these candidates, find the pair of vertices (v, u) that, when contracted, yields the smallest score according to a scoring function $score(v, u)$ whose variations we discuss in 4.2.1.
   (c) Add this pair (v, u) to the contraction sequence S, and update the graph G by contracting this pair.
3. Once the graph is contracted down to a single vertex, the contraction sequence S is returned as the output.

For the benchmarking, we choose $\eta = 20$ as the number of vertices that we take in each iteration. That means that we would consider $\frac{20 \cdot 21}{2} = 210$ pairs at each iteration. Further discussion of choosing η follows in 4.2.2. Here are the results of both versions of the heuristic:

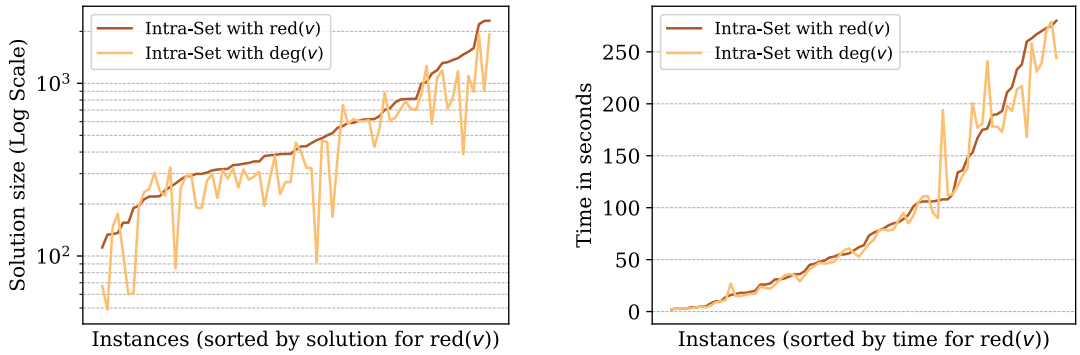


Figure 4.1: Comparison of intra-set heuristics based on low red degree or degree vertices. On the left, the solution sizes are displayed on a logarithmic scale, and on the right, the execution times.

Surprisingly, the degree-based heuristic performed much better than the red degree one, both in terms of solution size and required time. It solved 61 instances better compared to 14 for the latter one, and 52 instances quicker compared to 17 for the red degree based.

Our understanding of the results is the following: even though the degree-based heuristic doesn't consider red edges for filtering out the vertices, it minimizes the neighborhood size of the vertex pair. This implies that the number of potential red edges is bounded by this neighborhood size. On the contrary, the red degree heuristic filters out the vertices with the least red edges at the moment. However, this doesn't reveal anything about potential red edges after the contraction.

In Figure 4.2, we can see how the width of the contraction sequence changes after each contraction step. We visualized it for three cases: where solution size didn't differ much and where it was dominated by one of the heuristics significantly. In all cases, we can see

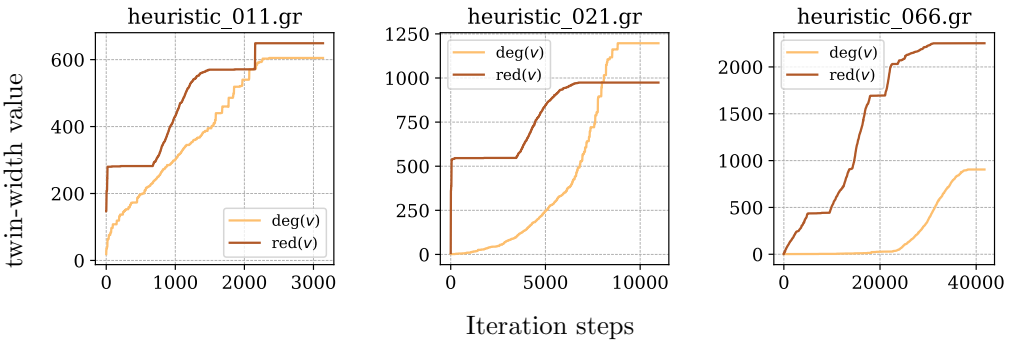


Figure 4.2: Change of the width of the contraction sequence over iteration steps. On the left plot both heuristics performed similarly, on the middle one the degree heuristic dominates, and on the right plot the red degree heuristic prevailed, all sorted by twin-width value for $deg(v)$.

one pattern clearly: degree-based heuristics doesn't have such drastic jumps in solution size compared to the one based on the red degree. This makes sense in the context of our previous hypothesis: the red degree heuristic doesn't consider total neighborhood sizes of the vertices and hence fails to identify near twins that have somewhat similar degrees.

We stick further with the intra-set pairing heuristic based on the total degree of the vertices since it produces better results.

4.1.2 Random walk

As an intuition for the next heuristic, we make the following observation:

Observation 4.1. *Two vertices v and u can have common neighbors if and only if they are within the second neighborhood of each other*

Compared to the degree-based heuristic, where we choose and pair vertices based on the size of the neighborhood, now we want to ensure that our candidates for a contraction would have some common neighbors, which wasn't always the case with the previous strategy that makes an effort to take vertices with the same degrees but it doesn't say much about common neighbors. The fact that vertices have some common neighbors might hint that they are located in one area of the graph and, as a result, it might be more beneficial to contract them, even though this is not always the case we will see in the paragraph about 2-Neighborhood.

In this heuristic, we commence with an empty contraction sequence, S. During each iteration, until the graph is reduced to a single vertex, we initially pick the first η vertices with the lowest degree (or red degree) as candidates. We employ both strategies as in the previous heuristic for the following reasons. In the previous approach, the heuristic based on the red degree performed worse compared to the degree one since it didn't account for the total degree difference within a vertex pair. Here, however, instead of finding pairs within this set, for each candidate vertex v_1, we perform a random walk: randomly deciding to take either one or two steps to reach a vertex v_2, moving to neighboring

vertices. We then calculate a score for contracting the pair (v_1, v_2) using a predefined scoring function. Among all the pairs obtained from these random walks, we identify the pair with the minimum score as the best pair for contraction, add it to the contraction sequence S, and update the graph by contracting this pair. This process iterates until the graph is contracted down to a single vertex, at which point the contraction sequence S is returned as the output. Here is the pseudo-code:

Algorithm 1 FindRandomWalkContraction

Input: $G = (V, E, R)$
Output: Contraction sequence $S = ((v_1, u_1)...(v_{n-1}, u_{n-1}))$

```
function FindRandomWalkContraction(G)
    S ← ∅
    while |V| > 1 do
        candidates ← first η vertices in V with the lowest red degree
        scores ← {}
        for v1 in candidates do
            v2 ← randomWalk(v1)
            scores[(v1, v2)] = score(v1, v2)
        end for
        best_pair ← pair (v, u) in scores with minimum score(v, u)
        S ← S ∪ {best_pair}
        Contract best_pair
    end while
    return S
end function

procedure randomWalk(v)
    neighbors ← getNeighbors(v)
    if flipCoin() = HEADS then
        return chooseRandom(neighbors)
    else
        v' ← chooseRandom(neighbors)
        neighbors' ← getNeighbors(v')
        return chooseRandom(neighbors')
    end if
end procedure
```

For the previous intra-set pairing heuristic, we selected $\eta = 20$ and evaluated 210 vertex pairs in each iteration. To maintain a comparable number of pairs with this approach, we opt for 10 random walks for every vertex taken instead of just one single random walk.

In contrast to the previous heuristic, where a degree-based approach predominantly yielded better outcomes than the one based on the red degree, the results from this heuristic did not exhibit a significant difference. As expected, the red degree random walk performed better, primarily because, unlike before, the neighborhood size factor

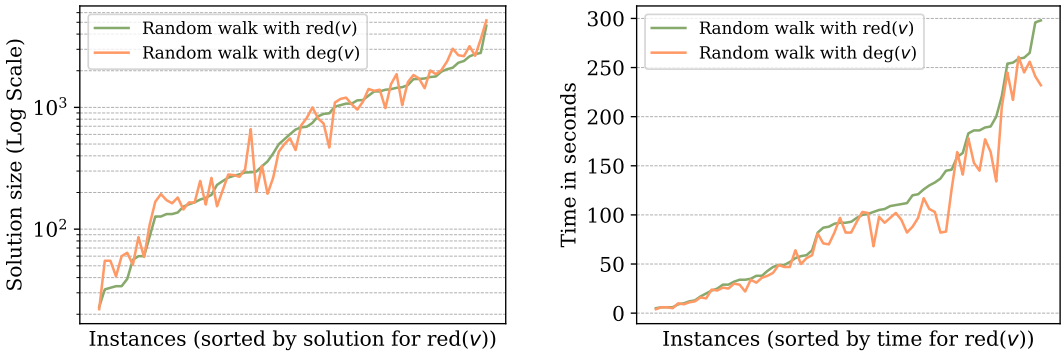
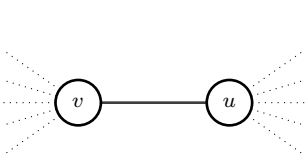


Figure 4.3: Comparison of random-walk heuristics based on low red degree or degree vertices. On the left, the solution sizes are displayed on a logarithmic scale, and on the right, the execution times.

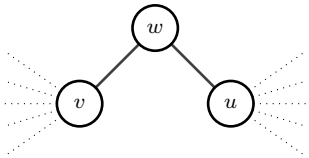
was not a determining element for either approach in this heuristic. It solved nearly twice as many instances successfully as the degree-based one, with a count of 44 versus 24. In terms of time, there isn't a substantial difference noted.

2-Neighboorhood problem Ideally, for any given vertex v, we would examine its entire first and second neighborhoods to identify the most suitable candidate for contraction. However, even for graphs with 10^3 vertices and 10^5 edges, this approach becomes computationally intensive. This is particularly concerning given our use of a while loop, which inherently carries a complexity of $\mathcal{O}(V)$.

Unfortunately, the random walk heuristic still does not guarantee the similarity of vertices. Consider the scenario in Figure 4.4a, a subplot of Figure 4.4. Even though v and u are adjacent, they lack common neighbors. Contracting such a vertex pair would significantly increase the contraction width, turning all edges from v and u red. A similar challenge arises in Figure 4.4b, where v and u might share just a single common neighbor, leading to a comparable situation. In the subsequent Subsection 4.1.3, we explore methods to optimize the random walk heuristic to minimize these adverse scenarios.

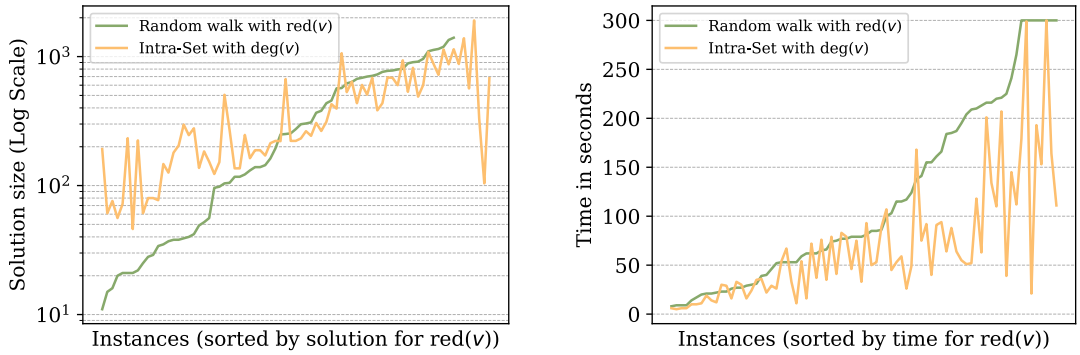


(a) Direct neighbors without common neighbors.



(b) Two-hop neighbors with a single common neighbor.

Figure 4.4: Illustrations of direct and two-hop neighbor relationships in a graph.



(a) Comparative performance of heuristics based on solution quality.

(b) Comparative performance of heuristics based on execution time.

Figure 4.5: Comparison of random-walk and intra-set heuristics in terms of solution quality and execution time.

4.1.3 Best of both worlds

In Figure 4.5, we compared the optimal versions of both heuristics. An intriguing trend emerged: in a substantial portion of the instances, the random walk heuristic outperformed the intra-set heuristic consistently and vice versa, as seen in Figure 4.5a. Both heuristics outperformed each other on an equal number of instances, with each solving 39 instances more effectively. Notably, of the 39 instances that the intra-set heuristic managed to solve, 7 could not be resolved by the random walk heuristic within the set time limit. The trend is further illustrated in Figure 4.5b: the random walk heuristic consistently requires significantly more time to solve an instance compared to the intra-set heuristic.

Upon closely examining the specific instances and their attributes, we noted a consistent trend: the random walk heuristic exhibited better performance on instances with low degree deviation. To understand why this behavior occurs, consider the following:

1. Both the random walk and intra-set heuristics choose vertices based on their degree. However, their approaches to utilizing this degree information differ:

   - **Random Walk Heuristic:** While it selects vertices based on degree, it doesn't necessarily seek pairs within this selected set. Therefore, the performance of the random walk heuristic depends on how strongly differ degrees of the vertices: high degree deviation means a wider range of vertices. This means it often misses the opportunity to leverage the fact that the vertices within the initially taken set have comparable degrees, potentially leading to suboptimal contraction sequences.
   - **Intra-Set Heuristic:** This heuristic not only selects vertices based on degree but also actively searches for pairs within this set. By doing so, it capitalizes on the inherent similarity in degrees, which can result in lower contraction width.

	Instances	V	E	Mean Degree	Avg Deviation	Intra-Set	Random Walk
Intra-Set	heuristic_006.gr	1500	5.69×10^5	758	193	**871**	1346
	heuristic_007.gr	1500	8.47×10^5	1129	96	**601**	960
	heuristic_013.gr	4087	1.85×10^5	90	57	**312**	435
	heuristic_017.gr	9072	4.65×10^5	102	107	**722**	1146
Random walk	heuristic_029.gr	14822	3.5×10^5	155	3	56	**20**
	heuristic_033.gr	18454	1.17×10^5	13	3	224	**22**
	heuristic_036.gr	20082	1.31×10^5	13	2	233	**21**
	heuristic_057.gr	32766	1.60×10^5	9	3	67	**11**

Table 4.1: Comparison of heuristics across various instances, grouped by the heuristic that performed better for each instance group. Bold values indicate better results.

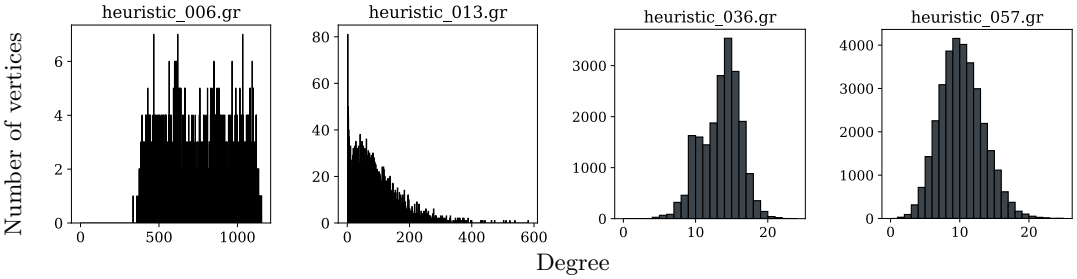
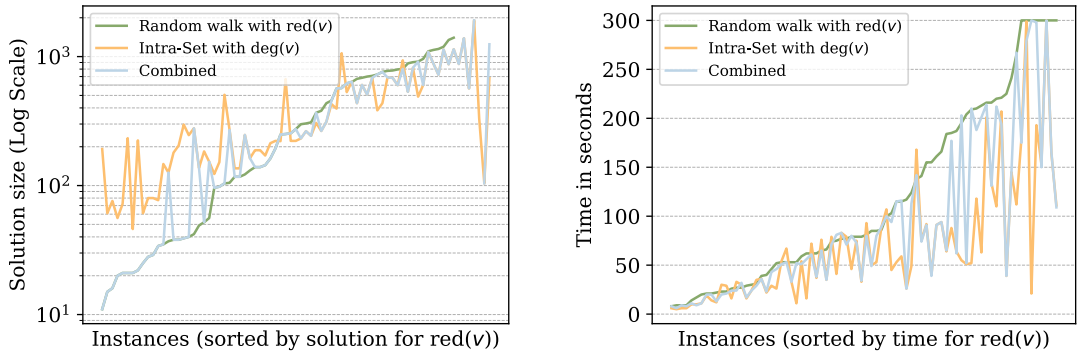


Figure 4.6: Degree distribution plots: The first two highlight graphs favoring the intra-set heuristic due to high degree deviation. The latter two showcase graphs where the random walk heuristic performs better due to minimal degree deviation.

2. Impact of low degree deviation on heuristics:

   - When the degree deviation of a graph is minimal, the random walk heuristic's chances of encountering a suboptimal vertex pair diminish significantly. Under these circumstances, the heuristic benefits from the inherent degree uniformity much like the intra-set heuristic.
   - Furthermore, the random walk heuristic has an additional advantage. Since it selects vertices from the second neighborhood, there's a higher likelihood that these vertices share common neighbors. This proximity often leads to more favorable outcomes in graphs with low degree deviations.

Through this analysis, it's evident that the degree distribution of the graph plays a crucial role in determining the effectiveness of the employed heuristic. Refer to Figure 4.6, where the degree distribution of various instances is visualized. In the first two plots, instances are presented where the intra-set heuristic outperformed, showcasing a distinct degree distribution. Contrarily, the latter two plots highlight instances where the random walk heuristic excels, and it's evident how the degree distribution differentiates from the previous. Delving deeper into the specifics, Table 4.1 further enumerates results



(a) Comparative performance of heuristics based on solution quality.

(b) Comparative performance of heuristics based on execution time.

Figure 4.7: Comparison of previous heuristics with combined approach in terms of solution quality and execution time.

across various instances. Examining this table reinforces our observation, corroborating the consistent trend seen in the degree distributions and the relative performance of the two heuristics.

Upon closely analyzing instances where the random walk heuristic outperformed its counterpart, we established a guideline for our solver: if the average degree deviation is less than 25, the random walk heuristic is employed; otherwise, the intra-set heuristic is used. As depicted in Figure 4.7, this differentiation effectively leverages the strengths of both heuristics in many scenarios. We used seed for our random generator to achieve reproducible results. From Figure 4.7a, it's evident that we successfully address both the instances with low solution sizes and those towards the higher end. Nonetheless, there are some instances where the differentiation based on average degree was suboptimal, leading the solver to select a less-than-ideal heuristic. Even so, employing this technique improves our solver's performance, allowing it to tackle 56 instances better, compared to the 39 instances by each heuristic individually.

4.2 Approximating optimal vertex pairs

This section delves into strategies for approximating optimal vertex pairs. In Section 4.2.1, we explore various score functions to evaluate vertex pair quality. Section 4.2.2 then discusses the threshold choice, determining how many vertices are considered during each iteration.

4.2.1 Score function

One of the most important aspects of getting an optimal contraction is estimating how merging a specific vertex pair will affect the width of the contraction. That's why we introduce a score function that is going to estimate the merge score for a vertex pair. The first approach to get a score for a vertex pair is to consider a symmetric difference

of the neighborhoods:

$$\text{score}(v, u) := |N(v) \cup N(u) - N(v) \cap N(u)| \tag{4.1}$$

It expresses a number of non-common neighbors for this vertex pair. Using this score function, we locally measure how many red edges would be introduced for a source vertex after the merge. The problem with this scoring method is its inaccuracy: even though we get the number of new red edges for a vertex v, it is possible that one of the neighbors' red degrees exceeds it and, indeed, increases the width of the contraction. Consider the following example:

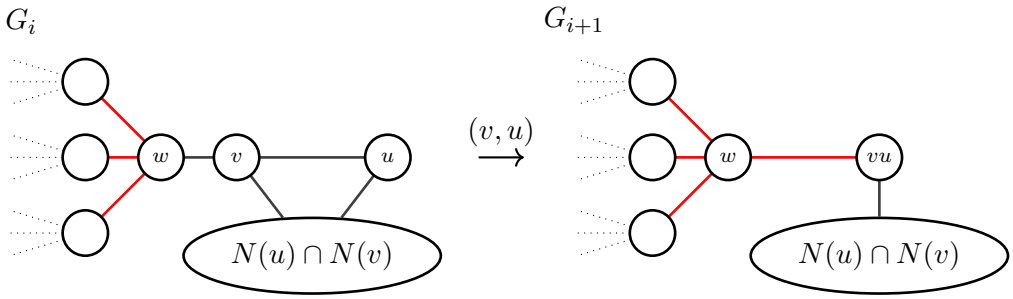


Figure 4.8: An example of a contraction step that, despite its seemingly low score, inadvertently leads to an increased contraction width.

Consider a graph state G_i on a Figure 4.8 after i contraction steps where the maximum red degree before that didn't exceed 3. Our algorithm would consider vertex pair (v, u) and compute score(v, u) that would equal 1 since v has one unique neighbor w. However, our score function doesn't say anything about the actual increase of the contraction width since it doesn't account for the increased red degree of the vertex pair neighbors like in this example - if though our score function reported a relatively good score, we still ended up increasing width of the contraction.

To avoid such cases, we try to simulate the merge, measure a new width value, and only after that decide whether this vertex pair is good enough. For this particular benchmarking, we used instances from the exact track of the PACE challenge since copying the graph turned out to be too expensive so the solver couldn't solve enough heuristic instances. Interestingly, we discover that simulating a merge and measuring the width consistently yields inferior results. Using symmetric difference, the average solution size stands at 42, whereas simulating merges results in an average of 52. The average time per instance also varies: 0.67 seconds for the symmetric difference compared to 9.53 seconds for the merge simulation. This disparity becomes even more pronounced in larger heuristic instances, which is why we opt for the symmetric difference.

To further explore different scoring strategies, we evaluate additional score functions on heuristic graphs solvable within a 300-second time limit. The additional score functions are as follows:

1. black_score$(v, u) = |N_B(v) \cup N_B(u) - N_B(v) \cap N_B(u)|$

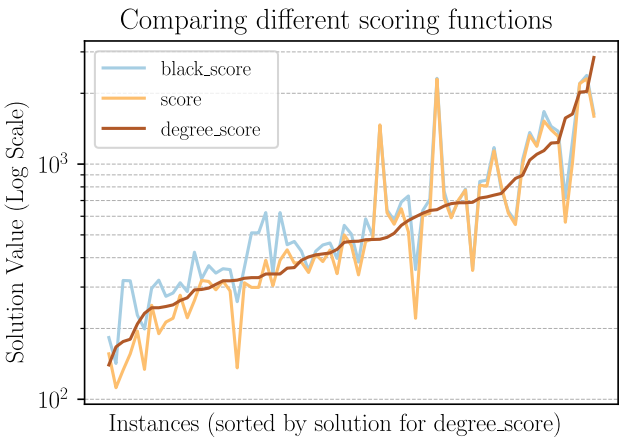


Figure 4.9: Analysis of scoring functions: the plot shows a difference in solution size on a logarithmic scale.

Metric	score	degree_score
Best solved	**36**	33
μ_{geo}	**467**	490
μ	614	**611**
σ	510	**487**

Table 4.2: Performance metrics for score function. Better values are bolded.

This function computes the symmetric difference but only considers black edges. Since we apply the score function to our base solver, where we already filter out vertices based on the number of red edges, we want to concentrate only on new red edges appearing after the merge in the hope of more accurate results.

2. $\text{degree\_score}(v, u) = \text{black\_score}(v, u) + |\deg(v) - \deg(u)| + \text{red}(v) + \text{red}(u)$

   This function adds to the black score a term based on the absolute degree difference between v and u, along with the current number of red edges associated with the vertices. The degree difference term could help prefer merging vertices with similar connectivity, potentially leading to better merge decisions. Adding a red degree of vertices serves as a penalty for vertex pairs that already have a lot of red edges.

As we can see in Figure 4.9 $\text{black\_score}(v, u)$ consistently delivers slightly worse results than our baseline $\text{score}(v, u)$ function. Function based on black score fails to identify red edges that would be transferred from existing vertices to the new vertex. Interestingly, the function $\text{degree\_score}(v, u)$ performs nearly as well as the $\text{score}(v, u)$ function in terms of the number of best-solved instances. In Table 4.2, we observe that the disparity between the geometric and arithmetic means, as well as the deviation for the score function, is minimal.

Upon closely examining the score functions, certain patterns become evident. Consider the example depicted in Figure 4.10, where two vertices, v and u, are present. Vertex v has two red edges, while vertex u has two black edges. Using our default score function, we find that $\text{score}(v, u) = 4$, since the pair has four non-common neighbors, all of which will eventually be red post-contraction. Conversely, the function that considers only the newly emerging red edges provides $\text{black\_score}(v, u) = 2$, indicating that the contraction turns two edges red. Lastly, the degree-based function yields $\text{degree\_score}(v, u) = 2 + 0 + 2 = 4$. The nuance here is that the degree score function combines both the black degree score and the red degrees, making it analogous to the default score function. However, this additional emphasis on degree difference tends to

favor vertices with similar degrees, as they are less likely to produce many red edges, and penalizes large degree disparities and this is the main difference.

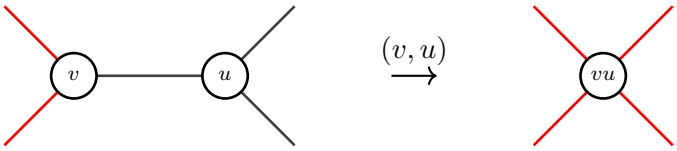


Figure 4.10: A subgraph to demonstrate different score functions.

We choose to retain our initial default score function, as its simplicity consistently delivers the best results in terms of the number of instances solved best.

4.2.2 Threshold choice

In this section, we delve into the selection of the threshold value η for the heuristics presented in Section 4.1. The number of vertices selected in each iteration is a crucial parameter for the heuristic, influencing its overall performance. Selecting too few can lead to a high contraction width, while choosing too many can result in substantial overhead, preventing the solver from completing within the allotted time. We use the combined solver as our reference point for these comparisons.

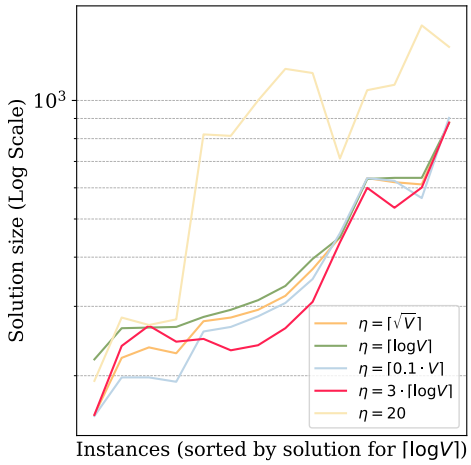
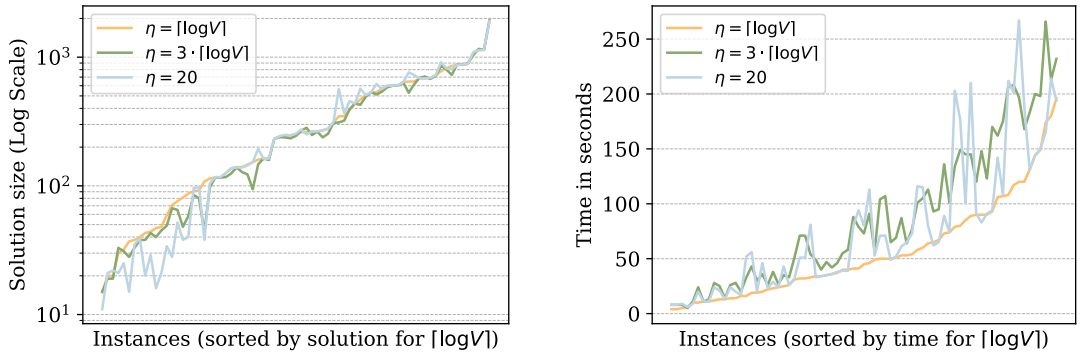


Figure 4.11: Comparison of different η parameters for the initial heuristic instances.

In Figure 4.11, we evaluate various vertex quantities, both fixed and proportional to the vertex count. This visualization focuses on a limited set of instances because the solver, for larger values like $\eta = \lceil 0.1 \cdot V \rceil$ or $\eta = \lceil \sqrt{V} \rceil$, exceeded the time constraints. Notably, a fixed value of 20 drastically underperforms compared to other values, which cluster closely. We excluded the time-intensive values from further analysis to maintain our goal of a swift heuristic.

Consider Figure 4.12, in which we compare a fixed threshold of 20 vertices against the logarithm of the vertex count and its augmented variant with a multiplication constant. Our overarching aim is to incorporate a dynamic threshold parameter. The underlying logic is that as the number of vertices in the graph escalates, pinpointing an optimal vertex pair becomes increasingly arduous. However, after numerous iterations, and especially with the scoring table introduced in Section 5.2, there's an advantage to evaluating



(a) Comparative performance based on solution quality.

(b) Comparative performance based on execution time.

Figure 4.12: Comparison of the number of vertices pro iteration in terms of solution quality and execution time.

fewer pairs. This strategy can expedite the solver's process, especially since a plethora of pairs have already been assessed.

Yet, as shown in Figure 4.12a, the differences between solution sizes are slight. In certain scenarios, particularly those with smaller solution sizes, the fixed threshold of 20 shows better performance. However, from a quantitative standpoint, the solver with $\eta = 3 \cdot \lceil \log V \rceil$ performs better in 42 instances, in contrast to the 29 instances where $\eta = 20$ prevails. Even though the results are somewhat similar, the determinative metric is runtime. As depicted in Figure 4.12b, the logarithmic approach consistently outstrips the other two methods. Both the fixed threshold and the logarithmic variant with an augmenting coefficient take more time on more substantial instances, nearly exhausting the given time limit.

For our dataset, with a maximum vertex set size of approximately $3.5 \cdot 10^5$ vertices, the logarithmic approach remains feasible, selecting around 45 vertices in the initial iterations. Nonetheless, this method's scalability is tied to the vertex set size. Consequently, in larger instances, a fixed value of 20 vertices will inevitably outpace the logarithmic approach in terms of runtime.

Chapter 5

Optimizing Twin-Width

This chapter outlines techniques for optimizing the computation of a graph's twin-width. In Section 5.1, we discuss preprocessing the graph to eliminate certain vertices. Section 5.2 discusses caching scores to avoid redundant computations. In Section 5.3, we focus on locating vertex pairs within bipartite graphs. Section 5.4 introduces a method for handling dense graphs.

5.1 Almost reduction rules

In many graph problems, kernelization techniques can be employed to obtain a reduced instance of the problem, known as a kernel, which in turn lessens the computational resources required for finding a solution. However, the computation of the twin-width of a graph doesn't readily lend itself to such kernelization, particularly when aiming for a polynomial-size kernel. Despite this, it's important to explore if practical compression techniques without any theoretical guarantees could still offer some level of computational relief in this context.

In Subsection 5.1.1, we eliminate twin vertices and evaluate performance, and in Subsection 5.1.2, we discuss different approaches to merging one-degree vertices.

5.1.1 Twin Rule

One of the most straightforward preprocessing techniques that we can apply is eliminating twins. This is crucial since contracting such vertex pairs doesn't increase the width of the contraction sequence and allows us to reduce the size of the instance we are trying to solve. We use an algorithm called *partition refinement* presented by Habib, Paul, and Viennot [HPV98] to identify twins in the graph. The algorithm returns a partitioned vertex set, where each subset with more than one element corresponds to twin vertices.

Each algorithm iteration processes a vertex v, extracting its neighborhood in the graph. The refinement operation is carried out for each existing partition in set P. The operation involves calculating both the intersection and difference between the current partition and the set of neighbors around the vertex. If either result of the intersection or difference operation is non-empty, it is added to an updated set of partitions. This

Algorithm 2 Partition Refinement

Input: $G = (V, E)$
Output: Partitioned vertex set P

```
1: procedure PartitionRefinement(G)
2:     P ← {V}
3:     for v in V(G) do
4:         P ← Refine(P, N(v))
5:     end for
6:     return P
7: end procedure

8: procedure Refine(P, X)
9:     P_new ← ∅
10:    for S in P do
11:        P_new ← P_new ∪ {S \ X, S ∩ X}
12:    end for
13:    return P_new
14: end procedure
```

(a) True twin vertices v and u.

(b) False twin vertices v and u.

Figure 5.1: Illustrations of different types of twin neighbors.

process effectively refines each partition in P by distinguishing between vertices based on their graph neighborhoods.

Following the refinement step for vertex v, the set P is updated with the newly computed partitions, and the algorithm proceeds to the next vertex until all vertices have been processed. After completing these iterations, partitions containing more than one vertex represent sets of twin vertices in the graph. Each set in the final partition P encompasses vertices that could not be distinguished from each other through the iterative refinement process, classifying them as twin vertices.

The complexity of executing this algorithm stands at $\mathcal{O}(V + E)$. However, this algorithm presents a subtle challenge. Graphs can have two distinct categories of twin vertices: true twins and false twins. As depicted in Figure 5.1a, true twins share the same open neighborhood, while false twins have a mutual closed neighborhood, as shown in Figure 5.1b. When executing our algorithm, we must choose between refining our vertex set using either the closed or open neighborhood. Given that our process permits the merging of both types of twin vertices without any penalty, our objective is to remove all twin vertices. This requires running the algorithm twice - once for true twins and once for

false twins. However, even a linear-time algorithm, when scaled with a constant, proves computationally expensive. Consequently, the solver consumes the entire allocated time merely to eliminate the twin vertices on instances from the heuristic track.

However, the problem is less pronounced on smaller graphs. We tested our solver on instances from the exact track of the PACE challenge, which are notably smaller than the heuristic ones.

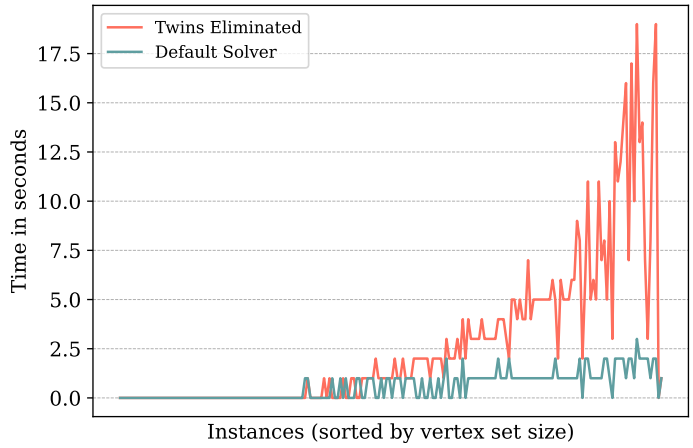


Figure 5.2: Time performance of twin rule compared to the default solver.

Metric	Twins	Default
$\mu_{\text{geo},tww}$	**17.8**	18.4
μ_{tww}	**30**	31.1
σ_{tww}	**24**	26
Time (total)	537	**121**
Time (avg.)	2.685	**0.605**

Table 5.1: Performance metrics for twin rule. Better values are bolded.

In Figure 5.2, we observe that the twin rule considerably impacts the time performance. While this is manageable for exact instances, Table 5.1 shows that the total time for the entire benchmark run has increased by over fourfold. However, the difference in solution sizes remains relatively minor. We observe only a slight difference in the solutions on heuristic instances. However, partition refinement significantly slows down the solver, limiting it to solving only the initial smaller instances.

Nevertheless, eliminating twins is a crucial step, especially when calculating the twin-width of a graph. An alternative to partition refinement is *modular decomposition*. Not only does it allow for the elimination of twins, as they end up in the same module after decomposing, but it also provides a systematic way to sort and identify further vertex pairs based on this information. The efficacy of this technique is further corroborated by Yosuke Mizutani and Sullivan [YMS23] in their winner solver for the exact track of the PACE challenge, which employs it among other strategies, underscoring its potential as a powerful tool for graph restructuring.

5.1.2 One Degree Rule

Many graph problems allow reducing instance size by eliminating one-degree vertices and including or excluding them from the solution, e.g., always taking a neighbor of a one-degree vertex in the Vertex Cover problem. In twin-width, this is a bit more complicated.

Suppose we want to contract a one-degree vertex with its direct neighbor. Then, the width of the graph would be automatically increased by the size of the neighborhood of this neighbor as shown in the Figure 5.3

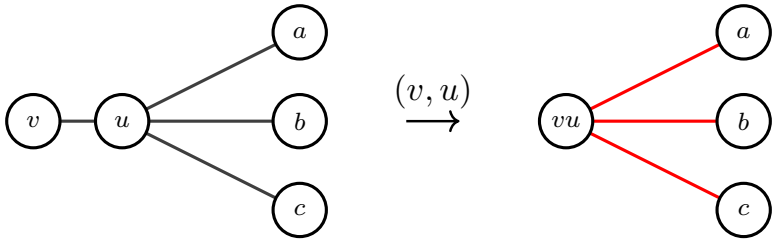


Figure 5.3: Contracting one-degree vertex with its neighbor

We experiment with different one-degree rule strategies to determine whether they could consistently cause reduced contraction width. For instance, we decide to try out the following approaches:

1. Initial Strategy

- At the beginning, identify all one-degree vertices
- Pair and contract these one-degree vertices randomly with each other. Only consider those vertices that had one degree in the initial state of the graph, even though after multiple contractions some of them are not one degree anymore

Intuitively, this strategy must fail since, after all contractions, we have accumulated all red edges at the single vertex, which is the opposite of our goal: distribute them among all vertices to minimize the width. However, there is still a hope that if the number of one-degree vertices is low enough then it wouldn't exceed the actual twin-width of the graph, allowing to still be under the current width at each contraction step.

2. Half Strategy

- At each iteration step, identify all one degree vertices
- Pair and contract these dynamic one-degree vertices randomly with other dynamic one-degree vertices. The vertices should have one degree at the moment of contraction.

This strategy is practically a lightweight version of the former one with the difference that we don't accumulate all edges at one single vertex but just stop after the first iteration. Given $|V_{d_1}|$ one degree vertices in the graph we basically perform $|V_{d_1}|$ or $|V_{d_1}| - 1$ contractions, depending on $|V_{d_1}|$ being odd or even number. In this version, we cannot guarantee an upper bound for the width after those contractions. This is because there might be instances where a vertex has numerous one-degree vertices that weren't contracted with their twins. Consequently, this vertex would accumulate all the red edges, making it impossible to guarantee a specific width.

However, if we eliminate all twins before performing this one-degree rule strategy, we can guarantee contraction width on G_i after i contraction steps to be 2. This is because both vertices from the contracted vertex pair have different neighbors and, as a result,

a vertex that remains after contraction would always have only two red edges, one with its initial neighbor and one new red edge with a neighbor of the eliminated vertex.

3. Threshold-Based Strategy

- At the beginning, identify all one-degree vertices
- For each one-degree vertex, check its neighboring vertex's degree. If the neighbor's degree is below a specified threshold σ, contract the one-degree vertex with its neighbor

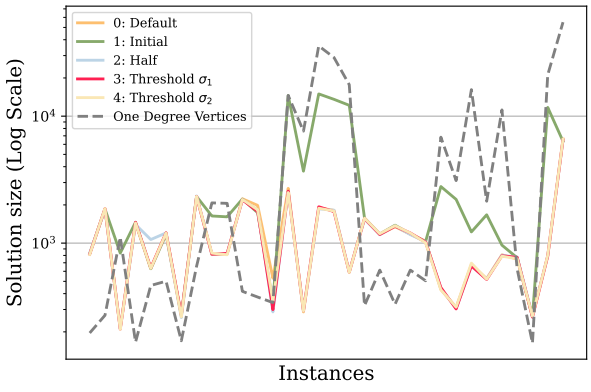
We try to play around with the threshold itself to identify what is the optimal degree of one-degree vertex's neighbor. On the one hand, we want to keep it low since we want to minimize the red edges of the neighbor. On the other hand, if we have too high value, we automatically skew results since the vertex out of all neighbors with maximum degree m out of all candidates would cause the width to be at least m since all its edges would become red. Similar to the first strategy, we were hoping to pick such a threshold so that after contracting one-degree vertices, the resulting width of the contraction would still be far below its final value, so we wouldn't affect it too much. With S being a one-degree vertex set, we choose two different thresholds:

$$\sigma_1 = \sqrt{|S|} \tag{5.1}$$

$$\sigma_2 = \frac{\sum_{v \in S, u \in N(v)} |N(u)|}{|S|} \tag{5.2}$$

The first threshold is just a square root of the one-degree vertex set size. The second one determines the average degree of neighbors of one-degree vertices.

Benchmarking one degree strategies To test our strategies, we choose all instances that contain at least 100 one-degree vertices to evaluate our strategies. We pick such instances that can be solved by our base solver described in the previous section. Here are the results:



#	Strategy	Best Instances
0	Default	6
1	Initial	5
2	Half	10
3	Threshold σ_1	4
4	Threshold σ_2	7

Figure 5.4: Dotted grey line represents the number of one-degree vertices in a given instance.

As we can see from benchmarking, results are pretty ambiguous. Even though the half strategy solved twice as many instances as others, it's still hard to say whether it plays a significant role in finding the optimal contraction sequence. From Figure 5.4 we can see that almost all strategies are not so far from each other, and results don't differ much. As expected, the initial strategy turned out to be the worst since the graphs contained relatively high number of one-degree vertices. However, even in such a setting it was able to beat other strategies in 4 instances but had terrible results in the others. In some cases where the strategies' solution is almost equal to the number of one-degree vertices, we suggest that the width of the contraction was significantly affected by this pre-processing and wasn't near to optimal.

To make sure, we also compare the half strategy with the default solver on exact instances with a ratio of one-degree vertices to all vertices over 0.2: half strategy again performed better with 10 instances won versus 7 instances for the default solver. However, in most of the cases, the difference in solution wasn't significant, so one could reason that it is still within the deviation of different contraction paths.

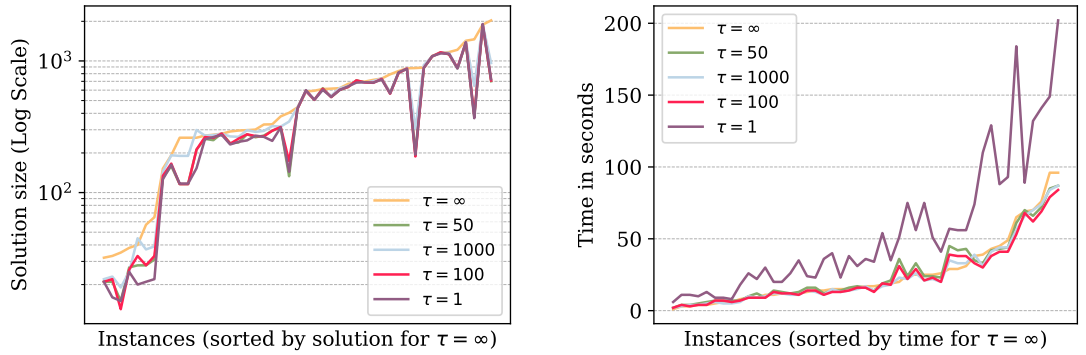
5.2 Caching scores

To address the overhead caused by computing the score(v, u) function mentioned in Section 4.2.1, we introduce a score caching mechanism. The principle is intuitive: once the score for a particular vertex pair has been computed, it is stored within a hash map. In subsequent iterations, the pre-calculated value is fetched and utilized rather than recomputing the score. The main problem is the dynamic nature of the graph since vertex contractions can render these cached scores obsolete or inaccurate. However, post-contraction, only the scores of the neighboring vertices of the contracted pair are subject to change, as we demonstrated in Figure 4.8. Also, we make following observation:

Observation 5.1. *Consider a vertex w in the graph state G_i where red$(w) = n$. If (v, u) is the next vertex pair in the contraction sequence and $w \in N(v) \cup N(u)$, then in the subsequent graph state G_{i+1}, the value of red(w) will be at most $n + 1$.*

After contracting a vertex pair (v, u), the value of red(v) may increase substantially. This is because v remains as the residual vertex, accumulating all new red edges. However, its neighbors—both existing and newly introduced—experience only minor changes. The rationale for adopting score caching stems from a crucial observation: the magnitude of change in the red degree post each contraction is marginal. Hence, the benefit derived from recalculating the exact scores instead of utilizing slightly outdated cached values is negligible. This trade-off between accuracy and computational efficiency proves favorable in our context.

However, it's important to acknowledge that relying solely on cached scores can lead to considerable deviations from true values over extended periods and after many contractions. To mitigate this, we propose the introduction of a reset parameter τ. If the difference between the current iteration and the last update for a specific vertex pair surpasses this threshold, the score table with all scores calculated so far is cleared. This strategy should ensure a judicious balance between accuracy and performance, enabling the heuristic to handle large graphs more efficiently.



(a) Comparative performance based on solution quality.

(b) Comparative performance based on execution time.

Figure 5.5: Comparison of different reset parameters τ for clearing score table.

As observed in Figure 5.5, the results provide intriguing insights. In Figure 5.5a, we note that the solution sizes remain relatively consistent for most reset parameters, except for a significant divergence at $\tau = \infty$. This makes sense since when the score for a vertex pair is computed once and never recomputed, there's a decline in performance, albeit not drastically. Turning our attention to Figure 5.5b, it becomes evident that recalculating the score every iteration, particularly when $\tau = 1$, incurs a substantially longer computation time compared to other parameters. Interestingly, for values ranging from $\tau = 50$ to $\tau = \infty$, the results cluster within a similar time frame, making it challenging to designate a clear best performer.

The results presented offer two main observations. Firstly, they provide partial validation for our initial hypothesis: once a score for a pair has been computed and potentially merged, it does not significantly influence the scores of other vertex pairs. Consequently, even if the score for a vertex pair was determined several iterations ago, it remains a viable choice as long as the score remains relatively low. Secondly, examining the time performance in Figure 5.5b reveals that recalculating scores less frequently does not yield a substantial improvement in our performance. This suggests that while we can afford to recalculate scores periodically, it is not essential to do so in every iteration.

5.3 Bipartite graphs

Bipartite graphs can be split into two partitions where vertices from one don't connect with others in the same partition. After figuring out that a graph is bipartite, we split it into two parts.

Establishing bipartiteness is accomplished using the breadth-first search. Each vertex is assigned a color, and the traversal ensures adjacent vertices are oppositely colored. If, at any point, adjacent vertices share a color, the graph is deemed non-bipartite. By the end of this process, the coloring effectively delineates the partitions.

With the partitions identified, the next phase targets optimal vertex pairing within each partition. Given that vertices from distinct partitions don't share edges, it's in-

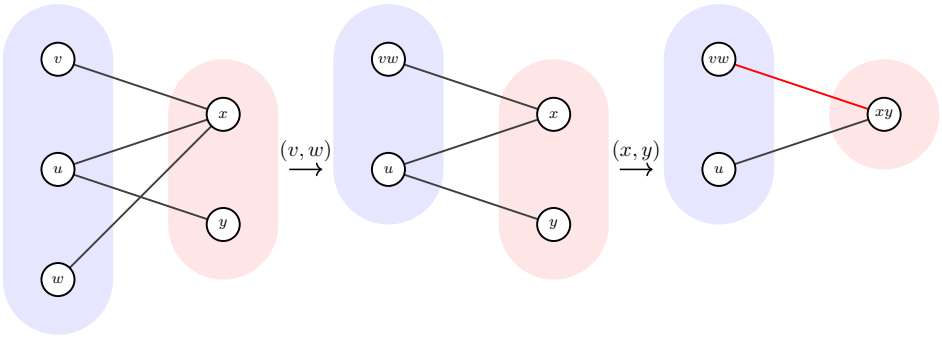


Figure 5.6: Contracting vertices in a bipartite graph.

efficient to consider vertices. That's why we consider only vertex pairs within each partition. After arriving at optimal pairs for both partitions, the final step merges the representative vertices from each partition.

Since we want to have an equal comparison of a base solver with this bipartite modification, we must decide which vertices we consider as candidates. We took the first 20 vertices with the lowest red degree in a base version. To be consistent, we decided to take 10 vertices from each partition correspondingly in an attempt to find the best possible vertex pair for contraction.

However, no improvements were observed after multiple runs on different test instances. After examining specific instances, we concluded that most of the connected components that are bipartite are much smaller in size compared to the non-bipartite ones. Also examining contraction sequences we see that our base solver never merges vertices from different partitions anyway since such pairs always demonstrate a bad score compared to other candidates within the same partition. Playing around with different numbers of vertices pro partition, like considering only 5 vertices within the first and 15 within the second partition, skewed results even more as expected; the solver failed to identify a better pair from another smaller partition since it contained fewer candidates for consideration.

5.4 Density trick

Bonnet et al. [Bon+20] proved that bounded twin-width is preserved under FO interpretations and transductions allowing complementing a graph. Consequently, devoting computational power to dense graphs becomes inefficient, especially when an optimal twin-width can be more effectively obtained from the graph's complement.

Determining Graph Density Before processing a graph, it is imperative to gauge its density. The density of a graph G with V vertices and E edges is determined by the formula:

$$\text{density}(G) = \frac{2 \cdot E}{V \cdot (V - 1)}$$

If the computed density surpasses 0.5, our system opts to work with the graph's complement.

Constructing the Complement Graph The procedure to derive the complement of a graph is uncomplicated. Initially, a complete graph is conceptualized, incorporating every conceivable edge. However, a graph of merely 1000 vertices results in around $5 \cdot 10^5$ edges. We abstain from using our existing graph data structure to sidestep the inefficiency of processing such a large graph. Instead, all potential edges are systematically added to a hash set, ensuring that each edge (v, u) adheres to the ordered pair format $(min(v, u), max(v, u))$. This deterministic ordering ensures a consistent representation.

Upon reading the actual graph, the edges recognized are instantly removed from our hash set. Only once this procedure is accomplished for all edges do we proceed to structure the graph using our original data structure.

Empirical Evaluation Although our dataset comprises a modest number of graphs with high density values, the results in these instances are noteworthy.

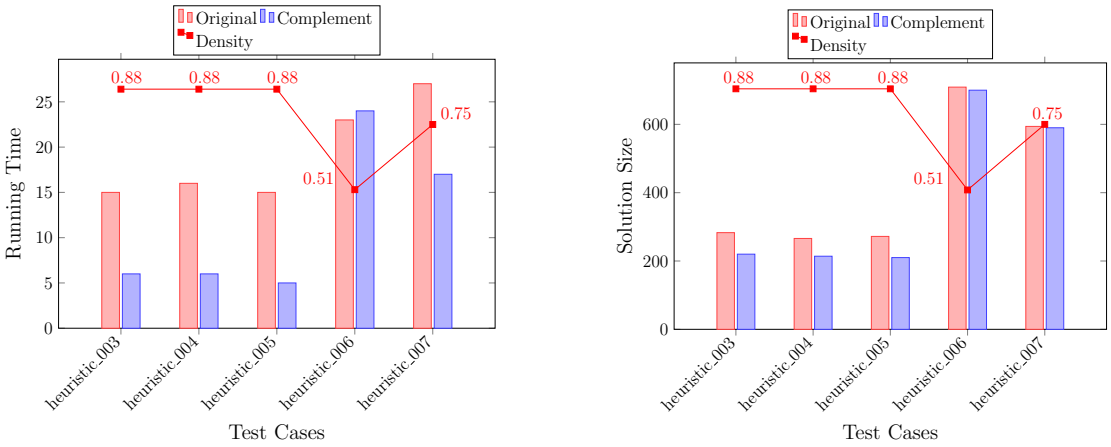


Figure 5.7: The left plot shows the time difference, while the right plot illustrates the improvement in solution size

Based on our observations, particularly dense graphs with a density of approximately 0.9 demonstrated up to a 3x acceleration in runtime. In contrast, graphs with a slightly above-average density, around 0.51, didn't show a speedup. In some cases, the runtime even increased, although this was within a certain deviation. Notably, even for such graphs, we observed an improvement in the solution size.

A pertinent question arises: Why is there a reduction in the solution size when the exact solution for both a graph and its complement should, in theory, be identical? We theorize that this phenomenon can be attributed to the nature of our heuristic. A less dense graph inherently possesses fewer edges. Our heuristic principally operates by ascertaining the count of red edges in every iterative step, thereby yielding a more optimal outcome with sparser graphs.

5.5 Twin-Width growth

Another strategy to enhance our likelihood of computing a near-optimal contraction sequence is to leverage previously identified sequences. If the solver successfully identifies a contraction sequence within a time limit, we aim to store and employ it in our exhaustive approach. This involves recording all contraction steps, the associated score for each vertex pair, and the progression of the contraction width. The rationale is straightforward: during the computation of a current sequence, if at the i-th iteration a vertex pair (v, u) elevates the contraction width beyond what was observed in the prior contraction sequence for the same pair, we discard this sequence and initiate a new search.

Nevertheless, after several iterations, it becomes evident that the majority of the newly computed sequences are eventually discarded. To understand this better, we allowed our solver to execute multiple runs, focusing on the evolution of the contraction width.

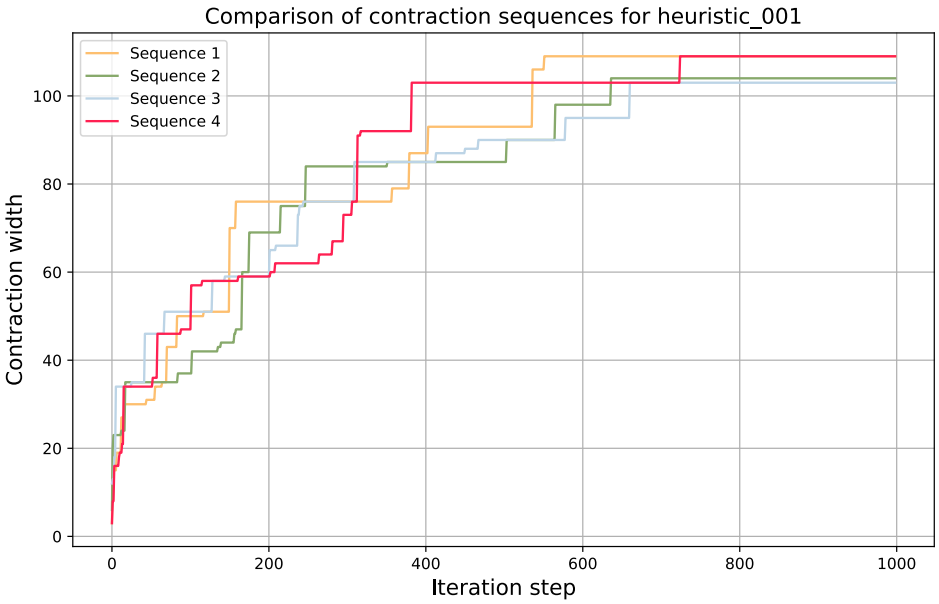


Figure 5.8: Example of different contraction sequences and growth of their width values.

As illustrated in Figure 5.8, the development of the contraction width is not direct or predictable. This underscores the challenges highlighted in Section 3.4. For example, after approximately 300 iterations, sequence 4 appears to offer the best width. Yet, at some subsequent point, the trajectory deviates, and this sequence concludes with the least favorable width. An alternative approach might be to consider only the final contraction width of previously computed sequences as an upper bound, rather than intermediate values. While this might offer a more consistent measure, it lacks the efficiency of early pruning suboptimal branches.

Chapter 6

Conclusion

In this thesis, we have explored the twin-width problem using a selection of heuristic strategies. Our primary heuristics centered on vertex attributes, such as degrees and neighborhood similarities. The intra-set pairing method grouped vertices with analogous degrees, aiming for minimized post-contraction red edges. The random walk heuristic was devised based on the locality of optimal vertex pairs. Our findings indicate that the success of a heuristic is often contingent on the graph's inherent properties.

The outcomes of our heuristics were instructive. While the density trick and twin rules led to better solution sizes, the one-degree rule and considerations of bipartite graphs had limited impact. Furthermore, our experiments showcased that expanding heuristic parameters, like the vertex set size or score recalibration interval, only improved results up to a certain threshold, after which there was no noticeable benefit. This demonstrates the natural limitation of current heuristic approaches and the need for more sophisticated strategies.

For future research directions, it would be worth investigating non-trivial reduction rules and considering alternative vertex similarity measures, such as the Sørensen–Dice or Jaccard coefficients. Additionally, the non-linear progression of contraction width remains an area worth more detailed study.

In conclusion, our heuristic methods have offered some insights into the twin-width problem. The solver and benchmarking tools developed in this research may be useful for subsequent studies in this area. This thesis provides a starting point for ongoing research and adjustments in twin-width computation.

Literature

[ALP23] Anselm Haak Frank Kammer Johannes Meintrup Ulrich Meyer Alexander Leonhardt Holger Dell and Manuel Penschuck. *Exact (GUTHMI) and Heuristic (GUTHM) Computation of Contraction Sequences with Minimum Twin-Width*. 2023 (cit. on p. 11).

[BBD21] Pierre Bergé, Édouard Bonnet, and Hugues Déprés. *Deciding twin-width at most 4 is NP-complete*. In: *CoRR* abs/2112.08953 (2021). arXiv: 2112.08953 (cit. on p. 17).

[BD23] Édouard Bonnet and Julien Duron. *RedAlert - Heuristic Track*. 2023 (cit. on p. 11).

[BH21] Jakub Balabán and Petr Hlinený. *Twin-Width is Linear in the Poset Width*. In: *CoRR* abs/2106.15337 (2021). arXiv: 2106.15337 (cit. on p. 9).

[Bon+20] Édouard Bonnet, Eun Jung Kim, Stéphan Thomassé, and Rémi Watrigant. *Twin-width I: tractable FO model checking*. In: *CoRR* abs/2004.14789 (2020). arXiv: 2004.14789 (cit. on pp. 9, 42).

[Bon+21] Édouard Bonnet, Eun Jung Kim, Amadeus Reinald, Stéphan Thomassé, and Rémi Watrigant. *Twin-width and polynomial kernels*. In: *CoRR* abs/2107.02882 (2021). arXiv: 2107.02882 (cit. on p. 9).

[CMR00] B. Courcelle, J. A. Makowsky, and U. Rotics. *Linear Time Solvable Optimization Problems on Graphs of Bounded Clique-Width*. In: *Theory of Computing Systems* 33.2 (2000), pp. 125–150 (cit. on p. 9).

[EAV23] Pål Grønas Drange Kenneth Langedal Farhad Vadiee Emmanuel Arrighi Petra Wolf and Martin Vatshelle. *Zygosity - Solver for the PACE 2023 Challenge on Twin-Width - Heuristic Track*. 2023 (cit. on p. 11).

[Gaj+15] Jakub Gajarský, Petr Hlinený, Daniel Lokshtanov, Jan Obdrzálek, Sebastian Ordyniak, M. S. Ramanujan, and Saket Saurabh. *FO Model Checking on Posets of Bounded Width*. In: *CoRR* abs/1504.04115 (2015). arXiv: 1504.04115 (cit. on p. 9).

[GKS14] Martin Grohe, Stephan Kreutzer, and Sebastian Siebertz. *Deciding first-order properties of nowhere dense graphs*. 2014. arXiv: 1311.3899 [cs.LO] (cit. on p. 9).

[GSV84] Yuri Gurevich, Larry Stockmeyer, and Uzi Vishkin. *Solving NP-Hard Problems on Graphs That Are Almost Trees and an Application to Facility Location Problems*. In: *J. ACM* 31 (July 1984), pp. 459–473 (cit. on p. 9).

[HJ23] Petr Hliněný and Jan Jedelský. *Twin-Width of Planar Graphs Is at Most 8, and at Most 6 When Bipartite Planar*. In: *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*. Ed. by Kousha Etessami, Uriel Feige, and Gabriele Puppis. Vol. 261. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 75:1–75:18 (cit. on p. 11).

[HPV98] Michel Habib, Christophe Paul, and Laurent Viennot. *A Synthesis on Partition Refinement: A Useful Routine for Strings, Graphs, Boolean Matrices and Automata.* In: Jan. 1998, pp. 25–38 (cit. on p. 35).

[Kos23] Denis Koshelev. *Heuristics for Twin-Width: Solver and Benchmarking Suite*. GitHub Repository. 2023 (cit. on p. 11).

[See96] Detlef Seese. *Linear time computable problems and first-order descriptions*. In: *Mathematical Structures in Computer Science* 6.6 (1996), 505–526 (cit. on p. 9).

[SS21] André Schidler and Stefan Szeider. *A SAT Approach to Twin-Width*. In: *CoRR* abs/2110.06146 (2021). arXiv: 2110.06146 (cit. on p. 9).

[YMS23] David Dursteler Yosuke Mizutani and Blair D. Sullivan. *Hydra Prime: Modular Decomposition and Bounds for Twin-Width Contraction*. 2023 (cit. on pp. 11, 37).