Exploring Edge Orientation Algorithms in the Static and Dynamic Case

Fabian Walliser

May 13, 2024

4173801

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University

Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisors:
Ernestine Großmann
Henrik Reinstädtler

Acknowledgments

I would like to express my gratitude to Prof. Dr. Christian Schulz for the opportunity to work on this fascinating topic under his supervision. His encouragement and flexibility greatly enriched this journey, allowing me the creative freedom to incorporate numerous personal ideas into this project. I would also like to thank Ernestine Großmann, who supported me from the very beginning and was always on hand with advice. A special note of appreciation goes to Henrik Reinstädtler as well for the considerable effort and time he devoted to helping me.

Furthermore, I would like to extend my gratitude to my family, who have consistently provided me with the best possible support throughout my entire life. Finally, I would like to give a special thanks to all of my friends, whose endless encouragement and love always carried me. I want each person to feel individually addressed.

Hiermit versichere ich, dass ich die Arbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich aus fremden Werken Übernommenes als fremd kenntlich gemacht habe. Ferner versichere ich, dass die übermittelte elektronische Version in Inhalt und Wortlaut mit der gedruckten Version meiner Arbeit vollständig übereinstimmt. Ich bin einverstanden, dass diese elektronische Fassung universitätsintern anhand einer Plagiatssoftware auf Plagiate überprüft wird.

Heidelberg, May 13, 2024

Fabian Walliser

Abstract

The edge orientation problem asks for assigning directions to the edges of an undirected graph G to form a directed graph $\mathcal{O}$, with the objective of minimizing the maximum outdegree within $\mathcal{O}$. This problem finds applications across various domains and can be solved optimally in polynomial time. However, due to the scarcity of linear-time approaches in the current literature, we explore a number of strategies for approximating the solution. Furthermore, the dynamic variant of this problem involves maintaining edge orientations through a sequence of update operations. To address this, we present an approach based on concepts of two existing algorithms, employing multiple iterations of a breadth-first search for descending nodes. Moreover, we propose a novel algorithm for optimally solving the dynamic edge orientation problem through invariants, supported by a proof. Building upon an existing optimal static algorithm, this approach operates by manipulating improving paths. We conduct experimental evaluations of our implementations against state-of-the-art algorithms. This demonstrates that even the best non-optimal competitor, utilizing a basic breadth-first search, is on average only 75.4% faster than our dynamic optimal algorithm while achieving 92% of the best solutions.

Contents

Abstract v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Our Contribution . . . 2
- 1.3 Structure . . . 3

2 Fundamentals 5
- 2.1 General Definitions . . . 5

3 Related Work 9
- 3.1 Static Edge Orientation Algorithms . . . 9
- 3.2 Fully Dynamic Edge Orientation Algorithms . . . 10

4 Algorithms 11
- 4.1 Underlying Data Structures: An Overview . . . 11
- 4.2 Static Algorithms . . . 13
- 4.3 Dynamic Algorithms . . . 28

5 Experimental Evaluation 43
- 5.1 Methodology . . . 43
- 5.2 Static Results . . . 47
- 5.3 Fully Dynamic Results . . . 53
- 5.4 Overall Comparison . . . 56

6 Discussion 63
- 6.1 Conclusion . . . 63
- 6.2 Future Work . . . 63

A Appendix 65
- A.1 Further Results . . . 65
- A.2 Instances . . . 69

A.3 Implementation Details 72

Abstract (German) **77**

Bibliography **79**

CHAPTER 1

Introduction

1.1 Motivation

Graph-based data modeling provides the basis for numerous applications in our daily lives, ranging from social networks and search engines to telecommunication and bioinformatics for simulating complex systems like the human brain. These applications often involve graphs with billions of nodes and edges, posing significant challenges in terms of data storage and processing. Since these graphs are typically sparse - meaning the number of edges is much lower than the maximal amount - efficiently managing them is an important area of computer science. A simple example of this is social networks, where despite the vast number of users, each individual typically connects to only a limited subset of the network. Accordingly, the storing of such a sparse graph with low memory requirements and yet fast adjacency queries is crucial. This also provides a foundation for more complex algorithms that operate on these data structures and are dependent on fast elementary requests.

Adjacency queries determine whether an edge $\{u, v\} \in E$ exists between two nodes u and v, returning *true* if it does and *false* otherwise, which takes constant time in the best case. Traditional data structures mostly provide trade-offs. For example, adjacency arrays require only $O(n + m)$ memory but may necessitate traversing a potentially large neighborhood for each adjacency query. In contrast, adjacency matrices are able to answer such queries in $O(1)$, but conversely also require $O(n^2)$ of memory.

A data structure, which requires linear storage space $O(n + m)$ and is able to answer adjacency queries in $O(\alpha)$ was proposed by Kannan et al. [25]. Here, α is defined as the minimum amount of forests into which a graph can be partitioned, referred to as the *arboricity*. Briefly described, each node maintains an adjacency list of its neighbors, whereas edges are only stored for exactly one of their endpoints. Since these are thus directed, a graph $\mathcal{O} = (V, A_{\mathcal{O}})$ that contains exactly (u, v) or (v, u) for each edge $\{u, v\} \in E$ is called an orientation of $G = (V, E)$. An adjacency query can now be performed simply by having both nodes search for the other in their lists, which is limited by the corresponding

list length. To ensure fast queries, the length of these lists, equivalent to the outdegree of the nodes, should therefore be bounded by a constant Δ. Minimizing Δ is the task of the edge orientation problem, which is also closely related to computing the pseudoarboricity as well as the maximum average density of graphs.

The edge orientation problem finds applications in various fields, such as storing optimal graphs [1]. For more examples and further details, we are referring the reader to Georgakopoulos and Politopoulos [21]. Given our primary focus on approximations in the static case, we are also stating an example where these excel. Specifically, they can serve as a preprocessing step for optimal algorithms, minimizing the need for resource-intensive actions, like the FastImprove method used by Reinstädtler et al. [38].

In real-world applications, graphs most often undergo changes over time, such as users following or unfollowing others in social networks. Consequently, the edge orientation problem extends to the dynamic case, where the focus shifts to maintaining a Δ-orientation over a sequence of insertions and deletions. This is used as a foundation for numerous dynamic graph algorithms [26]. For instance, it can be utilized to maintain a fully dynamic maximal matching with an amortized update time $O(\frac{\log n}{\log \log n})$ [34]. Additionally, it is valuable for reporting all maximum independent sets [15] and counting subgraphs in sparse graphs [14].

1.2 Our Contribution

While numerous polynomial-time algorithms exist for computing the optimal edge orientation [3, 28, 39], the literature offers only a few linear-time approximation approaches, one of the most renowned being a simple 2-approximation [2, 9, 21]. Therefore, a part of this thesis is devoted to developing and evaluating several strategies aimed at approximating the optimal solution. More specifically, we model strategies similar to the known 2-approximation, several approaches utilizing FOREST [31], and an algorithm that uses a lower bound obtained by the 2-approximation for increased solution quality. Furthermore, we combine two established dynamic algorithms (BFS, DescDegrees [7]) to develop a novel approach to the dynamic edge orientation problem. We also propose an optimal dynamic edge orientation algorithm based on concepts for the static problem by Venkateswaran [39]. For this algorithm, we also provide proof of correctness. Moreover, we engineer and evaluate these algorithms in comparison to each other as well as to algorithms from the current literature [7] on real-world dynamic graphs (for dynamic algorithms) as well as on real-world static graphs that have been modeled as dynamic sequences. Our results show a number of algorithms that improve on the 2-approximation, as well as strategies that fail to compete with it. Additionally, we compare the best algorithms in terms of quality with the ideal results of the dynamic optimal algorithm.

1.3 Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the definitions and notations that are used for the rest of the thesis. In Chapter 3 we delve into the existing literature, examining previous studies on the static and dynamic edge orientation problem. Next, Chapter 4 outlines the functionality of the newly proposed algorithms. It begins with a discussion of two foundational data structures and follows with a series of simple static algorithms for edge orientation approximation. More specifically, we discuss strategies analogous to the known 2-approximation [2, 9, 21], an algorithm that uses a lower bound obtained by the 2-approximation for increased solution quality and several approaches utilizing FOREST [31]. The subsequent sections introduce the Descending BFS Algorithm as our first dynamic approach, followed by an optimal algorithm for the dynamic edge orientation problem, complete with a detailed proof of correctness. In Chapter 5, we assess the performance of the proposed algorithms, comparing them against established methods. We begin by determining the best variations and parameters through a categorized evaluation of the algorithms. This is initially done separately for static and dynamic algorithms, with each category evaluated independently, and ends with an overall comparison. Chapter 6 completes this thesis by giving a conclusion from the experiments and providing an outlook on future work.

CHAPTER 2

Fundamentals

2.1 General Definitions

Directed and Undirected Graphs. Let $V = \{0, \ldots, n-1\}$ denote the set of nodes. A directed edge is defined as a tuple of two nodes (u, v) and is described as an outgoing edge of u as well as an incoming edge of v. An undirected edge is represented by $\{u, v\}$. Accordingly, we denote $A = \{(u, v) \mid u, v \in V\}$ as a set of directed edges and $E = \{\{u, v\} \mid u, v \in V\}$ as a set of undirected edges. A directed graph is a tuple (V, A) containing directed edges, whereas an undirected graph (V, E) consists of undirected edges. Let $G = (V, E)$ be an undirected graph with $E \subseteq \binom{V}{2}$. A Graph is always assumed to be simple, i.e., without self-loops or multiple edges. We denote $G = (V, A)$ when considering G as a directed graph. As is typical in graph theory, $n = |V|$ represents the number of nodes, while $m = |E|$ denotes the number of edges.

Orientation and Graph-Sequence. The *edge orientation* or *orientation* of G is a directed graph $\mathcal{O} = (V_{\mathcal{O}}, A_{\mathcal{O}})$ such that for each edge $\{u, v\} \in E$, either $e = (u, v)$ or its *inverse* edge $e^{-1} = (v, u)$ is an element of $A_{\mathcal{O}}$. The edge e is then said to be *oriented towards* v. We call the process of deleting an edge $(u, v) \in A_{\mathcal{O}}$ and in return inserting (v, u), a *flip*. Furthermore, for any $t \in \mathbb{N}_0$ we call $\mathfrak{G} = (G_0, \ldots, G_t)$ an *edit-sequence of graphs* if there is a single insertion or deletion that distinguishes G_i from G_{i+1}. In other words, there exists an edge $e \in V^2$ such that $G_i = G_{i-1} + e$ or $G_i = G_{i-1} - e$ for every update $0 < i \leq t$. The initial graph, denoted as G_0, may either be empty or contain any number of edges. Similarly, a *sequence of orientations* of $\mathfrak{G}$ is a sequence of directed graphs $\mathfrak{O} = (\mathcal{O}_0, \ldots, \mathcal{O}_t)$ where every $\mathcal{O}_i$ is an edge orientation of G_i. In this case, an arbitrary number of edges may be flipped during the transition from $\mathcal{O}_{i-1}$ to $\mathcal{O}_i$.

Outdegree and Δ-Orientation. The *outdegree* (*indegree*) of a node $u \in V$ indicates the number of edges starting from (ending in) u, or more precisely: $\text{odeg}(u,G) := |\{v \in V \mid (u,v) \in A\}|$ and $\text{ideg}(u,G) := |\{v \in V \mid (v,u) \in A\}|$. Further, the *maximum outdegree* of G is defined as $\Delta_G = \max_{v \in V}(\text{odeg}(v,G))$ as well as $\Delta_{\mathcal{O}} = \max_{v \in \mathcal{O}}(\text{odeg}(v,\mathcal{O}))$ for the orientation of G. If $\Delta_{\mathcal{O}} \leq \Delta$ for some $\Delta \in \mathbb{N}_0$, $\mathcal{O}$ is called a Δ-*orientation* of G. Correspondingly, if all $\mathcal{O}_i$ of $\mathfrak{O}$ are Δ-orientations, we obtain a *sequence of* Δ-*orientations*.

Path. A *path* $p = \langle u_0, u_1, \ldots, u_t \rangle \subseteq V$ is a sequence of distinct nodes such that there is a connecting edge for all consecutive nodes. In the case of directed graphs (orientations), the edge must be aligned according to the order of objects in the path. By definition, a path therefore does not contain any cycles. Depending on the situation, we are using an equivalent representation by a sequence of edges to display $p = \langle e_0 = (u_0, u_1), \ldots, e_{t-1} = (u_{t-1}, u_t) \rangle \subseteq A$. In this case, $t = |p|$ is called the *length* of p. We *flip* p by flipping every edge once and denote the obtained result as the *inverse path* p^{-1} of p. Two paths, p and q, are said to *share edges* if there is an edge $e \in p$ such that $e \in q$. At last, $p = \langle u_0, u_1, \ldots, u_t \rangle$ is also considered an improving path if $\text{odeg}(u_t, G) < \text{odeg}(u_0, G) - 1$.

Connected Component, Induced Subgraph and Maximum Average Density. A *connected component* is a subset $C \subseteq G$ in which every pair of nodes is connected to each other by a path. For an induced subgraph $S \subseteq V$, we determine $n_S = |S|$ and $e_S = |\{\{u,v\} \in E \mid u,v \in S\}|$. The maximum average density of a graph G is defined as $d^*(G) = \max_{S \subseteq V} \lceil e_S / n_S \rceil$.

BFS and DFS. A breadth-first search (BFS) algorithm systematically explores a graph to locate a node with a certain property. It begins at a chosen node and exhaustively explores all nodes at the current depth before proceeding to the next level. Conversely, a depth-first search (DFS) algorithm traverses along each path as far as possible before backtracking to the previous node.

Neighborhood. The *neighborhood* of a node $v \in V$ is defined as the set of direct neighbors, i.e., $N(v) = \{u \in V \mid (v,u) \in A\}$. Accordingly, a d-*neighborhood* contains all nodes for which a path $p = \langle u_0, u_1, \ldots, u_t \rangle$ of length $|p| \leq d$ exists in A. Using this, a BFS is considered *bounded by* d if it only scans nodes within the d-neighborhood of the source.

(Pseudo-)arboricity. A *spanning tree* of a connected, undirected graph G is a subgraph T that is a tree and includes all vertices of G. Subsequently, a *spanning forest* of a graph G is a union of vertex disjoint spanning trees. By the *arboricity* $\alpha(G)$ we refer to the minimum number of spanning forests required to obtain a complete partition of G. For graph sequences, if $\alpha(G_i) \leq \alpha$ for all $G_i \in \mathfrak{G}$, we say that $\mathfrak{G}$ has *bounded arboricity* α.
Moreover, a connected graph that contains at most one cycle is called a *pseudotree*. The

corresponding further definitions of *spanning pseudotrees*, *spanning pseudoforests*, the *pseudoarboricity* $\rho(G)$, and *bounded pseudoarboricity* are derived from pseudotrees in the same way as the non-pseudo versions are derived from trees. We write α and ρ where the graph is clear from context.

Problem Definition. The objective of the *static edge orientation problem* is to minimize $\Delta_{\mathcal{O}}$ for a given graph G and a corresponding orientation $\mathcal{O}$. One may also be interested in finding the optimal solution. On the other hand, the *fully dynamic edge orientation problem* requires minimizing $\Delta_{\mathcal{O}_i}$ during each step G_i of an edit-sequence of graphs $\mathfrak{G}$, while the optimal version also always requires the best solution after each insertion or deletion.

CHAPTER 3

Related Work

This section presents an overview of research concerning edge orientations. Given our exploration of both static and dynamic algorithms in this work, each topic is addressed separately.

3.1 Static Edge Orientation Algorithms

There are numerous studies that deal with the solution of the static edge orientation problem. To begin with, a linear time 2-approximation algorithm for pseudoarboricity and maximum average density has been extensively investigated [2, 9, 21] and can also be applied to the static edge orientation problem. This is explained in more detail in Section 4.2 and is used as a comparison for the static algorithms investigated in this thesis. Picard and Queyranne [36] show a connection between pseudoarboricity $\rho(G)$ and maximum average density $d^*(G)$: $\rho(G) = \lceil d^*(G) \rceil$. This value also equals the lowest maximum outdegree [16, 1, 28]. Venkateswaran [39] proposes an optimal algorithm with a worst-case running time of $O(m^2)$ that searches for improving paths for all nodes with maximum outdegree. This approach was recently further developed and practically evaluated by Reinstädtler et al. [38]. This is also the underlying algorithm for one of our proposed dynamic approaches in Section 4.3.2. Georgakopoulos and Politopoulos [21] also propose an algorithm using binary search for determining the maximum average density of a subgraph, which can be used to calculate the maximum outdegree but does not give an edge orientation. This is an extended approach to the method of Goldberg [22]. Asahiro et al. [3] offer a flow-based method that achieves a running time of $O(m^{3/2} \log d^*)$ for computing the exact solution d^*. Similarly, Kowalik [28] also utilizes flows to approximate the solution, which runs in $O(m \log n \max(1, \log d^*)/\epsilon)$ and results in an orientation with bound $\lceil (1 + \epsilon) d^* \rceil$ for $\epsilon > 0$. This algorithm can also be used to determine the optimal solution.

For an extensive practical evaluation of these flow-based solutions and specific bounds, we refer the reader to Blumenstock [6], who also reports that the best known worst-case time complexity is $O(m^{3/2}\sqrt{\log\log d^*})$.

3.2 Fully Dynamic Edge Orientation Algorithms

We are now giving a brief overview of related work regarding dynamic approaches to the edge orientation problem. For a more in-depth discussion, we direct the reader to Borowitz et al. [7].

Brodal and Fagerberg [8] are the first to consider the edge orientation problem in the dynamic case by introducing a linear space data structure for storing graphs bounded by the arboricity α. A bound c for the arboricity is required as input, whereas adjacency queries require $O(c)$ time complexity. Furthermore, edge insertions can be performed in amortized time $O(1)$ just as edge deletions take amortized time $O(c + \log n)$. These updates are allowed as long as the changes to the forest partition maintain a bounded arboricity. Kowalik [27] further develops this by showing that the algorithm can maintain an orientation of $O(\alpha \log n)$ with α bounded by c. Insertions and deletions are here performed in amortized time $O(1)$. An algorithm without the requirement of a bound for the arboricity is provided by Kopelowitz et al. [26], maintaining a $O(\log n)$ edge orientation by performing both update operations in $O(\log n)$ for a constant arboricity α. He et al.[23] present a trade-off between the objective function (maximum outdegree of the edge orientation) and the costs of updates. For variable β, an $O(\beta\alpha)$ orientation is obtained with insertions and deletions requiring amortized time of $O(\frac{\log(n/(\beta\alpha))}{\beta})$ and $O(\beta\alpha)$, respectively. Berglin and Brodal [5] present an algorithm that, depending on an input parameter k, also provides such a trade-off. This allows for either an $O(\alpha + \log n)$ orientation with $O(\log n)$ running time or an $O(\alpha \log^2 n)$ orientation in $O(1)$ time, both of which are worst-cases. The report by Christiansen et al. [10] also features several algorithms, one of which maintains an $O(\alpha)$ orientation while using a worst-case $O(\log^2 n \log \alpha)$ time for update operations. Another algorithm maintains an $O(\alpha + \log n)$ orientation for a worst-case update time $O(\log n \log \alpha)$. Finally, Borowitz et al. [7] also introduce some new algorithms with promising results. For instance, the best of their algorithms in regards to quality is based on a breadth-first search and features a worst-case insertion time of $O(\Delta_{\mathcal{O}}^d)$ and deletion time of $O(\Delta_{\mathcal{O}})$ depending on the search depth d. We use the implementations of [7] for comparison and briefly explain the algorithms in Section 5.1.

CHAPTER 4

Algorithms

This chapter provides a detailed examination of the algorithms discussed in this work. We begin with a concise overview of the two data structures that are predominantly employed by the presented algorithms. Following that, we explore a range of static algorithms that utilize different approaches to approximate the optimal edge orientation. Afterward, we propose two fully dynamic algorithms, with the latter guaranteeing an optimal solution.

4.1 Underlying Data Structures: An Overview

There are multiple methods for storing graphs, each offering its own advantages in distinct scenarios. Therefore, choosing an appropriate data structure is crucial. This is especially important given that we are featuring both static and fully dynamic algorithms. In this work, we mainly focus on the two simple structures explained in detail below.

4.1.1 Concatenated Adjacency Arrays

The first concept is based on a pair of arrays. The first array holds indices representing the set of edges, sorted in ascending order based on the index of their respective source nodes. The second array contains markers to indicate where one source node transitions to the next, thus enclosing the set of outgoing nodes for each edge. That results in a clear assignment of a node to its outgoing edges. This setup is particularly inefficient for inserting or deleting nodes, hence making it more suitable for static algorithms. An edge orientation is represented by an additional array, marking each edge as included or not. In this context, if elements are added to or removed from a set of edges A in the pseudocode, then this is merely to be understood as a corresponding mark. As many algorithms require access to the inverse edge (v, u) given an edge (u, v), we also maintain an array containing the position of the inverse edges for those cases. It is possible to integrate its creation into the initialization of the graph while retaining linear complexity. This is done by simultaneously

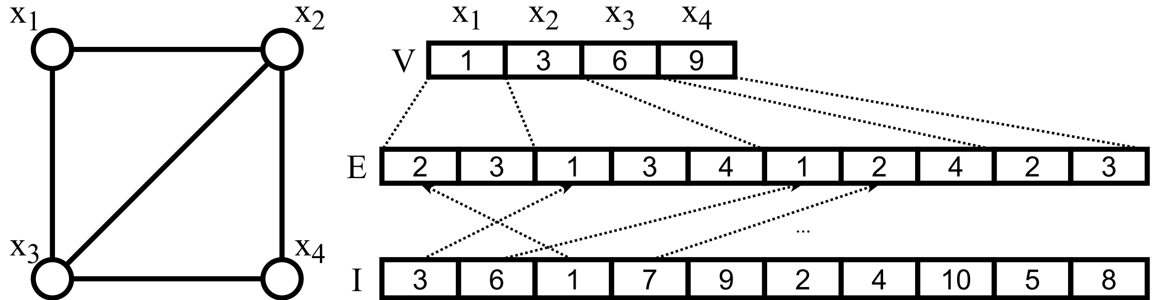


Figure 4.1: Concatenated adjacency array data structure with an inverse edge array I applied to an example graph G^1

inserting both the edge and its counterpart into the data structure. Consequently, their positions are instantly determinable, enabling saving them in the inverse edges array. Although preserving the inverse edges array does not impact the running time complexity, it adds unnecessary overhead. Therefore, for algorithms that solely necessitate access to the edges themselves, we omit this additional array. Since the second approach also utilizes arrays, we are referring to this concept by *concatenated adjacency arrays* from now on. Figure 4.1 shows an example of this data structure combined with an inverse edge array.

4.1.2 Individual Adjacency Arrays

The following data structure is well-suited for dynamic algorithms and offers an intuitive structure. Each node u maintains its own array $Adj[u]$ to store its adjacent nodes. These arrays are in turn grouped into a main array ordered by the index of the source nodes. To maintain dynamism despite the use of arrays, insertions can be achieved by pushing the new target node to the adjacency array of the source. Additionally, edges are deleted by swapping the target node with the last element of its corresponding array and then removing the last item. The position of inversed edges can be stored similarly as before, using a parallel data structure. These positions can be easily adjusted during update operations. Even within static scenarios, this version provides the advantage of enabling real edge deletions during algorithm execution, thereby eliminating the need to query the inverse edge. To differentiate this data structure from the first one, we refer to it as *individual adjacency arrays*. We provide an example of this data structure in Figure 4.2 as well.

In the experimental evaluation, we also use the MinDegNode algorithm to compare the speed of the two data structures for static scenarios and assess whether deleting edges can be beneficial in terms of time efficiency. Please refer to section 5.2 for details of the results. Otherwise, we generally assume the use of concatenated adjacency arrays for static algorithms and individual adjacency arrays for dynamic algorithms. However, we are mentioning whether an array for the inverse edges is mandatory.

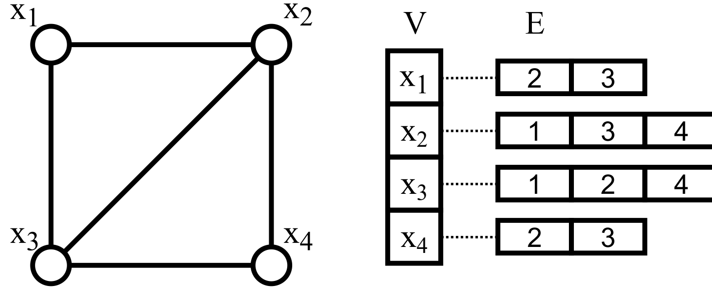


Figure 4.2: Individual adjacency array data structure applied to an example graph G^1

4.2 Static Algorithms

Several papers have already proposed methods for linearly approaching a solution to the static edge orientation problem, including the 2-approximation [2, 9, 21]. The result of a 2-approximation is guaranteed to be at most two times worse than the optimal solution. In this work, we compare this algorithm (MinDegNode) with novel greedy approaches that, to the best of our knowledge, have not been previously investigated. These new strategies are based on rather simple ideas, which may not necessarily guarantee approximation factors for the optimal solution, as we can observe in the experimental evaluation. Nevertheless, we include them for a comprehensive evaluation, which can either confirm their unsuitability or potentially reveal their effectiveness. We require each algorithm to be able to generate both the result of the edge orientation and the corresponding orientation of the edges.

4.2.1 Removing Nodes with Minimum Degree (MinDegNode)

We begin by introducing the strategy for computing a 2-approximation [2, 9, 21] for the edge orientation problem and give pseudocode in Algorithm 1. The algorithm consistently selects a node with minimum outdegree and orients all edges outwards. Selecting nodes based on outdegree can be efficiently implemented using a bucket priority queue. Therefore, at the beginning, we insert every node into the bucket corresponding to their initial outdegree. During each step of the algorithm, we delete one node inside the lowest bucket and demote all of its unscanned neighbors by one. This maintains an overall linear time complexity as it only requires decreasing the priority of a node by one for a total of m times during the entire execution. Furthermore, instead of removing incoming edges, we can successively add selected edges to an initially empty edge set $A_{\mathcal{O}}$. This makes a data structure for inverse edges obsolete. The resulting costs amount to $O\big(\sum_{x \in V} (\text{odeg}(x, G) + 1)\big) = (n + m)$. As previously mentioned, we are testing two versions: one utilizing concatenated adjacency arrays, indicated by MinDegNode,

Algorithm 1: Removing Nodes with Minimum Degree

```
1  procedure MinDegNode (G = (V, A)):
2  A_O := ∅; O := (V, A_O)
3  Label all nodes v ∈ V "unscanned"
4  while ∃ unscanned node in V do
5      u := arg min_{v∈V | v is unscanned}(odeg(v, (A \ {(v, u) | (u, v) ∈ A_O})))
6      foreach e = (u, v) ∈ A with v unscanned do
7          A_O = A_O ∪ {e} // edge oriented: u ⟶ v
8      end foreach
9      mark u scanned
10 end while
11 return (O, Δ(O))
```

and another one using individual adjacency arrays, named MinDegNodeList. We aim to investigate the viability of deleting edges in order to establish an orientation. Particularly, since the corresponding inverse edges have been removed, subsequent queries concerning them can be avoided.

Removing Nodes with Maximum Degree (MaxDegNode). In addition, let us compare this with the opposite strategy, which consistently removes a node with maximum outdegree and orients all edges towards it. The explicit pseudocode for MaxDegNode can be found in Algorithm 2. In this scenario, we initiate with an edge set $A_{\mathcal{O}} = A$, containing all edges of G. Subsequently, through the exclusion of edges adjacent to the current node from this set, the algorithm efficiently computes an edge orientation without necessitating an inverse edge array as well. Furthermore, a bucket priority is only required for sorting the nodes at the beginning. However, as no nodes are relocated to other buckets, this time overhead is avoided. Otherwise, the algorithm operates similarly to MinDegNode, including the running time complexity.

4.2.2 Orienting Edges Based on Progressive Degree (EdgeProgDeg)

We are presenting a similar algorithm to the one used by Reinstädtler et al. [38] for the FastImprove method of their static optimal algorithm. See Algorithm 3 for the detailed pseudocode. We begin by initializing an empty set $A_{\mathcal{O}}$, which contains all the edges selected during execution. Since this algorithm does not require the nodes to be processed in a specific order, we iterate through them based on their index. Each edge incident to such a node is then oriented towards the endpoint with the higher outdegree within $A_{\mathcal{O}}$. If there is a tie, the corresponding edge is oriented towards the current node. Accordingly, the outdegree of nodes within $A_{\mathcal{O}}$ is maintained in a bucket priority queue, equivalent to the

Algorithm 2: Removing Nodes with Maximum Degree

```
1  procedure MaxDegNode(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V "unscanned"
4  while ∃ unscanned node in V do
5  |   u := arg max_{v∈V|v is unscanned}(odeg(v, O))
6  |   foreach e = (u, v) ∈ A with v unscanned do
7  |   |   A_O = A_O \ {e} // edge oriented: u ⟵ v
8  |   end foreach
9  |   mark u scanned
10 end while
11 return (O, Δ(O))
```

Algorithm 3: Orienting Edges Based on Progressive Degree

```
1  procedure EdgeProgDeg(G = (V, A)):
2  A_O := ∅; O := (V, A_O)
3  Label all nodes v ∈ V "unscanned"
4  foreach u ∈ V do
5  |   foreach e = (u, v) ∈ A with v unscanned do
6  |   |   if odeg(u, O) < odeg(v, O) then
7  |   |   |   A_O = A_O ∪ {e} // edge oriented: u ⟶ v
8  |   |   else
9  |   |   |   A_O = A_O ∪ {e^{-1} = (v, u)} // edge oriented: u ⟵ v
10 |   |   end if
11 |   end foreach
12 |   mark u scanned
13 end foreach
14 return (O, Δ(O))
```

one used in MinDegNode. As both edge and inverse edge can be added to the orientation, we require an inverse edge array in this case. Utilizing the same bucket priority queue as for MinDegNode ensures a running time complexity of $O(n + m)$ as well.

Orienting Edges Based on Initial Degree (EdgeInitDeg). We now discuss a modification to EdgeProgDeg, which is detailed in Algorithm 4. This algorithm involves orienting edges not based on an initially empty set $A_{\mathcal{O}}$ but rather according to the initial outdegree of the nodes. The initial positions within the bucket priority queue are therefore introduced by inserting nodes based on their outdegree within G. The outdegree is then subsequently decreased whenever an adjacent edge is removed. Otherwise, this approach works identically.

Algorithm 4: Orienting Edges Based on Initial Degree

```
1  procedure EdgeInitDeg(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V "unscanned"
4  foreach u ∈ V do
5      foreach e = (u, v) ∈ A with v unscanned do
6          if odeg(u, O) < odeg(v, O) then
7              A_O = A_O \ {e^{-1} = (v, u)} // edge oriented:  u ⟶ v
8          else
9              A_O = A_O \ {e} // edge oriented:  u ⟵ v
10         end if
11     end foreach
12     mark u scanned
13 end foreach
14 return (O, Δ(O))
```

4.2.3 Orienting Individual Edges with a Minimum Degree Node (MinDegEdge)

We also propose an approach in Algorithm 5 that individually processes each edge, as opposed to sequentially handling entire nodes. Therefore, an unscanned node with minimum outdegree is successively selected, and exactly one of its yet unscanned edges is oriented outward each time. This process continues until no more unscanned edges are remaining. The outdegree of the nodes is again continuously corrected and managed using a bucket priority queue. Note that we must use an inverse edge array to ensure that both the edge and its inverse are marked as scanned after the inverse edge is removed from the orientation. This is to prevent processing from the opposite direction. This can be observed in Line 8 of Algorithm 5.

Orienting Individual Edges with a Maximum Degree Node (MaxDegEdge). We are also exploring the opposite approach to MinDegEdge. Similar to the previous method, we successively select an unscanned node with maximum outdegree and orient exactly one of its unscanned edges towards it. The implementation remains consistent in every other aspect and can be found in more detail in Algorithm 6.

Algorithm 5: Orienting Individual Edges with a Minimum Degree Node

```
1  procedure MinDegEdge(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V and all edges e ∈ A "unscanned"
4  while ∃ unscanned node in V do
5  |  u := arg min_{v∈V | v is unscanned}(odeg(v, O))
6  |  if ∃ unscanned edge e = (u, v) ∈ A then
7  |  |  A_O = A_O \ {e^{-1} = (v, u)} // edge oriented:  u ⟶ v
8  |  |  mark e and e^{-1} scanned
9  |  else
10 |  |  mark u scanned
11 |  end if
12 end while
13 return (O, Δ(O))
```

One specialty of these two algorithms is, that they react sensitively to the ordering of the edges, meaning that the outcome may vary based on how the edges are sorted. Since this usually depends on how the graph was stored in the first place, this offers an opportunity to adjust the reading process into the data structure accordingly. As explained in Section 4.1.1, both the edge and its inverse counterpart get inserted into the data structure simultaneously. While reading a node, the corresponding inverse edges are placed at the next free positions for all adjacent nodes. This results in them being potentially processed earlier during the execution of the algorithm. Hence, we aim to arrange the insertion of nodes based on their degree. If nodes are inserted in ascending order, we mark the corresponding algorithm by $o = min$. Conversely, if they are included in descending order, we indicate the algorithm by $o = max$. Note that sorting n nodes requires $O(n)$ time when using a bucket priority queue because every node has a maximum outdegree of $n - 1$ and is necessarily an integer.

4.2.4 Using 2-Approximation as Lower Bound (2ApproxMix)

The fundamental idea of this algorithm is calculating a lower bound and utilizing it to achieve a closer approximation for the edge orientation. The algorithmic framework is explicitly laid out in the pseudocode of Algorithm 7. We start by using the property of the MinDegNode algorithm for the computation of a 2-approximation, denoted as d_{approx}. This is then used to calculate a lower bound for the edge orientation:

$$d_{bound} = \frac{d_{approx}}{2} \tag{4.1}$$

Algorithm 6: Orienting Individual Edges with a Maximum Degree Node

```
1  procedure MaxDegEdge(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V and all edges e ∈ A "unscanned"
4  while ∃ unscanned node in V do
5  |   u := arg max_{v∈V|v is unscanned}(odeg(v, O))
6  |   if ∃ unscanned edge e = (u, v) ∈ A then
7  |   |   A_O = A_O \ {e} // edge oriented:  u ⟵ v
8  |   |   mark e and e^{-1} = (v, u) scanned
9  |   else
10 |   |   mark u scanned
11 |   end if
12 end while
13 return (O, Δ(O))
```

In the second step, at most d_{bound} edges are oriented outward for each node, with the order determined by the smallest degree. The degree is updated iteratively again during the process within a bucket priority queue. Any node with an outdegree still less than or equal to d_{bound} is subsequently removed. In step 3 we make use of one of the most promising linear time algorithms MinDegEdge (see Section 5.2 for the experimental evaluation) to align the remaining edges. The implementation is thus based on using underlying algorithms, which is why an inverse edge array is also required here. And since we exclusively execute algorithms with linear running time in each step, the total time only increases by a constant factor.

For this algorithm, it is evident that obtaining as sharp a lower bound as possible is significant. Although the factor $\sigma = 2$ provides a guaranteed bound, it may be significantly distant from the optimum if the solution of the 2-approximation of MinDegNode is good. Therefore, we also introduce some additional factors $\sigma \in (1, 2]$, which are not definite bounds but might get closer to the optimal value on average. We intentionally disregard factors less than or equal to 1, as this essentially results in a MinDegNode algorithm. Furthermore, a value above 2 only further approximates the behavior of MinDegEdge.

Algorithm 7: Using 2-Approximation as Lower Bound

```
   input: approximation factor σ
1  procedure 2ApproxMix(G = (V, A)):
2  // Phase 1:  compute lower bound
3  (G', d_approx) := MinDegNode(G)
4  d_bound := d_approx / σ
5  // Phase 2:  orient at most d_bound edges for every node
6  A_O := A; O := (V, A_O)
7  Label all nodes v ∈ V and all edges e ∈ A_O "unscanned"
8  while ∃ unscanned node in V do
9      u := arg min_{v∈V | v is unscanned}(odeg(v, O))
10     for i = 0 to min(odeg(u, O), d_bound) do
11         e := unscanned edge (u, v) ∈ A_O
12         A_O = A_O \ {e^{-1} = (v, u)} // edge oriented:  u ⟶ v
13         mark e scanned
14     end for
15     mark u scanned
16 end while
17 // Phase 3:  orient remaining edges
18 // perform MinDegEdge but the labelling of the edges in line 3
       is adopted from A_O
19 O := MinDegEdge(O)
20 return (O, Δ(O))
```

4.2.5 Edge Orientation Using Spanning Forests

Borowitz et al. [7] described the following technique, which we first explain before discussing potential algorithms. A closely related problem to finding the edge orientation is to fully partition a graph into the minimum number of spanning forests, denoted as α. Once such a decomposition has been calculated, we can use it as the basis for constructing an edge orientation. This is achieved by selecting an arbitrary root for each tree within a forest and orienting each edge of the respective tree towards it. As a result, within these forests, every node can only have a maximum outdegree of one. This arrangement yields a maximum edge orientation of α, as there are at most α forest partitions.

First of all, there is always an α-orientation for static graphs. According to Nash-Williams [32, 33, 11] , a graph G possesses arboricity α if and only if its edge set E can be partitioned into $E_1, \ldots, E_\alpha$, where each (V, E_i) forms a forest for all $i \in \{1, \ldots, \alpha\}$. Such a decomposition of the graph into the minimum number of spanning forests can be done in polynomial time and has already been well researched [36, 18, 19].

FOREST. Since we are focusing on fast linear-time algorithms, we are exploring a different approach that approximates α rather than aiming for an exact solution. FOREST, developed by Nagamochi and Ibaraki [31], is such a linear-time approach for finding a sparse k-connected spanning subgraph. Moreover, this algorithm also results in a partition of G into the spanning forests $(V, E_1), (V, E_2), \ldots, (V, E_{|V|-1})$. Specifically, they proved that each (V, E_i) is a maximum spanning forest in $(V, E \setminus (E_1 \cup E_2 \cup \cdots \cup E_{i-1}))$ for $i \in \{1, 2, \ldots, |V| - 1\}$ ([31], Lemma 2.5). The algorithm consistently selects the node that currently belongs to the most forests and adds all its unscanned neighbors to the next available forest. We now delve into the specifics and refer to the pseudocode provided in Algorithm 8 for more details. First of all, the number of forests a node belongs to is consistently maintained in a bucket priority queue denoted by r. This way, each forest is labeled with a unique index between 1 and $|V| - 1$. During execution, an unscanned node u with maximal $r[u]$ is systematically chosen and processed completely. This means that every unscanned neighbor v is integrated into the next feasible forest, which is achieved by adding the respective edge to the set $E_{r[v]+1}$ and increasing the value in r accordingly. Thus, an edge is always assigned to the forest indexed by $r[v] + 1$. Furthermore, cycles cannot form within a set E_i, as edges are only added if one of its endpoints is not yet part of that forest. Note that employing a bucket priority queue does not compromise the running time complexity, since only $|E|$ entries are incremented during execution. An example of the final result of FOREST applied to the graph in Figure 4.3 is illustrated in Figure 4.4.

Edge Orientation. In order to transform FOREST into the edge orientation algorithm NI, we additionally have to find a root for every tree and align the edges accordingly. Nagamoshi and Ibaraki observed that each nontrivial tree (a tree containing at least one edge) has exactly one root node, which is the first scanned node of that tree. This is due to the nature of the FOREST algorithm, which does not permit merging trees. Using Fig-

Algorithm 8: FOREST by Nagamochi and Ibaraki [31]

```
1  procedure FOREST(G = (V, E)):
2  E_1 := E_2 := ··· := E_|E| := ∅
3  Label all nodes v ∈ V and all edges e ∈ E "unscanned"
4  r[v] := 0 for all v ∈ V // Let r[v] := i denote that v has been reached
       by an edge of the forest F_i = (V, E_i)
5  while ∃ unscanned node in V do
6      u := arg max_{v∈V | v is unscanned} (r[v])
7      foreach unscanned edge e = {u, v} ∈ E do
8          E_{r[v]+1} = E_{r[v]+1} ∪ {e}
9          if r[u] = r[v] then r[u] = r[u] + 1
10         r[v] = r[v] + 1
11         mark e scanned
12     end foreach
13     mark u scanned
14 end while
15 return E_1, E_2, ..., E_|E|
```

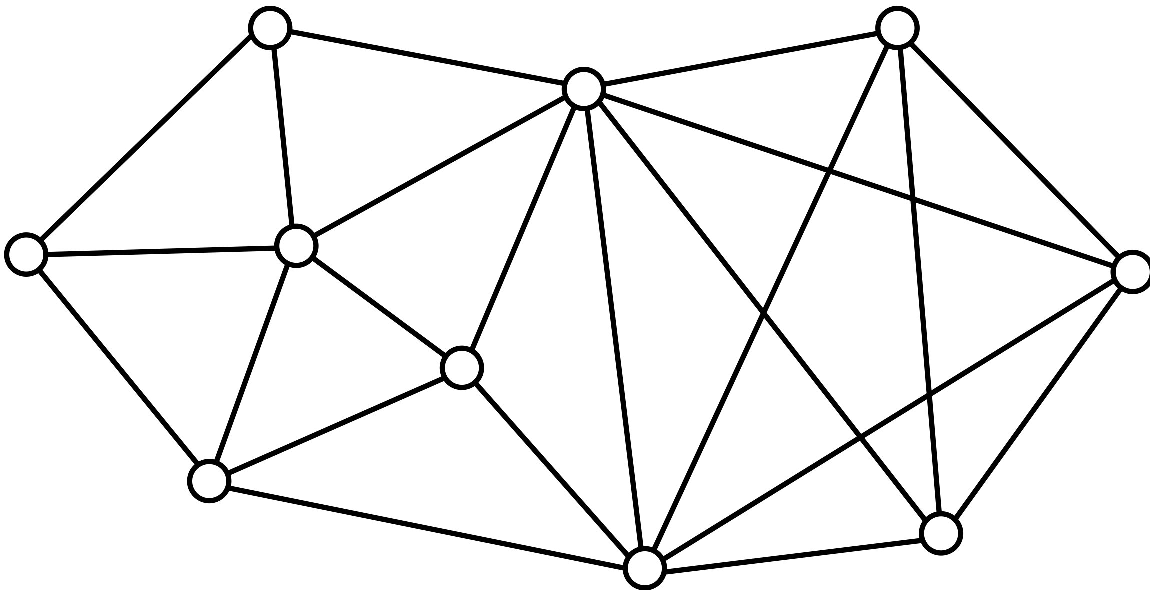


Figure 4.3: Example graph G^2 with $\Delta_{G^2} = 7$

ure 4.4 as an example, E_1, E_2 and E_4 contain exactly one nontrivial tree, while E_3 is a forest of two nontrivial trees. A new tree is initiated whenever the current node u and one of its unscanned neighbors v belong to an equal number of forests, i.e., $r[u] = r[v]$. In this scenario, node u becomes the root of a tree that belongs to the forest $(V, E_{r[u]+1})$.

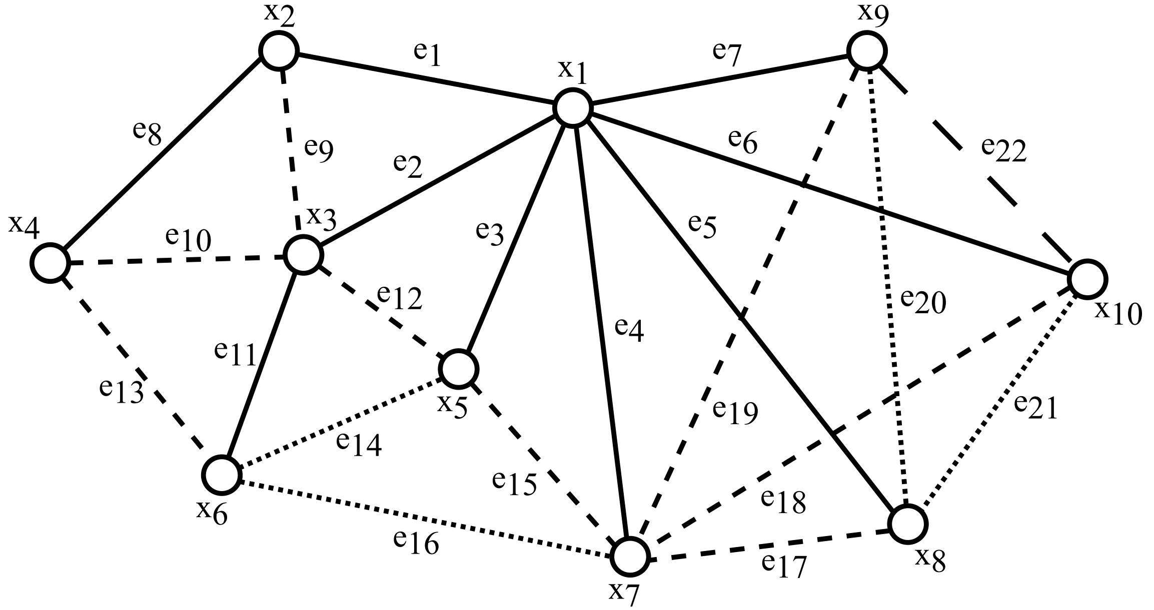


Figure 4.4: Partition of $G^2 = (V, E_1 \cup E_2 \cup E_3 \cup E_4)$ obtained by FOREST. Edges in E_1: O——O; edges in E_2: O- - - -O; edges in E_3: O·····O; edges in E_4: O— — O. Nodes and Edges are indexed in the order they were scanned.

Consequently, all unscanned edges containing u are oriented towards u, as this is either the root of the new tree or lies along the path to the corresponding root node. This results in a Δ-orientation, where Δ corresponds precisely to the resulting number of spanning forests as the following theorem shows.

Theorem 1: For a graph $G = (V, E)$, let $\mathcal{O}$ be the orientation obtained by Algorithm 9, then $\Delta_{\mathcal{O}} = \max_{v \in V}(r[v])$.

Proof. Theorem 1
Let us denote the resulting value after the execution by $k = \max_{v \in V}(r[v])$. Consider a step in the algorithm where we process an edge $e = (u, v)$ with $r[u] = r[v] = k - 1$. Such a situation must occur, as priorities of r only get increased by one and only the lower priority out of $r[u]$ and $r[v]$ gets increased. Therefore, e is oriented towards u. Since v is an unscanned node whose scanned edges are all oriented outwards and part of k spanning forests, v now has k scanned edges. That implies $k \leq \Delta_{\mathcal{O}}$. $\Delta_{\mathcal{O}} \leq k$ is immediate, as each node has only one outgoing edge per spanning forest. □

Since $|E_i| \leq |V| - 1$ for all $i \in \{1, \ldots, |V| - 1\}$, we also obtain a formula for calculating an upper bound δ for $\Delta_{\mathcal{O}}$ (see [31]):

$$|E| \leq \delta|V| - \delta(\delta + 1)/2 \tag{4.2}$$

Algorithm 9: Edge Orientation Algorithm Using FOREST

```
1  procedure NI(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V and all edges e ∈ A_O "unscanned"
4  r[v] := 0 for all v ∈ V
5  while ∃ unscanned node in V do
6      u := arg max_{v∈V | v is unscanned}(r[v])
7      foreach e = (u, v) ∈ A with v unscanned do
8          r[v] = r[v] + 1
9          A_O = A_O \ {e} // edge oriented:  u ⟵ v
10     end foreach
11     mark u scanned
12 end while
13 return (O, Δ(O))
```

Please refer to Algorithm 9 for the extended pseudocode and to Figure 4.5 for an exemplary result on an example graph G^2. In our implementation, we utilized concatenated adjacency arrays without an inverse edges array, as we do not require access to them. Also, note that Line 9 in Algorithm 8 is unnecessary for our purpose and therefore omitted in the NI approach as we do not calculate the forest partition itself. It is evident that the time complexity $O(n + m)$ is the same as for FOREST.

Selecting the Starting Node. Previously, the starting node was selected solely based on the index, which depends on the order in which the edges were stored. By introducing selection criteria, we aim to investigate the extent to which the starting node influences the outcome. To identify a suitable node in advance, we require an indicator, which in our scenario is the initial degree of the node. For the experimental evaluation, we are proposing two representative strategies: one selecting nodes based on the lowest degree (NIStartMinDeg) and another based on the highest degree (NIStartMaxDeg). The adapted pseudocode for both can be found in Algorithm 10, where for NIStartMaxDeg in Line 7, the argmin function has to be replaced with an argmax function. More specifically, a starting node must be selected for each connected component of a graph. This is the case if $r[u] = 0$ applies for each unscanned node u, as the following theorem shows.

Theorem 2: Let $G = (V, E)$ be an undirected graph. If $r[u] = 0$ holds for all unscanned nodes $u \in V$ during a step in the execution of Algorithm 10, then every connected component $C \subseteq G$ that contains an unscanned node exclusively consists of unscanned nodes.

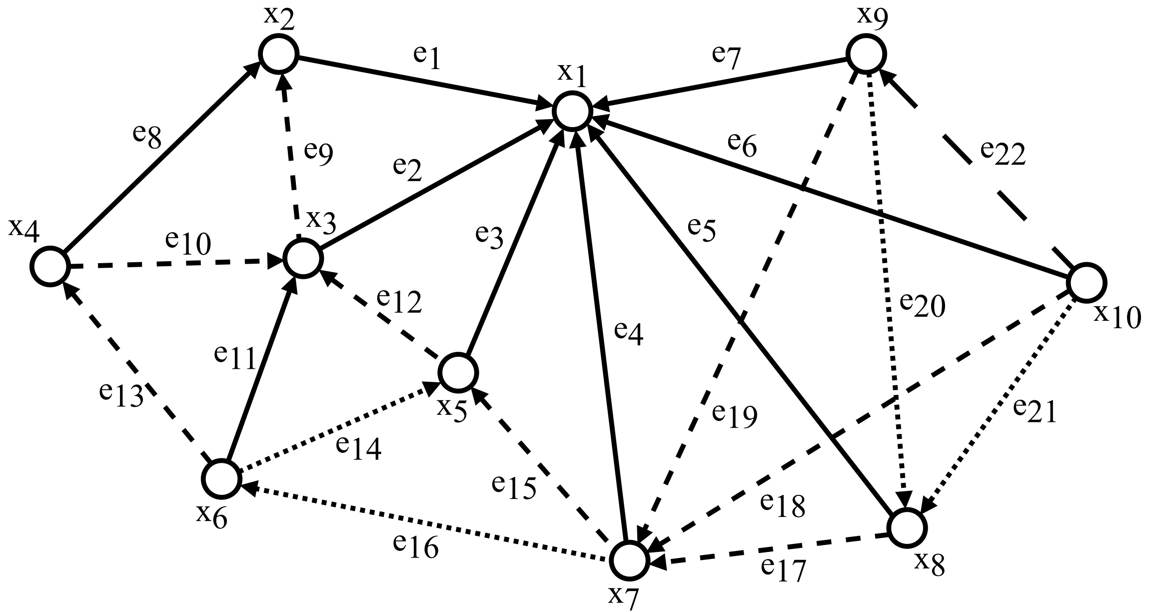


Figure 4.5: Edge Orientation $\mathcal{O}_1$ of G^2 obtained by NI with $\Delta_{\mathcal{O}_1} = 4$. For better visualization, the different forests (V, E_i) have been marked distinctively. Edges in E_1: O——O; edges in E_2: O- - -O; edges in E_3: O·····O; edges in E_4: O— — O. Nodes and Edges are indexed in the order they were scanned.

Proof. Theorem 2
Let C denote a connected component of G with at least one unscanned node $u \in C$. Assume that C also contains a scanned node $v \in C$. According to the definition of connected components, there is a path $p = \langle v, \ldots, u \rangle$ in C. As p contains either scanned or unscanned nodes, there must be an edge $e = (x, y)$ with a scanned node $x \in C$ and an unscanned node $y \in C$. Consequently, according to Algorithm 10, for every unscanned neighbor w of x holds $r[w] > 0$, contradicting the precondition $r[y] = 0$. ↯
□

Improving the Forest Approach: Pseudoarboricity. Overall, there are two possible improvements in this derivation of the edge orientation. As stated previously, the algorithm of Nagamochi and Ibaraki only provides an approximation of the arboricity. Secondly, even an α-orientation is not necessarily optimal. The latter can easily be demonstrated by examining a circle of three nodes. In this scenario, the optimal solution achieves a 1-approximation, whereas the arboricity is two. This aligns with an observation from our previous approach: the root nodes of each forest lacked outgoing edges, leaving room for one to be added without violating the 1-approximation criterion. Consequently, we are extending our approach beyond forests to also include pseudoforests in the partitioning process.

Algorithm 10: NI with Minimum Degree Starting Node

```
1  procedure NIStartMinDeg(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V and all edges e ∈ A_O "unscanned"
4  r[v] := 0 for all v ∈ V
5  while ∃ unscanned node in V do
6  |  if ∀ unscanned v ∈ V : r[v] = 0 then
7  |  |  u := arg min_{v∈V|v is unscanned}(outdegree(v,O))
8  |  else
9  |  |  u := arg max_{v∈V|v is unscanned}(r[v])
10 |  end if
11 |  foreach e = (u, v) ∈ A with v unscanned do
12 |  |  r[v] = r[v] + 1
13 |  |  A_O = A_O\{e}// edge oriented:  u ⟵ v
14 |  end foreach
15 |  mark u scanned
16 end while
17 return (O, Δ(O))
```

Kowalik [28] demonstrated that the problems of partitioning a graph into Δ pseudoforests and finding a Δ-orientation are equivalent. This means that conversions between the two problems can be done using only linear time proportional to the number of edges. Since there has been plenty of research about calculating the pseudoarboricity and constructing the corresponding decompositions in polynomial time, many algorithmic solution approaches have been created [36, 20, 19, 28]. Of course, these are thus also capable of solving the edge orientation problem. Furthermore, there are additional correlations between arboricity and pseudoarboricity as it can be shown that $\rho(G) \leq \alpha(G) \leq \rho(G) + 1$ (see [36], Corollary 3-1).

However, since our primary focus is on preserving a linear time complexity, we intend to propose an extension to the algorithm of Nagamochi and Ibaraki for pseudoforest partitioning. A first observation is that by removing one edge from the circle of a 1-orientation pseudotree, one endpoint of the former edge becomes the root of that tree, with all edges oriented towards it. Accordingly, the natural idea is to allow the algorithm to proceed as before, but upon the creation of each new tree, orient exactly one edge away from its root to potentially form a cycle. As previously established, a new tree is initiated by an edge $e = (u, v)$ with $r[u] = r[v]$. If such a scenario occurs, it is primarily important to create at most one cycle. The simplest approach to achieve this is to orient the first edge away from u, as demonstrated in Algorithm 11. Note that the corresponding neighbor is not added to the new forest for now (by not increasing $r[u]$), allowing it to potentially connect later through another edge, which can result in a cycle. We also provide an illustration of this

Algorithm 11: NI for Approximating Pseudoarboricity

```
1  procedure NIPseudoFirst(G = (V, A)):
2  A_O := A; O := (V, A_O)
3  Label all nodes v ∈ V and all edges e ∈ A_O "unscanned"
4  r[v] := 0 for all v ∈ V
5  while ∃ unscanned node in V do
6      u := arg max_{v∈V | v is unscanned}(r[v])
7      counter := 0
8      foreach unscanned edge e = (u, v) ∈ A_O do
9          if r[u] = r[v] and counter = 1 then
10             A_O = A_O \ {e^{-1} = (v, u)} // edge oriented:  u ⟶ v
11             mark e scanned
12         else
13             r[v] = r[v] + 1
14             A_O = A_O \ {e} // edge oriented:  u ⟵ v
15             mark e^{-1} = (v, u) scanned
16         end if
17         counter++
18     end foreach
19     mark u scanned
20 end while
21 return (O, Δ(O))
```

strategy applied to the graph G^2 in Figure 4.6. Given the possibility that the first edge of a tree might also be the last one, potentially rendering our initial alignment useless, we try an alternative approach for comparison. This time, we are aligning the second edge outwards, provided that the tree has at least two edges. To do this, we adapt Algorithm 11 by changing the condition of the if-statement in Line 9 to $counter = 2$. Note that we are only approximating ρ and also require an inverse edge array for this extension, as access to them is now necessary.

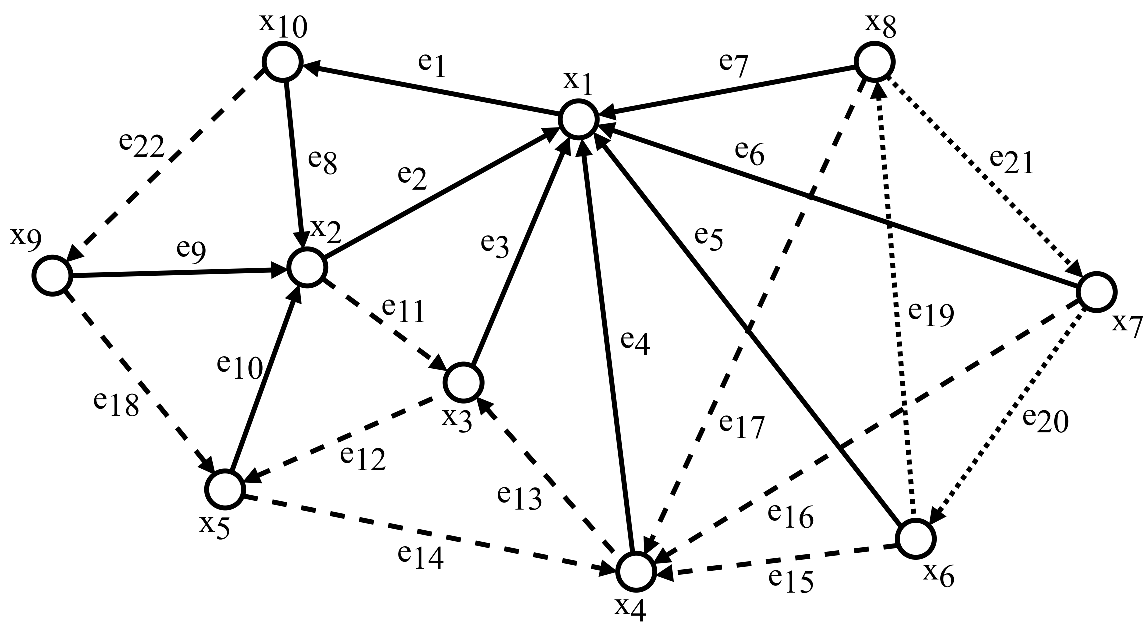


Figure 4.6: Edge Orientation $\mathcal{O}_2$ of G^2 obtained by NIPseudoFirst with $\Delta_{\mathcal{O}_2} = 3$. For better visualization, the different pseudoforests (V, E_i) have been marked distinctively. Edges in E_1: o——o; edges in E_2: o- - - -o; edges in E_3: o·····o. Nodes and Edges are indexed in the order they were scanned.

4.3 Dynamic Algorithms

4.3.1 Descending BFS Algorithm (DescBFS)

The paper of Borowitz et al. [7] featured a variety of fully dynamic edge orientation algorithms. We also add another version to this assemblage, which is intended to be a fusion between the Descending Degrees Algorithm and the Improving u-y-Path Search Algorithm.[1] On that note, we proceed in the same way as with the Descending Degrees Algorithm, but instead of only examining the immediate neighborhood for a suitable node, we extend every search to the d-neighborhood. The search for the minimum node across the entire d-neighborhood requires significant computational resources. However, considering that paths to nodes with lower degrees may still be flipped due to the descending steps, this extra effort is presumably not worth it. Therefore, we adapt the approach of the BFS Algorithm and terminate the search upon encountering a feasible node. Please refer to the pseudocode in Algorithm 12 for the precise procedure.

Let us now move on to a detailed explanation. The underlying data structure consists of individual adjacency arrays $Adj[u]$, storing all adjacent nodes of u. Additionally, we utilize a bucket priority queue to maintain the outdegree for each node and keep the current objective function value in a global variable $\Delta_{\mathcal{O}}$.

When inserting an edge $e = (u, v)$, v is added to the adjacency array $Adj[u]$ as a new neighbor of u. Subsequently, the algorithm immediately returns if the outdegree of u is less than $\Delta_{\mathcal{O}}$. This is a pruning step designed to prevent a time-consuming search when an insertion does not affect the objective function. Another pruning condition for immediate termination is $\Delta_{\mathcal{O}} = 1$, as it is impossible to improve the current solution since an efficient flip requires a node with an outdegree lower than $\text{odeg}(u, \mathcal{O}) - 1$. These data reduction strategies are adopted following the methodology of [7] for the sake of comparability.

If neither condition for a return is met, a search is then conducted in the d-neighborhood of u. We do this by running a breadth-first search (BFS) that checks all paths of length d with u as the source. If a candidate y is found such that $\text{odeg}(y, \mathcal{O}) < \text{odeg}(u, \mathcal{O}) - 1$, the entire path $p = \langle u, \ldots, y \rangle$ is flipped. Note that every edge in p is orientated towards y so that, apart from u and y, all nodes of the path have both an incoming and an outgoing edge. Therefore, only the outdegree of y increases, whereas the outdegree of u decreases by one. During the BFS search, the positions of the individual nodes of the path can be stored in the corresponding adjacency arrays, which is why the flipping of an entire path can be implemented in $O(|p|)$.

This process is then repeated by starting in y until a maximum of r searches have been performed or no feasible node was found for some source node. If the latter occurs, i.e. a search is unsuccessful even though the maximum number of searches has not yet been reached, the process is restarted from the original node u.

[1] For more details, see Section 5.1 and consult Algorithm 18 and Algorithm 20 (Appendix A.3.1), or for a more comprehensive version, see Borowitz et al. [7].

Algorithm 12: Descending BFS Algorithm

```
1  global variable: Δ_O
   input: depth d, # steps s
2  procedure Insertion(u, v):
3  Adj[u] = Adj[u] ∪ {v}
4  if odeg(u, O) < Δ_O or Δ_O = 1 then return
5  count := 0
6  while count < s and DescendingBFS(O, u, count) ;
7  procedure DescBFS(O = (V_O, Adj), u, count):
8  // find a path p = (u, ..., w) so that odeg(w, O) < odeg(u, O) − 1
9  p = BFS-Search(u, d) // bounded by depth d
10 count + +
11 if p ≠ ∅ then
12 |   flip all edges of p
13 |   if count < s then
14 |   |   DescendingBFS(O, u, count)
15 |   end if
16 |   return True
17 else
18 |   return False
19 end if
20 procedure Deletion(u, v):
21 Adj[u] = Adj[u] \ {v}
22 Adj[v] = Adj[v] \ {u}
```

Of course, this procedure is instantly canceled if the original search is unsuccessful. The deletions are kept simple in imitation of the competitors, simply removing the respective nodes from the adjacency arrays.

Since the maximum amount of breadth-first searches cannot exceed r and every one of them is bounded by d, the worst-case running time of an insertion is $O(r(\Delta_{\mathcal{O}})^d)$. On the other hand, the deletion operation requires only a running time complexity of $O(\Delta_{\mathcal{O}})$.

4.3.2 Dynamic Optimal Edge Orientation Algorithm (DynOptEO)

The discussion so far has been primarily focused on algorithms lacking a guarantee of optimality. However, in this section, we propose a dynamic algorithm designed to compute the exact solution for each step within a graph sequence $\mathfrak{G}$. This approach is an adaptation of a static algorithm initially proposed by Venkateswaran [39] and recently further refined by Reinstädtler et al. [38]. Before exploring the new dynamic version, we first review the underlying static algorithm. We are then detailing its connections to the dynamic version while discussing adopted techniques and also outlining the necessary basic components, including invariants. Next, we examine the update operations, starting with insertions and followed by deletions, whose correctness we also prove. This is concluded by a running time analysis.

Solving the Static Edge Orientation Problem

Venkateswaran [39]. The original algorithm minimizes the maximum indegree, but since this can easily be transferred to the outdegree case, we are only considering the latter. Venkateswaran begins with an arbitrary orientation and defines two sets $S = \{v \in V \mid \text{odeg}(v, \mathcal{O}) = k\}$ and $T = \{v \in V \mid \text{odeg}(v, \mathcal{O}) \leq k - 2\}$ that depend on the maximum outdegree $k = \max_{v \in V}(\text{odeg}(v, \mathcal{O}))$. The core idea of this algorithm is to search for improving paths, starting from nodes within S and ending in T. Upon discovering such a path $p = \langle u, \ldots, v \rangle$, it is flipped, consequently decreasing $\text{odeg}(u, \mathcal{O})$. Therefore, u is removed from S and if $\text{odeg}(v, \mathcal{O}) > k - 2$, v is also removed from T. If S becomes empty, the two sets S and T are formed again for $k = k - 1$. This process continues until a complete run over all peak nodes in S fails to yield any improvement, at which point the algorithm terminates. For a more in-depth explanation of this approach, we refer to Reinstädtler et al. [38]. A significant part of the contribution provided in their paper consists of proposing enhancing engineering techniques, some of which can also be transferred to the dynamic variant.

Adapted Techniques. In the dynamic variant of the algorithm, we are able to utilize previously gathered information to optimize our search for improving paths for each update operation. This approach significantly reduces the need to re-solve the entire graph for most updates. However, there are exceptional cases where it is unavoidable to regain optimality by completely solving the graph. For these instances, we are using a modified version of the algorithm provided by Reinstädtler et al. [38]. This especially includes some engineering techniques, which can be transfered to the dynamic case. Those are discussed in the following. The respective pseudocode for finding and potentially flipping an improving path is presented in FindPath and can be found in Algorithm 14. For a comprehensive solution to the static edge orientation problem, please refer to the pseudocode for FindOptimal, which is provided in Algorithm 13 and extensively utilizes FindPath.

Algorithm 13: DynOptEO: FindPath (see [38, 39])

```
1  global variables: maxOutDegree = 0, maxNodeCount = 0,
     visited = [False] × |V|
2  procedure Flip(O = (V_O, A_O), e = (u, v)):
3  A_O = (A_O \ e) ∪ {(v, u)}
4  procedure FindPath(O = (V_O, A_O), u, d):
5  if visited[u] then return False
6  for e = (u, v) ∈ A_O do
7      if odeg(v, O) < d − 1 then
8          Flip(O, e)
9          return True
10     end if
11 end for
12 visited[u] = True
13 for e = (u, v) ∈ A_O do
14     if odeg(v, O) = d − 1 then
15         if FindPath(O, v, d) then
16             Flip(O, e)
17             visited[u] = False^2
18             return True
19         end if
20     end if
21 end for
22 return False
```

Since the experimental evaluation in [38] suggests that the algorithm performs better using a DFS compared to a BFS, we also adopt this for the dynamic variant. We further refine our approach by using *independent paths* for the DFS, so that paths intersecting nodes with maximum outdegree are not further explored. This is because flipping a path that overlaps with another can invalidate that one. All possible paths of the other peak node are explored anyway, which upon an improvement allows searches to run through this node in the next iteration. Additionally, we maintain a shared visited array across all peak nodes, which is reset only after a complete cycle over all peak nodes. This strategy helps maintain efficiency by marking subgraphs that lacked improving nodes during initial searches as visited, thereby preventing redundant searches by other peak nodes during the same cycle.

Algorithm 14: DynOptEO: FindOptimal (see [38, 39])

```
1  global variables: maxOutDegree = 0, maxNodeCount = 0,
     visited = [False] × |V|
2  procedure FindOptimal(𝒪 = (V_𝒪, A_𝒪)):
3  path_improved := True
4  while path_improved do
5  |   path_improved = False
6  |   all_paths_improved := True
7  |   for u ∈ V_𝒪 with odeg(u, 𝒪) = maxOutDegree do
8  |   |   if FindPath(𝒪, u, odeg(u, 𝒪)) then
9  |   |   |   path_improved = True
10 |   |   else
11 |   |   |   all_paths_improved = False
12 |   |   end if
13 |   end for
14 |   if all_paths_improved then
15 |   |   maxOutDegree − −
16 |   end if
17 |   reset visited
18 end while
19 maxNodeCount = |{v ∈ V_𝒪 | odeg(v, 𝒪) = maxOutDegree}|
```

Engineering DynOptEO. We are now explaining the modifications for the dynamic version of this algorithm. As already laid out in Algorithm 13, we maintain three variables throughout the entire operation sequence: $maxOutDegree$, $maxNodeCount$ and $visited$. Implementing a shared $visited$ array as a global variable has the advantage of avoiding the need to rebuild the entire array after each individual search. This approach allows precisely resetting the nodes that were traversed during the DFS after the search has been performed. Considering that most often only one DFS is executed, resetting all nodes is redundant. Note that the re-solving in FindOptimal requires resetting $visited$ (Line 17) only after a complete traversal over all peak nodes, as previously discussed. The other two variables, $maxOutDegree$ and $maxNodeCount$, are used to maintain the current maximum outdegree and the count of peak nodes, respectively. To accommodate insertions and deletions, the algorithm utilizes a modified version of individual adjacency arrays, incorporating a strategy from Reinstädtler et al. [38] to store both outgoing and incoming edges for each node. Consequently, each adjacency array is divided into two segments to distinctly manage these two types of connections. Note that only the outgoing edges of a node actively contribute to the orientation. Figure 4.7 provides an illustrated example of the extended data structure.

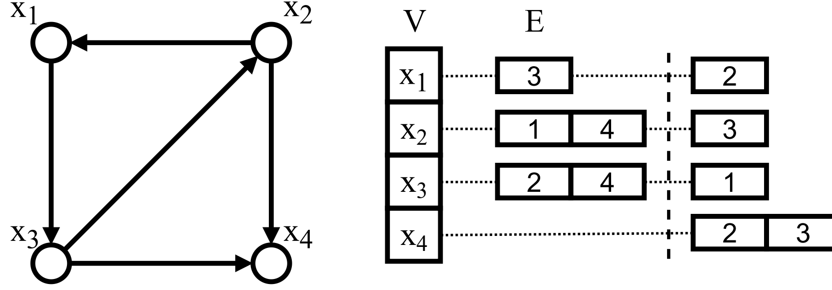


Figure 4.7: Extended individual adjacency arrays data structure applied to an orientation of the example graph G^1

Invariants

Since re-solving after every insertion or deletion is very time-consuming, we aim to minimize the number of searches for improving paths. To achieve this, we establish invariants that the algorithm must maintain after each update operation. These invariants guide our decisions on how to adjust the edge orientation to preserve optimality based on elaborated information. We discuss and analyze the explicit cases in detail later. First, we introduce the following invariants that have to hold true for the global variables both before and after each operation:

Invariant 1: $maxOutDegree = \max_{v \in V_{\mathcal{O}}}(\text{odeg}(v, \mathcal{O})) = \Delta(\mathcal{O})$

Invariant 2: $maxNodeCount = |\{v \in V_{\mathcal{O}} \mid \text{odeg}(v, \mathcal{O}) = maxOutDegree\}|$

Invariant 3: $\forall v \in V_{\mathcal{O}}$ with $\text{odeg}(v, \mathcal{O}) = maxOutDegree$: For every path $p = \langle v, \ldots, w \rangle$ in $A_{\mathcal{O}}$: $\text{odeg}(w, \mathcal{O}) \in \{maxOutDegree, maxOutDegree - 1\}$

To validate that maintaining these invariants ensures an exact solution, we require the next theorem similar to a statement by Venkateswaran [39].

Theorem 3: Given a graph $G(V, E)$, if an edge orientation $\mathcal{O}$ produces a maximum outdegree k such that $k = \lceil e_S/n_S \rceil$ for some node subset $S \subseteq V$, then k solves the edge orientation problem.

The proof is based on Theorem 2 by Venkateswaran [39] and is done analogous to Corollary 1 [39]. With the help of Theorem 3, we can now prove the preceding statement. The proof follows the demonstration of Venkateswaran for the correctness of his extremal orientation algorithm.

Theorem 4: Given a graph $G = (V, E)$, if invariant 1 and 3 are satisfied for some edge orientation $\mathcal{O}$, then the resulting maximum outdegree $\Delta_{\mathcal{O}}$ is optimal.

Algorithm 15: DynOptEO: Insertion

```
1  global variables: maxOutDegree = 0, maxNodeCount = 0,
     visited = [False] × |V|
2  procedure Insertion(O = (V_O, A_O), u, v):
3  W.l.o.g.: odeg(u, O) ≤ odeg(v, O)
4  A_O = A_O ∪ {(u, v)}
5  if odeg(u, O) = maxOutDegree then
6  |  if not FindPath(O, u, odeg(u, O)) then
7  |  |  maxNodeCount++
8  |  end if
9  |  reset visited
10 else if odeg(u, O) = maxOutDegree + 1 then
11 |  if FindPath(O, u, odeg(u, O)) then
12 |  |  maxNodeCount ++
13 |  else
14 |  |  maxOutDegree++
15 |  |  maxNodeCount = 1
16 |  end if
17 |  reset visited
18 end if
```

Proof. Theorem 4
Let $S = \{v \in V \mid \text{odeg}(v, \mathcal{O}) = \Delta_{\mathcal{O}}\}$ and $T = \{v \in V \mid \text{odeg}(v, \mathcal{O}) \leq \Delta_{\mathcal{O}} - 2\}$. Let U be the set of S and all nodes reachable from S by direct paths and $\bar{U} = V \setminus U$. Then $T \subseteq \bar{U}$ and there is no directed path from U to $\bar{U}$ because of invariant 1 and 3. Thus follows $e_U = \sum_{u \in U} \text{odeg}(u, \mathcal{O}) > n_U(\Delta_{\mathcal{O}} - 1)$ since at least one node in U has outdegree $\Delta_{\mathcal{O}}$. Therefore, $\Delta_{\mathcal{O}} = \lceil e_U / n_U \rceil$ and with Theorem 3 follows that $\Delta_{\mathcal{O}}$ is optimal. □

Insertion

At this point, we are prepared to examine the process for handling insertions. The pseudocode outlined in Algorithm 15 illustrates the specific cases. The algorithm consistently inserts edges $e = \{u, v\}$ with an orientation (u, v) such that $\text{odeg}(u, \mathcal{O}) \leq \text{odeg}(v, \mathcal{O})$. If this node subsequently has an outdegree less than $maxOutDegree$, the function can terminate immediately, as it is impossible to further improve the objective function. It is noteworthy that even when the outdegree is higher, the algorithm only requires one single DFS for each insertion. The last step is to update the global variables based on the outdegree of the respective node and the search result. We now provide proof for these assertions.

Proof. Algorithm 15: $\texttt{Insertion}(\mathcal{O} = (V_{\mathcal{O}}, A_{\mathcal{O}}), u, v)$
If $A_{\mathcal{O}}$ is empty, it is easily seen that invariant 1, 2 and 3 hold. We now proceed by induction. Let $\underline{\mathcal{O}}$ denote the state of $\mathcal{O}$ before and $\overline{\mathcal{O}}$ after the execution of Algorithm 15. The procedure is to assume that invariant 1, 2 and 3 hold true for $\underline{\mathcal{O}}$ and show that this is also true for $\overline{\mathcal{O}}$.
W.l.o.g. let odeg$(u, \underline{\mathcal{O}}) \leq$ odeg$(v, \underline{\mathcal{O}})$.

Let $\hat{\mathcal{O}}$ be the state of $\mathcal{O}$ after Line 4 so that $E_{\hat{\mathcal{O}}} = E_{\underline{\mathcal{O}}} \cup \{(u, v)\}$ and odeg$(u, \hat{\mathcal{O}}) =$ odeg$(u, \underline{\mathcal{O}}) + 1$
Since further actions depend on odeg$(u, \hat{\mathcal{O}})$, we are examining each case independently. Using invariant 1, we can directly determine that odeg$(u, \hat{\mathcal{O}}) \leq maxOutDegree + 1$.

Case 1: odeg$(u, \hat{\mathcal{O}}) < maxOutDegree$:
End of function: The function takes no further actions because the outdegree of every other node was not altered and there is no path from a node with maximum outdegree to u because of invariant 3. Therefore invariant 1, 2 and 3 hold.

Case 2: odeg$(u, \hat{\mathcal{O}}) = maxOutDegree$:
Line 6: Search for an improving path using FindPath.

Case 2.1: FindPath$(\hat{\mathcal{O}}, u,$ odeg$(u, \hat{\mathcal{O}}))$ found a path $p = \langle u, \ldots, w \rangle$ in $\hat{\mathcal{O}}$ with odeg$(w, \hat{\mathcal{O}}) \leq maxOutDegree - 2$. Therefore the edges in p have flipped which results in $E_{\overline{\mathcal{O}}} = (E_{\hat{\mathcal{O}}} \cup p^{-1}) \setminus p$
$\Rightarrow$ odeg$(u, \overline{\mathcal{O}}) = maxOutDegree - 1$ and odeg$(w, \overline{\mathcal{O}}) \leq maxOutDegree - 1$

End of function: odeg$(u, \overline{\mathcal{O}}) =$ odeg$(u, \underline{\mathcal{O}})$, while odeg$(w, \overline{\mathcal{O}})$ is still smaller than $maxOutDegree$. Therefore invariant 1 and 2 hold.
Assume there is a path $\tilde{p} = \langle x, \ldots, y \rangle$ in $\overline{\mathcal{O}}$ with odeg$(x, \overline{\mathcal{O}}) = maxOutDegree$ and odeg$(y, \overline{\mathcal{O}}) \leq maxOutDegree - 2$. Since only the outdegree of w, the insertion of (u, v) and flipping the edges in p^{-1} distinguishes $\underline{\mathcal{O}}$ from $\overline{\mathcal{O}}$, invariant 3 implies that $\tilde{p}$ must either have an edge with node w, contain (u, v) or share edges with p^{-1}. We are going to show that, based on this, we can find a path in $\underline{\mathcal{O}}$ that contradicts invariant 3.

Let z be the first node of an edge in $\tilde{p}$ that is also part of an edge in p^{-1}, resulting in $\tilde{p} = \langle x, \ldots, z, \ldots, y \rangle$ and $p = \langle u \ldots, z, \ldots, w \rangle$. Such a node exists because u and w are also part of edges in p^{-1}.
$\Rightarrow$ The sub-paths $\tilde{p}_1 := \langle x, \ldots, z \rangle$ of $\tilde{p}$ and $p_1 := \langle z, \ldots, w \rangle$ of p both share no edges with p^{-1} and $(u, v) \notin \tilde{p}_1$.

Case $z \neq u$ or $(u, v) \notin p : (u, v) \notin p_1$
$\Rightarrow$ There is a path $\langle x, \ldots, z, \ldots, w \rangle$ in $\underline{\mathcal{O}}$ which is a contradiction to invariant 3 ↯

Case $z = u$ and $(u, v) \in p$: $(u, v) \notin \tilde{p}$ because $(u, v) \notin E_{\overline{\mathcal{O}}}$ got flipped. Therefore $\tilde{p}$ has to have an edge with node w or share an edge with p^{-1}. Let $\tilde{z} \neq u$ be the first node of an edge in $\tilde{p}$ that is also part of an edge in p^{-1} (note that $w \in p^{-1}$), resulting in $\tilde{p} = \langle x, \ldots, u, \ldots, \tilde{z}, \ldots, y\rangle$ and $p = \langle u, v, \ldots, \tilde{z}, \ldots, w\rangle$.
⇒ The sub-paths $\tilde{p}_2 := \langle x, \ldots, \tilde{z}\rangle$ of $\tilde{p}$ and $p_2 := \langle \tilde{z}, \ldots, w\rangle$ of p both share no edges with p^{-1} and $(u, v) \notin \tilde{p}_2$, $(u, v) \notin p_2$
⇒ There is a path $\langle x, \ldots, \tilde{z}, \ldots, w\rangle$ in $\underline{\mathcal{O}}$ which is a contradiction to invariant 3 ↯

⇒ Invariant 3 holds

Case 2.2: FindPath$(\hat{\mathcal{O}}, u, \text{odeg}(u, \hat{\mathcal{O}}))$ did not find a path $p = \langle u, \ldots, w\rangle$ in $\hat{\mathcal{O}}$ with odeg$(w, \hat{\mathcal{O}}) \leq maxOutDegree - 2$.
End of function: The outdegree of u increased to $maxOutDegree$, therefore $maxNodeCount$ was increased by one. Since no other outdegree has been changed, invariant 1 and 2 hold.
Assume there is a path $\tilde{p} = \langle x, \ldots, y\rangle$ in $\overline{\mathcal{O}}$ with odeg$(x, \overline{\mathcal{O}}) = maxOutDegree$ and odeg$(y, \overline{\mathcal{O}}) \leq maxOutDegree - 2$. Since only the outdegree of u changes and (u, v) is inserted from $\underline{\mathcal{O}}$ to $\overline{\mathcal{O}}$, invariant 3 implies that $\tilde{p}$ must contain u, which is a contradiction to the fact that FindPath did not find a path $\langle u, \ldots, y\rangle$ ↯
⇒ Invariant 3 holds

Case 3: odeg$(u, \hat{\mathcal{O}}) = maxOutDegree + 1$:
Line 11: Search for an improving path using FindPath.

Case 3.1: FindPath$(\hat{\mathcal{O}}, u, \text{odeg}(u, \hat{\mathcal{O}}))$ found a path $p = \langle u, \ldots, w\rangle$ in $\hat{\mathcal{O}}$ with odeg$(w, \hat{\mathcal{O}}) \leq maxOutDegree - 1$. Therefore the edges in p have flipped which results in $E_{\overline{\mathcal{O}}} = (E_{\hat{\mathcal{O}}} \cup p^{-1}) \setminus p$
⇒ odeg$(u, \overline{\mathcal{O}})$ = odeg$(v, \overline{\mathcal{O}}) = maxOutDegree$
⇒ odeg$(w, \underline{\mathcal{O}}) = maxOutDegree - 1$ and odeg$(w, \overline{\mathcal{O}}) = maxOutDegree$

End of function: odeg$(u, \overline{\mathcal{O}})$ = odeg$(u, \underline{\mathcal{O}})$, while odeg$(w, \overline{\mathcal{O}})$ increased to $maxOutDegree$. Therefore $maxNodeCount$ increased by one, so invariant 1 and 2 hold.
Assume there is a path $\tilde{p} = \langle x, \ldots, y\rangle$ in $\overline{\mathcal{O}}$ with odeg$(x, \overline{\mathcal{O}}) = maxOutDegree$ and odeg$(y, \overline{\mathcal{O}}) \leq maxOutDegree - 2$. Since only the outdegree of w, the insertion of (u, v) and flipping the edges in p^{-1} distinguishes $\underline{\mathcal{O}}$ from $\overline{\mathcal{O}}$, invariant 3 implies that $\tilde{p}$ must either have an edge with node w, contain (u, v) or share edges with p^{-1}. We are going to show that, based on this, we can find a path in $\underline{\mathcal{O}}$ that contradicts invariant 3.

Let z be the last node of an edge in $\tilde{p}$ that is also part of an edge in p^{-1}, resulting in $\tilde{p} = \langle x, \ldots, z, \ldots, y \rangle$ and $p = \langle u, \ldots, z, \ldots, w \rangle$. Such a node exists because u and w are also part of edges in p^{-1}.
$\Rightarrow$ The sub-path $\tilde{p}_1 := \langle z, \ldots, y \rangle$ of $\tilde{p}$ does not share edges with p^{-1}.
Furthermore, v is not part of an edge in $\tilde{p}_1$, because otherwise there is a sub-path $\tilde{p}_2 := \langle v, \ldots, y \rangle$ of $\tilde{p}_1$, which also does not share any edges with p^{-1} and $(u, v) \notin \tilde{p}_2$, therefore contradicting invariant 3.
$\Rightarrow (u, v) \notin \tilde{p}_1$

Case $(u, v) \in p$: $p = \langle u, v, \ldots, z, \ldots, w \rangle$, the sub-path $p_1 := \langle v, \ldots, z \rangle$ of p does not share edges with p^{-1} and $(u, v) \notin p_1$.
$\Rightarrow$ There is a path $\langle v, \ldots, z, \ldots, y \rangle$ in $\underline{\mathcal{O}}$ which is a contradiction to invariant 3 ↯

Case $(u, v) \notin p$: The sub-path $p_1 := \langle u, \ldots, z \rangle$ of p does not share edges with p^{-1} and $(u, v) \notin p_1$.
$\Rightarrow$ There is a path $\langle u, \ldots, z, \ldots, y \rangle$ in $\underline{\mathcal{O}}$ which is a contradiction to invariant 3 ↯

$\Rightarrow$ Invariant 3 holds

Case 3.2: FindPath($\hat{\mathcal{O}}, u$, odeg($u, \hat{\mathcal{O}}$)) did not find a path $p = \langle u, \ldots, w \rangle$ in $\hat{\mathcal{O}}$ with odeg($w, \hat{\mathcal{O}}$) $\leq maxOutDegree - 1$
$\Rightarrow u$ is the only node with odeg($u, \overline{\mathcal{O}}$) $= maxOutDegree + 1$ and invariant 3 holds.
End of function: $maxOutDegree$ got increased by one and $maxNodeCount = 1$
$\Rightarrow$ invariant 1 and 2 hold

□

Deletion

We are now concluding the discussion of this algorithm by examining the deletion operation, which is more precisely available in the form of pseudocode in Algorithm 17. Deletion update operations are initiated by removing an edge $e = \{u, v\}$. If the outdegree of u is subsequently less than $maxOutDegree - 2$, the process can be promptly concluded as all invariants remain intact. In the case of $maxOutDegree - 1$, a peak node got decreased, and an update to $maxNodeCount$ is required. However, for the outdegree falling to $maxOutDegree - 2$, it is necessary to search for an improving path within the inverted edge orientation, essentially moving in a reverse direction. To do this, we introduce a function similar to FindPath, tasked with finding an inverse path to a node with an outdegree that is at least greater by two. The pseudocode for FindPathReversed is detailed in Algorithm 16. Successful identification and subsequent path flipping from a peak node also lead to a decrement of $maxNodeCount$.

Algorithm 16: DynOptEO: FindPathReversed

```
1  global variables: maxOutDegree = 0, maxNodeCount = 0,
     visited = [False] × |V|
2  procedure FindPathReversed(O = (V_O, A_O), u, d):
3  if visited[u] then return False
4  for e = (v, u) ∈ A_O do
5  |  if odeg(v, O) > d + 1 then
6  |  |  Flip(O, e)
7  |  |  return True
8  |  end if
9  end for
10 visited[u] = True
11 for e = (u, v) ∈ A_O do
12 |  if odeg(v, O) = d + 1 then
13 |  |  if FindPathReversed(O, v, d) then
14 |  |  |  Flip(O, e)
15 |  |  |  return True
16 |  |  end if
17 |  end if
18 end for
19 return False
```

If the count of peak nodes decreases to zero, we have reached the case where re-solving the orientation is mandatory. We are now demonstrating how this preserves the invariants, which is used for validating the optimality of Deletion later.

Lemma 5: If invariant 1 is true before the execution of FindOptimal($\mathcal{O}$), then invariant 1, 2 and 3 hold true afterwards.

Proof. Lemma 5
The proof falls naturally into three parts, one for every invariant. It is easily seen that invariant 2 holds, which is clear from Line 19 of FindOptimal (Algorithm 14). We are now demonstrating that invariant 1 is maintained during each loop iteration as well as at the end. In the beginning, invariant 1 holds because of the lemmas requirement. Consider an arbitrary step during the while loop. FindOptimal($\mathcal{O}$) checks potential improving paths in Line 8 for each node with an outdegree equal to $maxOutDegree$. By applying invariant 1, we can now conclude that these nodes are the ones with maximum outdegree.

Algorithm 17: DynOptEO: Deletion

```
1  global variables: maxOutDegree = 0, maxNodeCount = 0,
     visited = [False] × |V|
2  procedure Deletion(O = (V_O, A_O), u, v):
3  W.l.o.g.: (u, v) ∈ A_O // the edge is directed from u to v
4  A_O = A_O \ {(u, v)}
5  if odeg(u, O) = maxOutDegree − 1) then
6  |  maxNodeCount−−
7  else if odeg(u, O) = maxOutDegree − 2) then
8  |  if FindPathReversed(O, u, odeg(u, O)) then
9  |  |  maxNodeCount−−
10 |  end if
11 |  reset visited
12 end if
13 if maxNodeCount = 0 then
14 |  maxOutDegree − −
15 |  FindOptimal(O)
16 end if
```

Case 1: $\exists u \in V_{\mathcal{O}}$ with odeg$(u, \mathcal{O}) = maxOutDegree$:
FindPath$(\mathcal{O}, u,$ odeg$(u, \mathcal{O}))$ was unsuccessful
$\Rightarrow$ At Line 11: $all_paths_improved =$ False
$\Rightarrow$ End of while: $maxOutDegree$ keeps the same value because odeg$(u, \mathcal{O}) = maxOutDegree$.

Case 2: $\forall u \in V_{\mathcal{O}}$ with odeg$(u, \mathcal{O}) = maxOutDegree$:
FindPath$(\mathcal{O}, u,$ odeg$(u, \mathcal{O}))$ was successful
$\Rightarrow$ At Line 11: $\forall u \in V_{\mathcal{O}}$: odeg$(u, \mathcal{O}) < maxOutDegree$ and $all_paths_improved =$ True
$\Rightarrow$ At Line 15: $maxOutDegree$ is decreased
$\Rightarrow$ End of while: $maxOutDegree = \max_{v \in V_{\mathcal{O}}}($odeg$(v, \mathcal{O}))$ because any node can only be decreased by one in a single iteration

Thus, invariant 1 holds. Furthermore, since the while loop only terminates when no more improving paths can be found, invariant 3 follows directly, which completes the proof. □

We finally demonstrate that Deletion also maintains the invariants.

Proof. Algorithm 17: `Deletion`$(\mathcal{O} = (V_\mathcal{O}, A_\mathcal{O}), u, v)$
Let $\underline{\mathcal{O}}$ denote the state of $\mathcal{O}$ before and $\overline{\mathcal{O}}$ after the execution of Algorithm 17. Our proof starts with the assumption that invariant 1, 2 and 3 hold true for $\underline{\mathcal{O}}$ and show that this is also true for $\overline{\mathcal{O}}$.
W.l.o.g. let $(u, v) \in E_{\underline{\mathcal{O}}}$, i.e., the edge is oriented from u to v.

Let $\hat{\mathcal{O}}$ be the state of $\mathcal{O}$ after Line 4 so that $E_{\hat{\mathcal{O}}} = E_{\underline{\mathcal{O}}} \setminus \{(u, v)\}$ and odeg$(u, \hat{\mathcal{O}}) =$ odeg$(u, \underline{\mathcal{O}}) - 1$
Since further actions depend on odeg$(u, \hat{\mathcal{O}})$, we are examining each case independently. Using invariant 1, we can directly determine that odeg$(u, \hat{\mathcal{O}}) < maxOutDegree$.

Case 1: odeg$(u, \hat{\mathcal{O}}) < maxOutDegree - 2$:
End of function: The function takes no further actions because the outdegree of every other node was not altered and there is no path from a node with maximum outdegree to u because of invariant 3. Therefore invariant 1, 2 and 3 hold.

Case 2: odeg$(u, \hat{\mathcal{O}}) = maxOutDegree - 2$:
Line 8: Search for an improving path using FindPathReversed.

Case 2.1: FindPathReversed$(\hat{\mathcal{O}}, u,$ odeg$(u, \hat{\mathcal{O}}))$ found a path $p = \langle w, \ldots, u\rangle$ in $\hat{\mathcal{O}}$ with odeg$(w, \hat{\mathcal{O}}) = maxOutDegree$. Therefore the edges in p have flipped which results in $E_{\overline{\mathcal{O}}} = (E_{\hat{\mathcal{O}}} \cup p^{-1}) \setminus p$
$\Rightarrow$ odeg$(u, \overline{\mathcal{O}}) =$ odeg$(w, \overline{\mathcal{O}}) = maxOutDegree - 1$
Line 9: odeg$(w, \overline{\mathcal{O}})$ decreased, so $maxNodeCount$ is decreased by one.

Case 2.1.1: $maxNodeCount = 0$:
Line 14: Because the outdegree of each node could only have been decreased by one, invariant 2 implies that $maxOutDegree$ has to be decreased to match the equation of invariant 1.
Line 15: Execute FindOptimal$(\overline{\mathcal{O}})$. Lemma 5 $\Rightarrow$ Invariant 1, 2 and 3 hold

Case 2.1.2: $maxNodeCount \neq 0$:
End of function: w is the only node with a decrease in outdegree from $\underline{\mathcal{O}}$ to $\overline{\mathcal{O}}$ which is why $maxNodeCount$ is decreased in Line 9, while remaining greater than zero. Therefore invariant 1 and 2 hold.
Assume there is a path $\tilde{p} = \langle x, \ldots, y\rangle$ in $\overline{\mathcal{O}}$ with odeg$(x, \overline{\mathcal{O}}) = maxOutDegree$ and odeg$(y, \overline{\mathcal{O}}) \leq maxOutDegree - 2$. Since only the outdegree of w, the deletion of (u, v) and flipping the edges in p^{-1} distinguishes $\underline{\mathcal{O}}$ from $\overline{\mathcal{O}}$, invariant 3 implies that $\tilde{p}$ must either have an edge with node w or share edges with p^{-1}. We are going to show that, based on this, we can find a path in $\underline{\mathcal{O}}$ that contradicts invariant 3.

Let z be the last node of an edge in $\tilde{p}$ that is also part of an edge in p^{-1}, resulting in $\tilde{p} = \langle x, \ldots, z, \ldots, y \rangle$ and $p = \langle w, \ldots, z, \ldots, u \rangle$. Such a node exists because w is also part of an edge in p^{-1}.
$\Rightarrow$ The sub-paths $\tilde{p}_1 := \langle z, \ldots, y \rangle$ of $\tilde{p}$ and $p_1 := \langle w, \ldots, z \rangle$ of p both share no edges with p^{-1}.
$\Rightarrow$ There is a path $\langle w, \ldots, z, \ldots, y \rangle$ in $\underline{\mathcal{O}}$ which is a contradiction to invariant 3 ↯
$\Rightarrow$ Invariant 3 holds

Case 2.2: FindPathReversed$(\hat{\mathcal{O}}, u, \text{odeg}(u, \hat{\mathcal{O}}))$ did not find a path $p = \langle w, \ldots, u \rangle$ in $\hat{\mathcal{O}}$ with odeg$(w, \hat{\mathcal{O}}) = maxOutDegree$.
End of function: The only node which has changed between $\underline{\mathcal{O}}$ and $\overline{\mathcal{O}}$ is u with odeg$(u, \underline{\mathcal{O}}) =$ odeg$(u, \overline{\mathcal{O}}) + 1 = maxOutDegree - 1$ and there is no path from a node with maximum outdegree to u. Therefore invariant 1, 2 and 3 hold.

Case 3: odeg$(u, \hat{\mathcal{O}}) = maxOutDegree - 1$:
Line 6: odeg$(u, \overline{\mathcal{O}})$ decreased, so $maxNodeCount$ is decreased by one.

Case 3.1: $maxNodeCount = 0$:
We can now proceed analogously to case 2.1.1.

Case 3.2: $maxNodeCount \neq 0$:
End of function: The only node which has changed between $\underline{\mathcal{O}}$ and $\overline{\mathcal{O}}$ is u with odeg$(u, \underline{\mathcal{O}}) =$ odeg$(u, \overline{\mathcal{O}}) + 1 = maxOutDegree$ and there is no path from u to a node w with odeg$(w, \overline{\mathcal{O}}) \leq maxOutDegree - 2$ because of invariant 3. Therefore invariant 1, 2 and 3 hold.

□

Complexity Analysis

In the worst case, Insertion performs a search spanning the entire graph to find an improving path. Since the time complexity is equivalent to a DFS, the running time is bounded by $O(m)$. For Deletion, there is a possibility of having to resolve the graph using FindOptimal, resulting in the same worst-case running time $O(m^2)$ as the algorithm of Venkateswaran [39]. Note that these bounds represent worst-case scenarios, and in practice, the algorithm tends to operate more efficiently on average (see Section 5.3).

CHAPTER 5

Experimental Evaluation

In this chapter, we present and discuss the results of the experimental evaluation concerning the algorithms introduced in Chapter 4. We begin by outlining the structure and procedure for presenting the results. Next, we describe the hardware conditions of the machine used for the experiments. We then explain the setup and give a short introduction to performance profiles. This is followed by an overview of the instances used in the experiments as well as a brief description of the competitors used for comparison. Finally, we execute the proposed algorithms with different configurations, compare them to existing algorithms, and analyze the results.

5.1 Methodology

Structure

We deal with a large number of static and dynamic algorithms in our experiments. For this reason, we divide the experiments into three categories. In the first section, we are exclusively inspecting static algorithms for approximating the edge orientation. This is done by first examining the algorithms in smaller subsets based on similar approaches or an isolated parameter analysis. The evaluation starts with a group of greedy algorithms, which always process nodes completely, followed by those, that process edges individually. We then proceed with a parameter analysis for 2ApproxMix, which is followed by the algorithms that employ FOREST by Nagamochi and Ibaraki [31]. The most successful of these algorithms are then compared to provide a comprehensive review of their performance. This stepwise filtering method allows us to present less successful algorithms and delve deeper into those of significant interest due to their superior performance. The second category features the two dynamic algorithms. We start with DescBFS, exploring different combinations for its two parameters. This is followed by an analysis of the dynamic optimal algorithm, focusing particularly on the efficiency of insertion and deletion operations and the frequency

of function calls. To conclude this section, these dynamic algorithms are compared with those from the study of Borowitz et al. [7]. The entire evaluation is then concluded with an overall comparison between selected static and dynamic algorithms.

Hardware

All of the experiments were conducted on a machine equipped with an Intel(R) Xeon(R) Silver 4216 CPU and a base clock speed of 2.10GHz. It obtains 96GB of main memory, a 22MiB L3 Cache, and operates on Ubuntu 20.04.1 LTS as well as the Linux kernel version 5.4.0-169-generic. Each algorithm was implemented in C++ and compiled using g++ version 12.3.0 with full optimization enabled (-O3 flag).

Setup

In our experiments, each instance is run 10 times for static algorithms and 5 times for dynamic algorithms. We always calculate the geometric mean of the corresponding repetitions to determine the average running time on each instance. For running times exceeding 5 hours, we limit the execution to a single run per instance. The graphs are loaded from their respective files into main memory before timing begins, to prevent random file access times from affecting the results. This process is implemented to ensure a reliable measurement for the initialization. For static algorithms, we differentiate between the time it takes to load the graph sequence into the data structure required for the corresponding algorithm and the actual execution time. This distinction allows us to accurately measure the raw execution time as well as the total time, which also includes the loading (initialization) phase. This approach highlights how some algorithms, while benefiting from a more complex data structure, also incur time penalties during initialization.

Because dynamic algorithms continually update their data structure while processing a sequence, we include the initialization time for static algorithms when comparing both types. If only static algorithms are compared with each other, we primarily operate with the overall time as well. Results excluding initialization are specifically mentioned in the analysis if the comparison involves different data structures or if significant differences are observed. When only the running time including initialization is mentioned, it can be assumed that the raw execution time does not provide any significant additional insights. Additionally, all average values are calculated using the geometric mean by default, unless specified otherwise.

Performance Profiles

In order to present the measured results appropriately, we most often utilize *performance profiles* for plotting. Please refer to Dolan and Moré [13] for a more comprehensive discussion. This kind of presentation is widely used for displaying comparisons of optimization software in relation to certain objectives. In this section, the performance is assessed on

two key metrics: the running time of algorithms and the quality of solutions they generate. This results in a 2D graph where the x-axis represents a variable $\tau \geq 1$, starting at 1 and increasing continuously. On the other hand, the y-axis shows a fraction of instances. The performance of an algorithm is represented as a line, where a point $(\tau, p(\tau))$ on this line indicates the percentage $p(\tau)$ of instances on which the algorithm performs better than τ times the overall best performance on that instance. This allows us, for example, to find out the percentage of instances where the algorithm achieves the best solution among its competitors by looking at $\tau = 1$. Additionally, the first point with $p(\tau) = 1$ indicates the maximum factor τ by which this algorithm deviates from the best solution in any instance. It is important to note that these best results are not necessarily achieved by the same algorithm across different instances. Performance profiles offer a wide range of insights, particularly due to their relative nature, which makes them a valuable asset for comparisons.

Instances

The experiments were conducted using a dataset provided by Borowitz et al. [7]. The corresponding set of 87 large graphs consists of both static and truly dynamic instances and were collected from various backgrounds including numerical simulations and complex networks [4, 12, 17, 24, 29, 30, 35, 37]. Given that the algorithms discussed in this work are designed for undirected and unweighted graphs, we disregard the direction and weight of edges and remove any self-loops and parallel edges. Notably, only a subset of these graphs, specifically *amazon-ratings*, *dewiki*, *movielens10m*, and *wiki_simple_en*, originally include deletions. Henceforth, we use the terms *dynamic sequence* to refer to graph sequences including deletions as well as *static sequence* for excluded deletions. Static algorithms obtain the graph input sequence sorted by nodes, whereas dynamic algorithms consistently receive edges in random order.

Since the majority of instances did not feature any deletions which are essential for the the empirical evaluation of the DynOptEO algorithm, we extend the otherwise static sequences with further update operations. This is divided into two stages, the first of which includes both deletions and insertions. Therefore, a number of additional update operations are calculated by taking the maximum of either 1 000 or 10% of the number of edges. When adding a new update to the sequence, we store all previously deleted edges and then select randomly between insertion or deletion. There are only two exceptions to this procedure. The first is a complete graph (where every pair of nodes is connected) with an empty list of deleted edges, where a deletion is selected by default. Conversely, the second is a graph with an empty set of edges, where an insertion is necessary. A deletion is selected by randomly choosing and removing an existing edge of the graph, whereas an insertion is always selected from the set of formerly deleted nodes. It is important to note that this strategy only involves update operations concerning edges that have already been part of the original static sequence. As a result, important properties of that graph, such as planarity, are preserved during this process. In the second phase, all edges are removed until an empty graph occurs. The deleted edges are selected following the same procedure as in phase one.

In summary, we generate a truly dynamic graph sequence that includes an equal number of insertions and deletions, allowing a comparison between the overall running times generated by both operations. Additionally, the graph offers alternations between insertion and deletion updates, illustrating the interactions between the two operations. Statistics of both the original as well as the extended graphs can be found in Appendix Table A.1.

Competitors

This section briefly explains the algorithms that are used for comparison to the newly proposed ones. We already introduced an algorithm from the literature with the 2-approximation [2, 9, 21] in Section 4.2.1. The algorithms described below are thoroughly discussed in the paper by Borowitz et al. [7], where readers can find a comprehensive review of each algorithm. Pseudocode of every algorithm is nevertheless available in Appendix A.3.1 if needed. Also, most of the algorithms mentioned below perform a deletion by simply removing the edge without further actions. Accordingly, we primarily describe the actions performed during an insertion of $e = (u, v)$ in the following.

The Improving u-y-Path Search Algorithm(BFS) is based on a breadth-first search in the d-neighborhood of u to find an improving path. On discovery, the path is flipped, and the search concludes. Similarly, the Random Path Algorithm (RPath) also seeks an improving path in the d-neighborhood of u, but in this case, the paths are randomly chosen. If unsuccessful, the search is repeated up to r times. The main idea for the Descending Degrees Algorithm (DescDegrees) is always flipping the edge to the immediate neighbor with the lowest outdegree. We continue this process for the neighbor as long as the flipping operation remains successful. Note that pruning methods, as described in Section 1, are implemented in these three algorithms as well. K-Flips by Berglin and Brodal [5] operates globally, flipping k edges (x, y) where x has maximum outdegree. This is performed after each insertion and deletion. Furthermore, the algorithm by Brodal and Fagerberg [8] (BroFag) checks whether the outdegree of u exceeds a variable $\tilde{\alpha}$ after inserting the edge. If so, the algorithm continuously selects a node w with a higher outdegree than $\tilde{\alpha}$ and flips all of its out-going edges until no such node can be found. The variable $\tilde{\alpha}$ can be an upper bound for the arboricity α, but in the adaptive variant of the implementation, $\tilde{\alpha}$ is initialized with 1 and increased by $\tilde{\alpha} = \beta\tilde{\alpha}$ with factor $\beta \in [1, 2]$ if the number of re-orientations becomes too large. In such cases, a rebuild is performed and the bound for re-orientations is increased by $\tilde{\alpha} + 1$. Finally, the naive strategy simply orients each inserted edge starting from the node with the lower outdegree.

We also use parameters that were selected by Borowitz et al. [7] and give the best solution quality for the respective algorithms. This results in the following set: BFS_{20}, $\text{RPath}_{d=50,r=10}$, DescDegrees, $K\text{-Flips}_{50}$, $\text{BroFag}_{1.01}$ and Naive.

5.2 Static Results

All average results for static algorithms, regarding running time and solution quality, are provided in Table 5.1. This includes the average running times with and without consideration of the initialization. If these specifics are not explicitly mentioned in the text, they can be referenced there for further clarification. Performance profiles for static algorithms are given in Figure 5.1 and Figure 5.2, whereas alternative diagrams comparing the running time with and without initialization are available in Appendix A.1 (Figure A.1 and Figure A.2).

Algorithms for Completely Processing Nodes

This section compares various algorithms whose strategies revolve around selecting a node and then orienting all of its edges. All algorithms included in this category are MinDegNode and MaxDegNode from Section 4.2.1 as well as EdgeProgDeg and EdgeInitDeg from Section 4.2.2. At this stage, we also integrate a comparison between adjacency arrays and lists, hence the inclusion of MinDegNodeList in this group. The insights derived from this provide us with information about which representation is the more suitable choice for these static algorithms. Performance profiles for the solution quality are illustrated in Figure 5.1 (a), while Figure 5.1 (b) depicts the running time.

Beginning with the two different data structures, it is obvious that the solution quality of the MinDegNode algorithm remains consistent between the version utilizing adjacency arrays and the one using lists. Although one might think that the ability to remove edges completely and not having to revisit them would benefit the version with adjacency lists, it does not seem to outweigh the higher cost of the data structure itself compared to adjacency arrays. This is prominently confirmed in Figure 5.1 (b) where MinDegNode is on average 4.1 times faster than MinDegNodeList.

We proceed by evaluating the performance in terms of solution quality for the remaining algorithms. MinDegNode evidently stands out as the more effective option among these algorithms, securing 84.3% of the best instances in this comparison. Notably, MinDegNode does not exceed the limit of $\tau = 2$ as must be the case for a 2-approximation, but this property also holds for EdgeProgDeg. The remaining two algorithms generally perform worse, whereby there are instances for both with particularly bad results. For example, MaxDegNode reaches a peak τ factor of 4.5, and EdgeInitDeg has a result that is 9 times worse than the best solution. Summarizing the quality aspect, MinDegNode outperforms EdgeProgDeg, EdgeInitDeg and MaxDegNode clearly by 33.6%, 49.7% and 51.9%, respectively.

The situation is quite the opposite when it comes to the running time. In this context, the MaxDegNode algorithm is a factor 1.1, 1.1, 1.6, faster on average than EdgeInitDeg, EdgeProgDeg and MinDegNode, respectively. However, while there might be quicker algorithms in this lineup, there is no real competitor for the solution quality of MinDegNode.

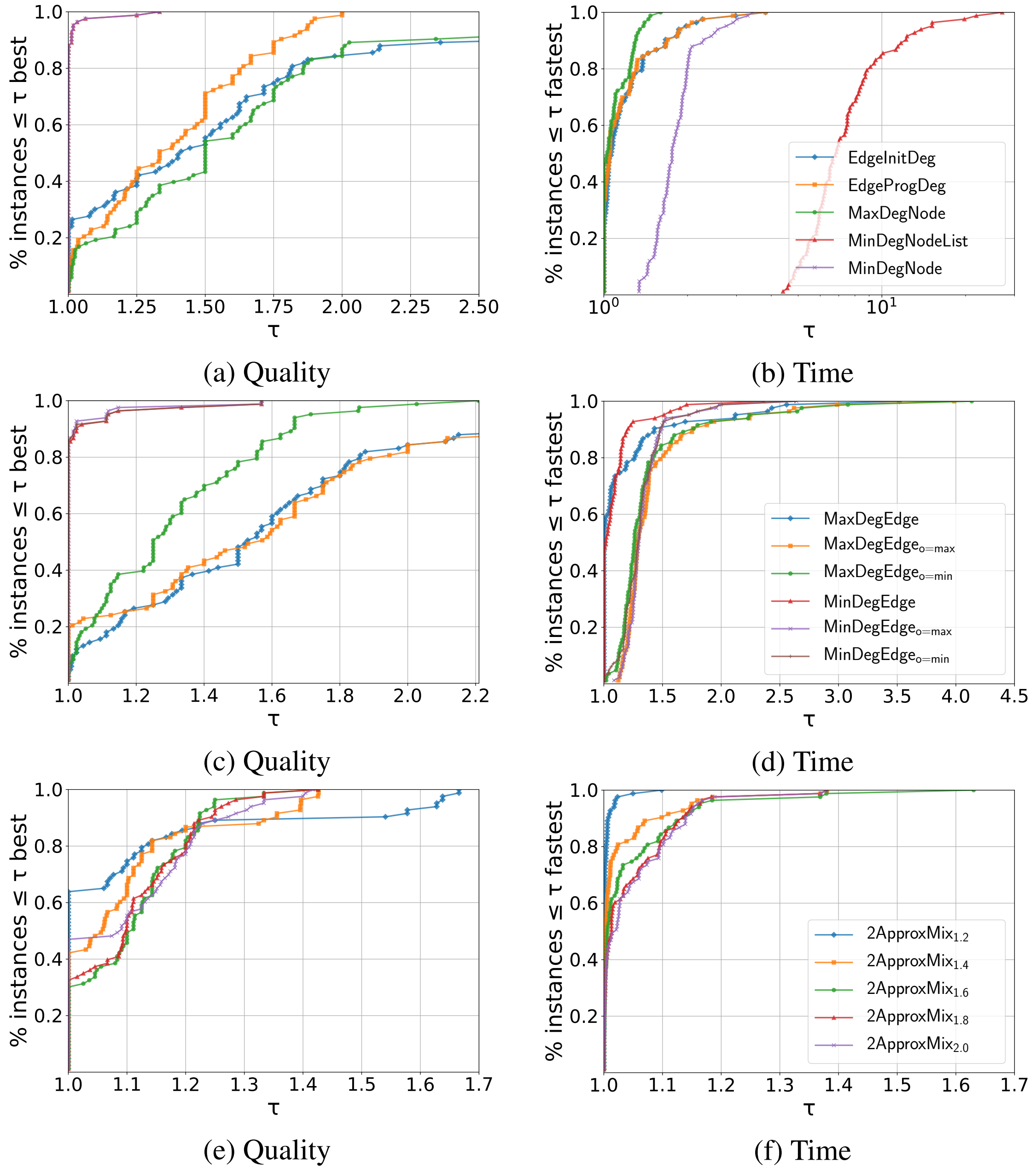


Figure 5.1: Performance profiles illustrating solution quality on the left and running time on the right side. The rows refer to the categories of algorithms for completely processing nodes, algorithms for processing individual edges, and 2ApproxMix (factor σ) (top to bottom). The dataset consists of static sequences (see Table A.1).

Running Time Excluding Initialization. When considering the raw execution time, the disparity between the two versions of MinDegNode utilizing different data structures is reduced significantly. This indicates a more expensive initialization process for individual

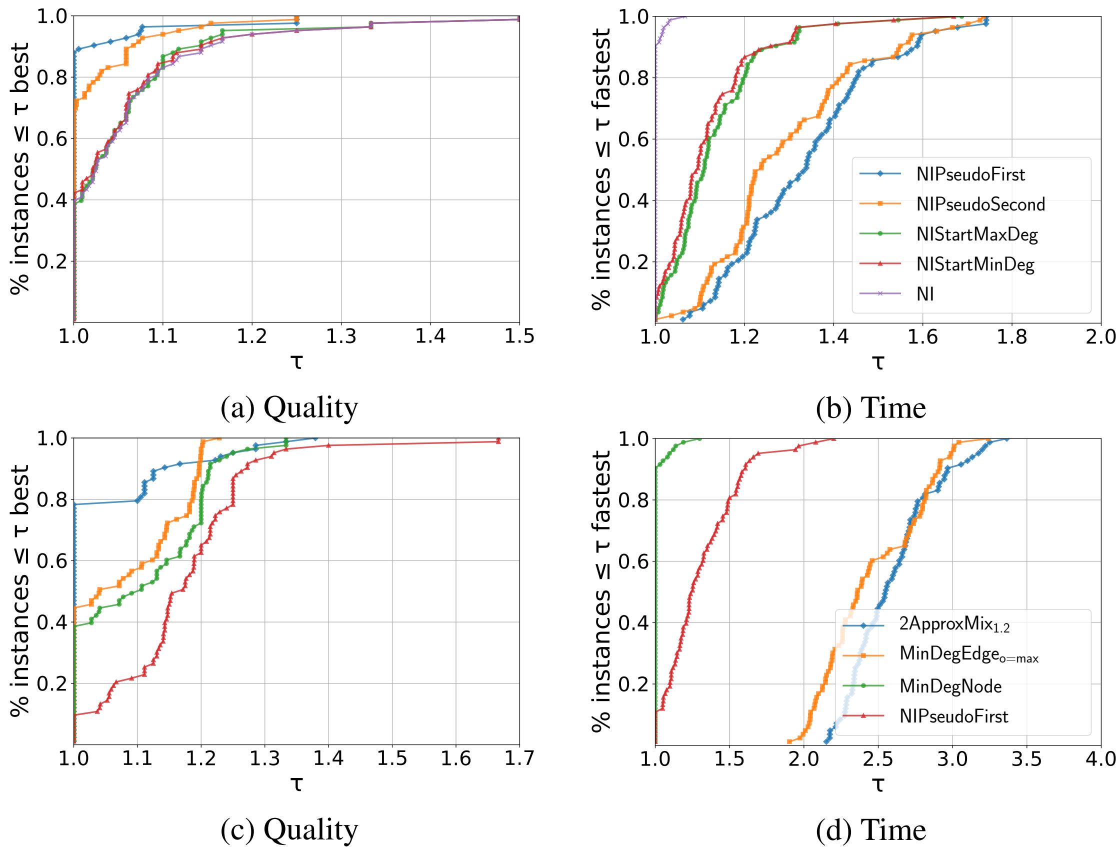


Figure 5.2: Performance profiles illustrating solution quality on the left and running time on the right side for algorithms based on FOREST [31] and the comparison of best static algorithms (top to bottom). The dataset consists of static sequences (see Table A.1).

adjacency arrays, caused by a high array count. Nonetheless, MinDegNode is still twice as fast even in this comparison. Moreover, both EdgeProgDeg and EdgeInitDeg, which rely on an inverse edge array, outperform MaxDegNode once initialization costs are excluded. Notably, EdgeProgDeg now leads as the fastest by factors of 1.3, 2, and 4 compared to EdgeInitDeg, MaxDegNode, and MinDegNode, respectively.

Algorithms for Processing Individual Edges

This section primarily concentrates on the algorithms MinDegEdge and MaxDegEdge from Section 4.2.3, with a specific focus on analyzing the effects of different strategies for initializing the graph data structure. The initialization method of inserting nodes into the data structure in ascending order of degree is characterized by $o = min$. A descending order, on the other hand, is indicated by $o = max$. Otherwise, nodes are inserted in the order they were stored, i.e., after the index. The performance profiles for solution quality are shown in Figure 5.1 (c), while the execution time is presented in Figure 5.1 (d).

The initialization order for nodes appears to have a minimal impact on the performance of the MinDegEdge algorithm, as the average solution quality barely differs at all. Nevertheless, both MinDegEdge$_{o=max}$ and MinDegEdge$_{o=min}$ demonstrate slight improvements in this regard. In terms of speed, MinDegEdge predictably leads, benefiting from its simpler initialization process.

A similar pattern in terms of running time can also be observed for the MaxDegEdge algorithm, as Figure 5.1 (d) illustrates. This is not unexpected, as we are observing the same initialization strategies, but for two different algorithms. The situation differs regarding quality, with MaxDegEdge$_{o=min}$ demonstrating significantly better performance than the other two. In particular, both of these reach a τ value of 9.2, corresponding to a similarly poor result. Besides, MaxDegEdge$_{o=max}$ is on average slightly better than the standard variant and additionally yields the highest percentage of best instances with 19% out of these three.

In the overall comparison regarding quality, MinDegEdge consistently outperforms the MaxDegEdge strategy in all configurations. Specifically, MinDegEdge$_{o=max}$ achieves on average 0.3%, 0.4%, 26.3%, 55.9% and 57.4% better solutions than MinDegEdge$_{o=min}$, MinDegEdge, MaxDegEdge$_{o=min}$, MaxDegEdge$_{o=max}$ and MaxDegEdge, respectively. Additionally, the common variant of MinDegEdge is also a factor 1.03, 1.2, 1.2, 1.3, 1.3 faster than MaxDegEdge, MinDegEdge$_{o=min}$, MinDegEdge$_{o=max}$, MaxDegEdge$_{o=min}$, MaxDegEdge$_{o=max}$. As solution quality is our primary concern, we are prioritizing MinDegEdge$_{o=max}$ for further comparisons. Note that MinDegEdge, being a Pareto configuration, can also be a viable option, especially if time is a critical factor.

Running Time Excluding Initialization. Given that the different configurations are based exclusively on the initialization, the actual execution time aligns with those of the common variants. This can be observed in Table 5.1, where the running times for MinDegEdge$_{o=min}$, MinDegEdge$_{o=max}$ and MinDegEdge are essentially identical without the initialization costs. The same also applies to MaxDegEdge.

Parameter Analysis for 2ApproxMix

Figure 5.1 (e) illustrates the performance profiles for quality, whereas Figure 5.1 (f) displays the running time of 2ApproxMix (see Section 4.2.4) for various parameters. We selected $\sigma \in \{1.2, 1.4, 1.6, 1.8, 2\}$, representing factors for the 2-approximation obtained by the MinDegNode algorithm and designated the corresponding configuration 2ApproxMix$_\sigma$.

In terms of best results relative to the other configurations, the algorithm appears to thrive with factors near the extremes. For instance, according to Figure 5.1 (e), it achieves percentages of 63.9% for $\sigma = 1.2$ and 47% for $\sigma = 2$. On the other hand, the fewer suboptimal results are all the worse for it. This is prominently reflected by a flattened performance curve, especially in the case of 2ApproxMix$_{2.0}$. As the values approach the center of the

set, the corresponding curve increasingly tends to adopt a more vertical orientation. This is particularly evident for the factor $\sigma = 1.6$ with 30.1% of best cases and no outcome worse than 1.5 times the optimal solution for the given set of graphs.

As far as the average running time is concerned, it appears to decrease steadily as the factors become smaller, which is illustrated in Figure 5.1 (f). This is understandable, as a smaller factor allows more edges to be oriented directly. However, since 2ApproxMix$_{1.2}$ is only 1.05 times faster than the slowest configuration, these differences are relatively minor and therefore can be considered negligible. It is not immediately evident from Figure 5.1 (e) which parameter yields the best performance. Nonetheless, the average quality performance offers a clear distinction with parameter $\sigma = 1.2$ surpassing $\sigma = 1.4$ by 0.3%, $\sigma = 2.0$ by 1.2%, $\sigma = 1.8$ by 1.4% and $\sigma = 1.6$ by 1.7%. It is important to note that the average is calculated using the geometric mean, which assigns less weight to single weak performances. For instance, the arithmetic mean actually favors $\sigma = 2.0$ over $\sigma = 1.2$ by 11.3%. With this method, however, 2ApproxMix$_{1.2}$ is actually the only Pareto configuration of this set, delivering the best performance for both objectives. Due to this conclusion, only this algorithm is used for further comparison in the following.

Algorithms Based on FOREST by Nagamochi and Ibaraki [31]

Figure 5.2 (a) presents the performance profiles for the algorithms NI, NIStartMaxDeg, NIStartMinDeg, NIPseudoFirst and NIPseudoSecond in terms of quality, while Figure 5.2 (b) displays their running time profiles. All of these algorithms are detailed in Section 4.2.5. The first thing to point out is the similarity of the solution quality of NI, NIStartMaxDeg and NIStartMinDeg. These are all variants of the same algorithm but represent different strategies for choosing a starting node. These approaches include selecting the node by index, which depends on the order in which nodes are stored within the graph structure, as well as selecting nodes with the highest or lowest degree, respectively. However, as Figure 5.2 (a) shows, there are no significant disparities in the quality profile when choosing a different initial node. Even when this was tested for each specific node, not a single case revealed a noteworthy deviation. Therefore it can be concluded that the starting node selection does not significantly affect the resulting quality and is hence not a suitable option for enhancing the algorithm.

In any case, the approach of approximating the pseudoarboricity does result in a notable improvement. As illustrated in Figure 5.2 (a), orienting the first edge of a new spanning tree away from the root node evidently yields the best solution quality in this comparison, achieving 88% of the best instances compared to 40% for NI. Even doing this with the second edge already demonstrates a decrease with 69%, although it is nonetheless an improvement to the forest approach.

When examining the running time in Figure 5.2 (b), it is apparent that a better performance quality mostly comes with the expense of additional running time. For instance, NIPseudoFirst yields the best average quality, outperforming NIPseudoSecond, NIStartMaxDeg, NIStartMinDeg and NI by 0.8%, 4.1%, 4.1% and 4.4%, respectively.

Simultaneously, NI is faster by a factor of 1.1, 1.1, 1.3 and 1.3 than NIStartMinDeg, NIStartMaxDeg, NIPseudoSecond and NIPseudoFirst. It is to be expected, that NI stands out as slightly faster compared to the variants that require more complex strategies. In summary, it can be concluded that the NIPseudoFirst algorithm produces the best results among this group when disregarding the minor running time differences.

Running Time Excluding Initialization. The only variation between both time measurements is the repositioning of NIPseudoSecond and NIPseudoFirst, now ranked as the second and third fastest algorithms after NI, which still maintains its lead by 9% to both. This adjustment seems logical, considering that edges must be able to orient in both directions which requires the initialization of an inverse edge array.

Comparison of Static Algorithms

This study now turns to a comparative analysis of the most effective algorithms from Section 4.2 in terms of quality, selected from each preceding group. The proposed candidates include: MinDegNode, NIPseudoFirst, MinDegEdge$_{o=max}$ and 2ApproxMix$_{1.2}$. The quality performance profiles are depicted in Figure 5.2 (c), while Figure 5.2 (d) showcases the corresponding time profiles.

Starting with the most inferior quality performance, the NIPseudoFirst algorithm significantly falls behind, with a maximum outdegree exceeding that of the best algorithm, 2ApproxMix$_{1.2}$, by 12.6%. Additionally, while the results of MinDegEdge$_{o=max}$ and MinDegNode appear similar, the former consistently outperforms the latter. This is evidenced by an average quality being higher than 3.9% and 6.1% compared to 2ApproxMix$_{1.2}$, respectively. It is also important to highlight that 2ApproxMix$_{1.2}$ also achieves 79.5% of the best performances. An interesting observation is that none of the competitors exceeds $\tau = 1.7$ for any instance. This is particularly evident for the MinDegEdge$_{o=max}$ algorithm, which never deviates more than 1.23 times from the best value of this set.

Upon examination of the running time depicted in Figure 5.2 (d), it becomes evident that there are notable discrepancies between the four algorithms. Notably, MinDegNode exhibits a remarkably faster performance than NIPseudoFirst, MinDegEdge$_{o=max}$ and 2ApproxMix$_{1.2}$ by factors of 1.3, 2.4, and 2.6. However, these results can be expected, as they exactly mirror the increasing complexity of the algorithms.

Running Time Excluding Initialization. Apart from the use of an inverse edge array, NIPseudoFirst proceeds in similar steps as MinDegNode, so the running time is alike when ignoring the initialization. In addition, the difference between the two slower algorithms increases too, as the sorting of nodes during the initialization of MinEdge is now omitted.

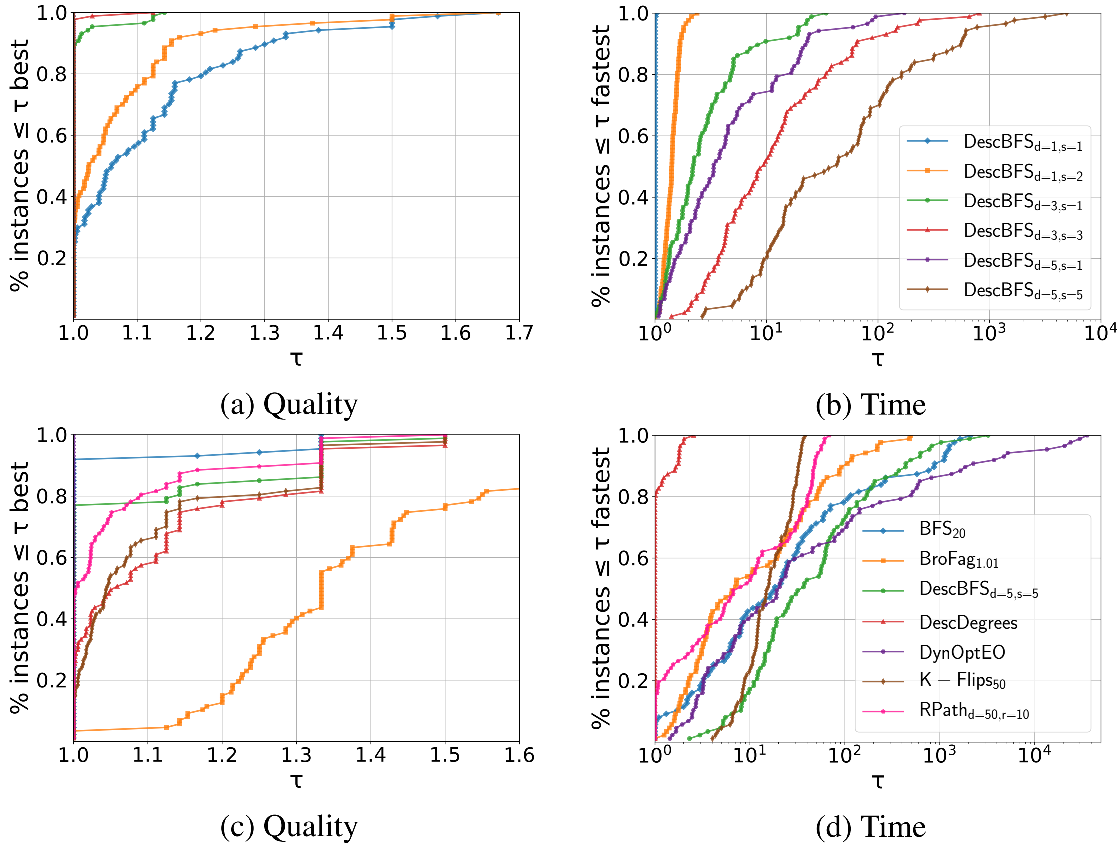


Figure 5.3: Performance profiles illustrating solution quality on the left and running time on the right side for the parameter analysis of DescBFS and the comparison of best dynamic algorithms (top to bottom). The dataset consists of all instances (see Table A.1).

In summary, if the primary objective (among linear-time algorithms) is achieving the highest average performance quality, the empirical evaluation tends to favor the 2Approx-Mix$_{1.2}$ algorithm. But MinDegEdge$_{o=max}$ might also be considered a viable choice given its low rate of poor results. However, both options come with the disadvantage of increased running times, a domain where MinDegNode prevails.

5.3 Fully Dynamic Results

The average results for dynamic algorithms on all instances are presented in Table 5.2. Additionally, Table 5.1 provides results exclusively for sequences without deletions. Throughout this section, we analyze all instances, as our focus is solely on dynamic algorithms. Please refer to Table A.1 for a comprehensive list.

Paramter Analysis for DescBFS

The performance profiles for the parameter study of DescBFS (see Section 12) are depicted in Figure 5.3, with panel (a) illustrating quality and panel (b) showing the running time. The algorithm (see Section 4.3.1) is evaluated based on two parameters, the maximum depth of each breath-first search as well as the maximum number of descending steps. Both of these parameters are exhaustive, with higher values leading to longer running times while at the same time improving the solution quality. We performed tests for different values of depth $d \in \{1, 2, 3, 5\}$ and number of steps $s \in \{1, 2, 3, 5\}$. This results in a total number of 16 different combinations, which is why we calculated the Pareto configurations based on average quality and running time. This left a quantity of 6 combinations: $(d, s) \in \{(1, 1), (1, 2), (1, 3), (2, 1), (3, 1), (5, 1)\}$. Apart from depth 1, every remaining combination has $s > d$, indicating that depth yields more influence on the quality than the amount of descending steps. Especially considering that any combination with $d = 5$ yields a better average solution than all the others. On this set of graphs, $(5, 1)$ being as a simple BFS_5 achieves the same average quality as the other configurations with depth five. To appropriately study the increase in time, we also examine $(5, 5)$ and $(3, 3)$ in addition to the Pareto configurations. In order to present the algorithms properly, we select representatives from both the most expensive and fastest algorithms as well as from the center. Accordingly, the choice of the more recourse-intensive algorithms falls on $\text{DescBFS}_{d=5,s=5}$ and $\text{DescBFS}_{d=5,s=1}$, with $\text{DescBFS}_{d=3,s=3}$ and $\text{DescBFS}_{d=3,s=1}$ representing the middle, and $\text{DescBFS}_{d=1,s=2}$ as well as $\text{DescBFS}_{d=1,s=1}$ as the fastest variations.

As anticipated, the fastest algorithm is $\text{DescBFS}_{d=1,s=1}$ with an average running time of 0.067, but it also yields the worst quality results in return. It outperforms $\text{DescBFS}_{d=1,s=2}$, $\text{DescBFS}_{d=3,s=1}$, $\text{DescBFS}_{d=5,s=1}$, $\text{DescBFS}_{d=3,s=3}$, $\text{DescBFS}_{d=5,s=5}$ by a factor of 1.4, 2.8, 4.6, 11.7, and 44.5, respectively. On the other side of the spectrum, algorithm $\text{DescBFS}_{d=5,s=5}$ is the most expensive, yet it achieves the highest average quality result, matching $\text{DescBFS}_{d=5,s=1}$ with a score of with 19.186. They therefore surpass $\text{DescBFS}_{d=3,s=3}$, $\text{DescBFS}_{d=3,s=1}$, $\text{DescBFS}_{d=1,s=2}$, and $\text{DescBFS}_{d=1,s=1}$ in terms of solution quality by 0.3%, 0.7%, 6.7%, and 11.2%, respectively.

Dynamic Optimal Edge Orientation Algorithm

We are now examining the behavior of the dynamic optimal algorithm from Section 4.3.2. As a dynamic algorithm, DynOptEO consists of two functions: Insertion and Deletion. Since, in contrast to DescBFS, deletions do not necessarily only remove edges, we analyze the two functions in relation to each other in more detail. For insertions, either FindPath is used to search for a single improving path or the node is removed directly. In the case of deletions, either the node is simply removed or a reverse search is conducted for an improving path, and in some cases, the graph is additionally solved to optimality. As a result, the execution time of a single operation naturally varies greatly, which is why the overall running time heavily depends on the cost of expensive functions as well as how often

they are called. To investigate this, we turn to Figure 5.4, which shows the proportions of FindPath, FindPathReversed and FindOptimal per executed update operation. Note that we are only considering extended graphs so that the number of deletions corresponds to the number of insertions. A first observation reveals that FindPath is called three times more often than FindPathReversed and even 3 871 times the amount of FindOptimal on average. It is also interesting to note that the proportions of the functions evidently distribute fairly well across the total number of update operations. This indicates that the absolute number of edges does not have a major influence on the percentage of expensive function calls. If anything, for all three functions, a slight decrease can be observed for increasing update counts, which is particularly prominent for FindOptimal. However, this is not a surprise, as the number of edge cases relevant to optimality declines as the overall size of the graph increases. If we now consider Figure 5.5, it can also be noticed that the average duration of an insertion or deletion is also slightly increasing in general. This indicates that although there are fewer expensive calls, these are much more time-consuming, enough to at least compensate for their lower numbers. Surprisingly, the running times of insertions and deletions are quite similar with Deletion being on average only 19% slower than Insertion over all graphs. This evidently shows that the infrequent calls to solve the graph completely combined with FindPathReversed have a similar impact on the running time as the numerous individual searches for improving paths for insertions. However, deleting tends to be more expensive on average than inserting, especially for a large number of edges. Therefore, it is possible that this observation applies specifically to the graph sizes used in this work.

Comparison of Dynamic Algorithms

This section compares the two dynamic algorithms DescBFS and DynOptEO from Section 4.3 with selected approaches BFS_{20}, $\text{RPath}_{d=50,r=10}$, DescDegrees, K-Flips_{50}, and $\text{BroFag}_{1.01}$ (see Section 5.1 for a brief explanation). We exclude Naive to improve clarity and focus, as it is more suitable for the overall comparison including static algorithms in Section 5.4. Performance profiles for this comparison are available in Figure 5.3 (c) for quality as well as in Figure 5.3 (d) for running time. We choose $\text{DescBFS}_{d=5,s=5}$ for the performance profiles, as $\text{DescBFS}_{d=5,s=1}$ closely resembles BFS_5 and thus does not adequately represent the algorithm.

Our initial focus is on solution quality, where DynOptEO consistently delivers the best result on every instance, as expected from an optimal algorithm. However, $\text{DescBFS}_{d=5,s=5}$ also performs quite well, being only further surpassed by BFS_{20} and $\text{RPath}_{d=50,r=10}$, achieving on average 3.7% and 0.2% better solutions, respectively. Furthermore, with 77% of the best results, it even achieves the optimal solution more frequently than $\text{RPath}_{d=50,r=10}$ with 47.1%. However, it is mainly the configurations of DescBFS with higher depth that demonstrate particularly strong results, suggesting that the number of steps does not significantly impact the performance.

This is particularly evident when analyzing the running time in Figure 5.3 (d). For this objective, $\text{DescBFS}_{d=5,s=5}$ performs particularly poorly, being even worse than DynOptEO, which is on average 13.3% faster. However, note that DynOptEO is significantly slower for certain instances and consistently outperformed by every other competitor. Consequently, DescDegrees emerges as the fastest option, surpassing $\text{DescBFS}_{d=5,s=1}$, $\text{RPath}_{d=50,r=10}$, $\text{BroFag}_{1.01}$, K-Flips_{50}, BFS_{20}, DynOptEO and $\text{DescBFS}_{d=5,s=5}$ by factors of 4.2, 7.1, 9.8, 14.3, 20.5, 35.6, and 40.3, respectively. While $\text{DescBFS}_{d=5,s=1}$ is faster than most algorithms, it also largely corresponds to BFS_5. Therefore, we conclude that in most cases, a BFS with higher depth is preferable to DescBFS to avoid incurring high costs for additional steps.

5.4 Overall Comparison

This chapter concludes with a comparison between static and dynamic edge orientation algorithms. As these are intensively discussed individually in Section 5.2 and Section 5.3, this chapter only highlights important discoveries between the two types as well as representatives from the current literature (see Section 5.1 for a brief explanation). Furthermore, performance profiles for a selected set of algorithms are presented in Figure 5.6 with panel (a) displaying quality and panel (b) illustrating the running time.

The dynamic algorithm most suitable for comparison with static algorithms is Naive, as it employs a similarly simple strategy. This is evident when comparing their solution quality, as Naive falls between the static algorithms in terms of ranking. Accordingly, $\text{2ApproxMix}_{1.2}$ achieves, on average, 3.9%, 4.4%, 6.1%, 11.6%, and 12.6% better solutions than $\text{MinDegEdge}_{o=max}$, Naive, MinDegNode, $\text{BroFag}_{1.01}$, and NIPseudoFirst, respectively. In particular, the Naive strategy is relatively similar to MinDegEdge, except that it operates with incomplete information during graph construction, whereas MinDegEdge already has knowledge of all final edges. Additionally, we observe that $\text{BroFag}_{1.01}$ ranks below the three static algorithms and only outperforms NIPseudoFirst in terms of solution quality. Despite the static algorithms lagging behind the dynamic ones in terms of quality, they exhibit significantly faster performance. Correspondingly, MinDegNode is faster than Naive, NIPseudoFirst, $\text{MinDegEdge}_{o=max}$, and $\text{2ApproxMix}_{1.2}$ by factors of 1.25, 1.3, 2.4, 2.6. Even $\text{2ApproxMix}_{1.2}$ is still a factor 4.3 quicker than the next fastest dynamic algorithm DescDegrees.

Running Time Excluding Initialization. When excluding the initialization for static algorithms, NIPseudoFirst also outperforms Naive by a factor of 1.7, as can be observed in Figure A.3. While $\text{MinDegEdge}_{o=max}$ and $\text{2ApproxMix}_{1.2}$ reduce the gap considerably, Naive remains 1.2 and 1.7 times faster, respectively.

Optimal Solutions. Since comparing the solution quality of an algorithm with DynOptEO is equivalent to examining the frequency of optimal solutions, we are also presenting some interesting results in this context. For a detailed overview, please refer to Table 5.3. While Naive fails to achieve any optimal solutions, the static algorithms still perform well. Specifically, NIPseudoFirst achieves 4.8%, MinDegNode 7.2%, 2ApproxMix$_{1.2}$ 8.4% and MinDegEdge$_{o=max}$ 12%, all surpassing BroFag$_{1.01}$ with 3.6%. Moreover, DescBFS$_{d=5,s=5}$ is directly behind BFS$_{20}$ with 75.9% of optimal solutions compared to 91.6% on static sequences.

Table 5.1: Average performance of all algorithm configurations over all static sequences (see Table A.1), sorted by quality ($\Delta_{\mathcal{O}}$) in ascending order. Column 3 (Avg t [s]) displays average running times including the initialization of the graph data structure. For static algorithms, the time without initialization (no init.) is presented in column 4. Pareto optimal configurations, determined by average quality and running time including initialization, are highlighted in grey. The average is calculated using the geometric mean.

Algorithm	Avg $\Delta_{\mathcal{O}}$	Avg t [s]	Avg t [s] *(no init.)*
DynOptEO	17.490	2.484	-
BFS$_{20}$	17.876	1.473	-
RPath$_{d=50,r=10}$	18.518	0.519	-
DescBFS$_{d=5,s=1}$	18.573	0.293	-
DescBFS$_{d=5,s=5}$	18.573	2.640	-
DescBFS$_{d=3,s=3}$	18.606	0.716	-
DescBFS$_{d=3,s=1}$	18.697	0.175	-
K-Flips$_{50}$	19.341	0.982	-
DescDegrees	19.485	0.068	-
DescBFS$_{d=1,s=2}$	19.767	0.088	-
DescBFS$_{d=1,s=1}$	20.456	0.062	-
2ApproxMix$_{1.2}$	21.829	0.041	0.033
2ApproxMix$_{1.4}$	21.893	0.042	0.034
2ApproxMix$_{2.0}$	22.084	0.043	0.035
2ApproxMix$_{1.8}$	22.141	0.043	0.035
2ApproxMix$_{1.6}$	22.191	0.043	0.035
MinDegEdge$_{o=max}$	22.691	0.039	0.024
MinDegEdge$_{o=min}$	22.764	0.038	0.024
MinDegEdge	22.775	0.032	0.024
Naive	22.789	0.020	-
MinDegNode	23.165	0.016	0.012

Continuation on the next page

Table 5.1: (Continuation)

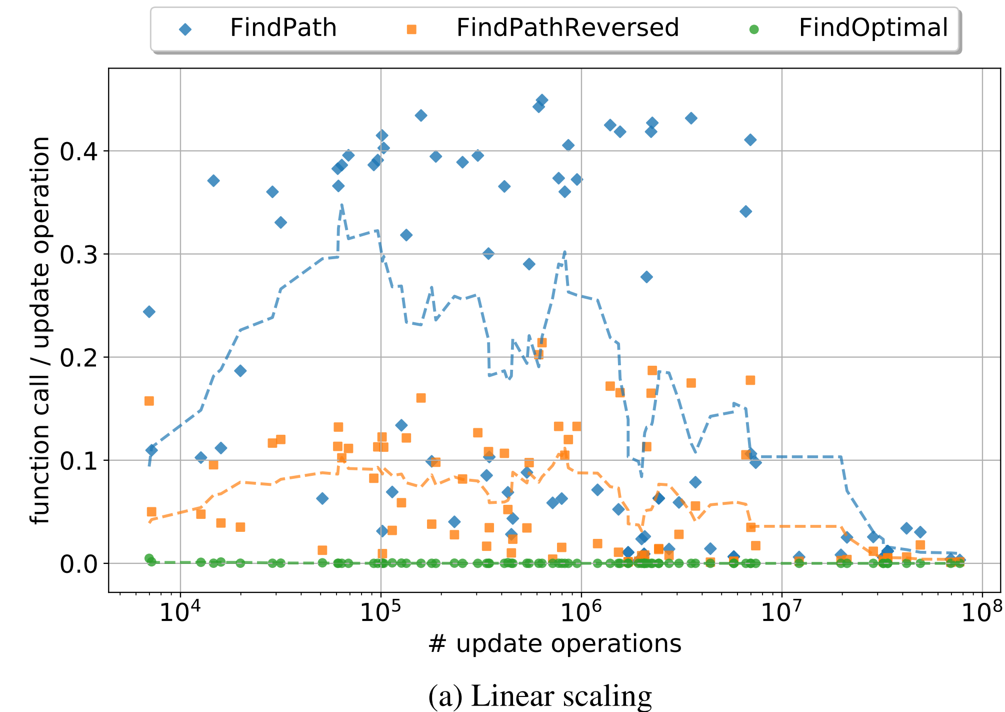
MinDegNodeList	23.165	0.066	0.024
$\text{BroFag}_{1.01}$	24.354	0.664	-
NIPseudoFirst	24.587	0.021	0.012
NIPseudoSecond	24.773	0.020	0.012
NIStartMaxDeg	25.586	0.018	0.014
NIStartMinDeg	25.593	0.017	0.014
NI	25.661	0.016	0.011
$\text{MaxDegEdge}_{o=min}$	28.652	0.040	0.025
EdgeProgDeg	30.940	0.011	0.003
EdgeInitDeg	34.684	0.011	0.004
MaxDegNode	35.178	0.010	0.006
$\text{MaxDegEdge}_{o=max}$	35.372	0.041	0.026
MaxDegEdge	35.709	0.033	0.026

Table 5.2: Average performance of all dynamic algorithm configurations, sorted by quality ($\Delta_{\mathcal{O}}$) in ascending order. Column 3 (Avg t [s]) displays the running time. The instances include static as well as dynamic sequences, see Table A.1 for more information. Pareto optimal configurations, determined by average values over all instances, are highlighted in grey. The average is calculated using the geometric mean.

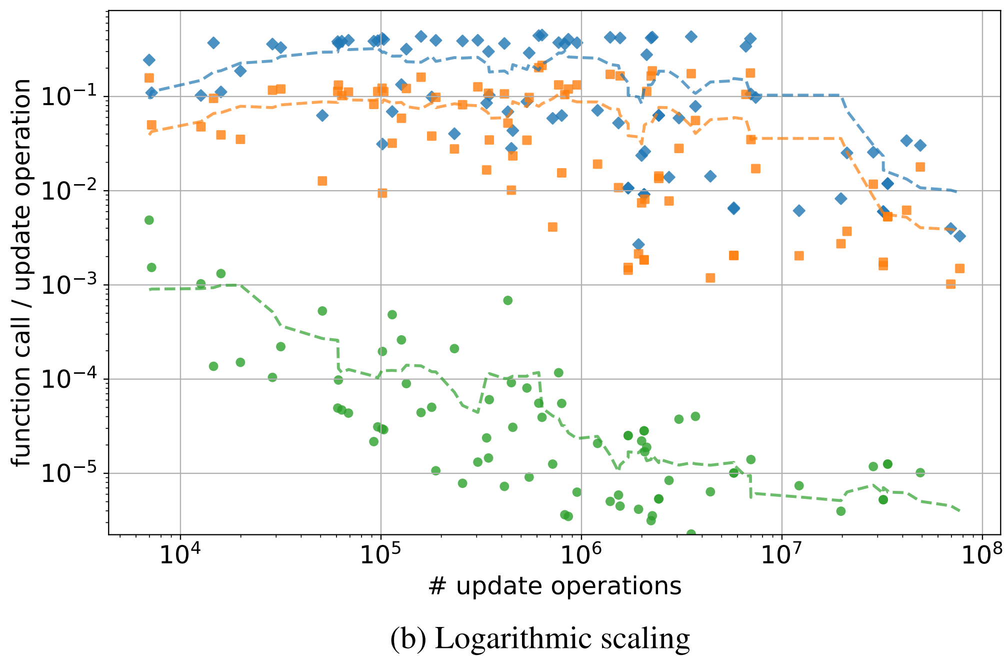
Algorithm	Avg $\Delta_{\mathcal{O}}$	Avg t [s]
DynOptEO	18.117	2.633
BFS_{20}	18.499	1.518
$\text{RPath}_{d=50,r=10}$	19.146	0.525
$\text{DescBFS}_{d=5,s=1}$	19.186	0.309
$\text{DescBFS}_{d=5,s=5}$	19.186	2.983
$\text{DescBFS}_{d=3,s=3}$	19.218	0.787
$\text{DescBFS}_{d=3,s=1}$	19.310	0.186
K-Flips_{50}	19.960	1.055
DescDegrees	20.124	0.074
$\text{DescBFS}_{d=1,s=2}$	20.449	0.094
$\text{DescBFS}_{d=1,s=1}$	21.306	0.067
Naive	23.792	0.022
$\text{BroFag}_{1.01}$	25.087	0.727

Table 5.3: Percentage of results on which the algorithms achieve optimal performances. Column 2 represents results on static sequences, whereas column 3 displays results on all sequences (see Table A.1).

Algorithm	opt solutions	opt solutions *(including deletions)*
DynOptEO	100%	100%
BFS_{20}	91.6%	92%
$\text{DescBFS}_{d=5,s=5}$	75.9%	77%
$\text{RPath}_{d=50,r=10}$	48.2%	47.1%
DescDegrees	24.1%	24.1%
K-Flips_{50}	13.3%	13.8%
$\text{MinDegEdge}_{o=max}$	12%	-
$\text{2ApproxMix}_{1.2}$	8.4%	-
MinDegNode	7.2%	-
NIPseudoFirst	4.8%	-
$\text{BroFag}_{1.01}$	3.6%	3.4%
Naive	0%	0%



(a) Linear scaling



(b) Logarithmic scaling

Figure 5.4: Plot displaying the fraction of function calls for DynOptEO, including FindPath, FindPathReversed and FindOptimal. The y-axis is given in linear scaling above (a) and logarithmic scaling below (b). The dataset consists of static sequences extended by deletion operations (#insertions = #deletions). For better visualization, trend lines with floating arithmetic average and window size 10 are shown.

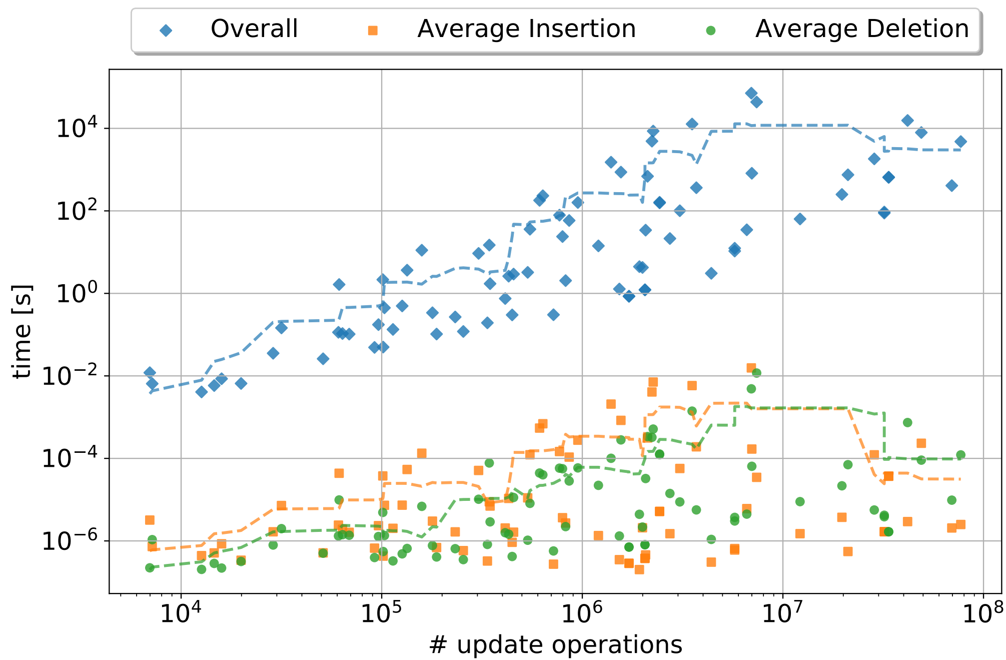
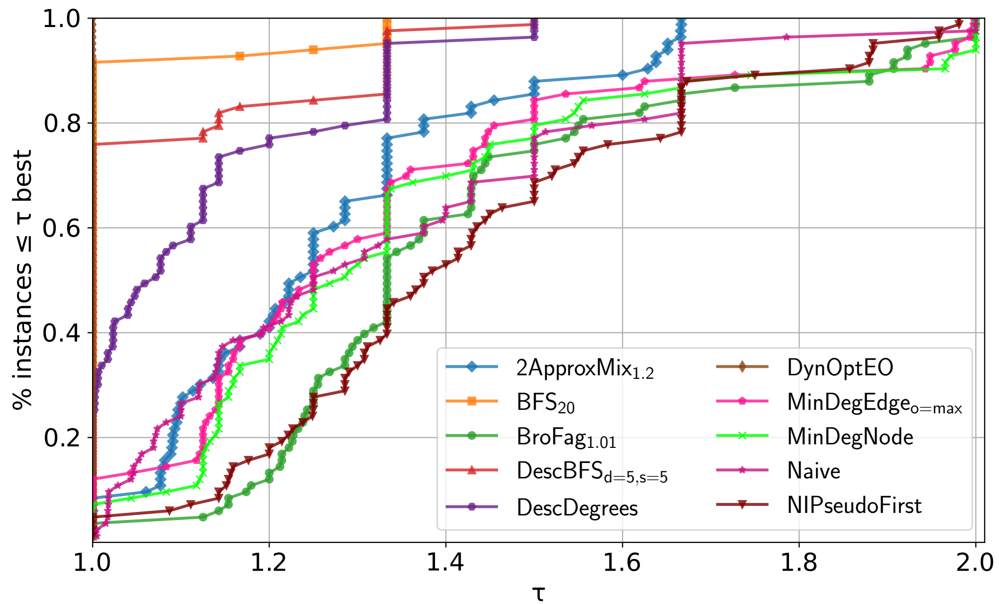
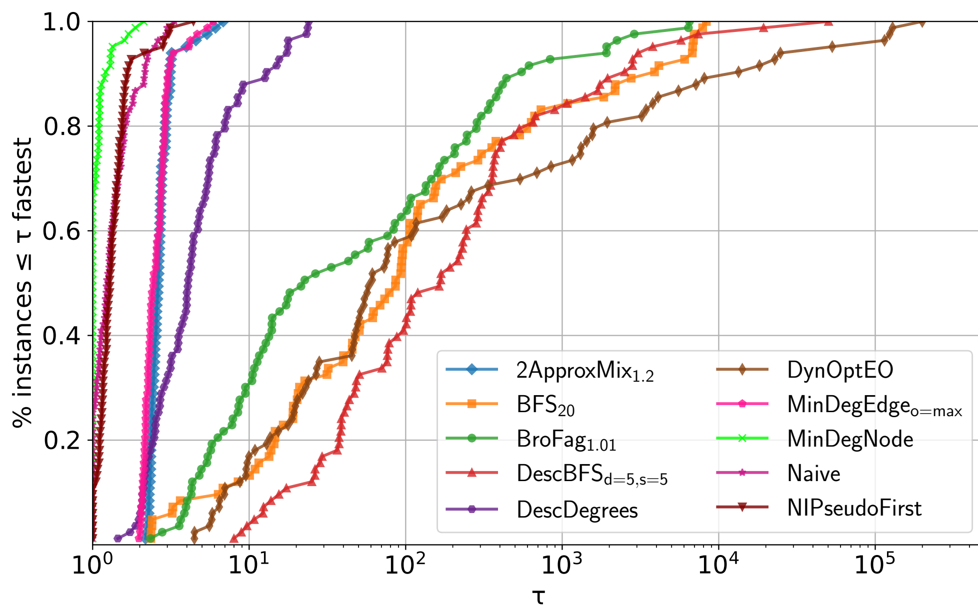


Figure 5.5: Plot illustrating the running time for DynOptEO, representing the overall graph computation time, alongside the arithmetic averages for Insertion and Deletion. The dataset consists of static sequences extended by deletion operations (#insertions = #deletions). For better visualization, trend lines with floating arithmetic average and window size 10 are shown.



(a) Quality



(b) Time

Figure 5.6: Performance profiles illustrating solution quality on the top and running time on the bottom for the overall comparison. The dataset consists of static sequences (see Table A.1).

CHAPTER 6

Discussion

6.1 Conclusion

In this work, we engineer and analyze numerous static linear-time algorithms for approximating the edge orientation problem, which is to minimize the maximum outdegree. These algorithms range from simple greedy strategies to utilizing FOREST for computing a forest partition. In terms of solution quality, 2ApproxMix and MinDegEdge stand out in particular, which outperform the 2-approximation MinDegNode [2, 9, 21] as well as Naive [7], even though the latter are faster. More specifically, we are able to further enhance the performance of 2ApproxMix by fine-tuning factors ($\sigma = 1.2$) for guessing the lower bound. Similarly, MinDegEdge can slightly be improved by manipulating the order in which edges are loaded into the graph data structure. However, the remaining algorithms yield less promising results. Most notably, even the strongest version of NI is inferior to MinDegNode in both objectives.

The second aspect of this work consists of two dynamic algorithms for maintaining edge orientations over a sequence of update operations. The experiments for DescBFS reveal that the search depth is a more important parameter than the number of descending steps, with the latter failing to sufficiently enhance the solution quality relative to their computational costs. Therefore, DescBFS is dominated by the underlying algorithm BFS [7]. Conversely, our novel dynamic optimal algorithm demonstrated promising results, with BFS being only 75.4% faster on average while achieving 92% of the best solutions.

6.2 Future Work

In order to achieve a more accurate approximation of the edge orientation, it could be beneficial to investigate the potential of combining algorithms. Using a 2-approximation to compute or guess a lower bound also falls into this category. We have already examined such an approach with 2ApproxMix, but instead of using MinDegEdge, one

could try various other algorithms for aligning the remaining edges. Another attempt could be to repeatedly run the same or different algorithms multiple times in succession. That is of course not possible for all discussed algorithms, but it maintains the same time complexity. It would also be interesting to determine whether the better-performing algorithms offer theoretical bounds.

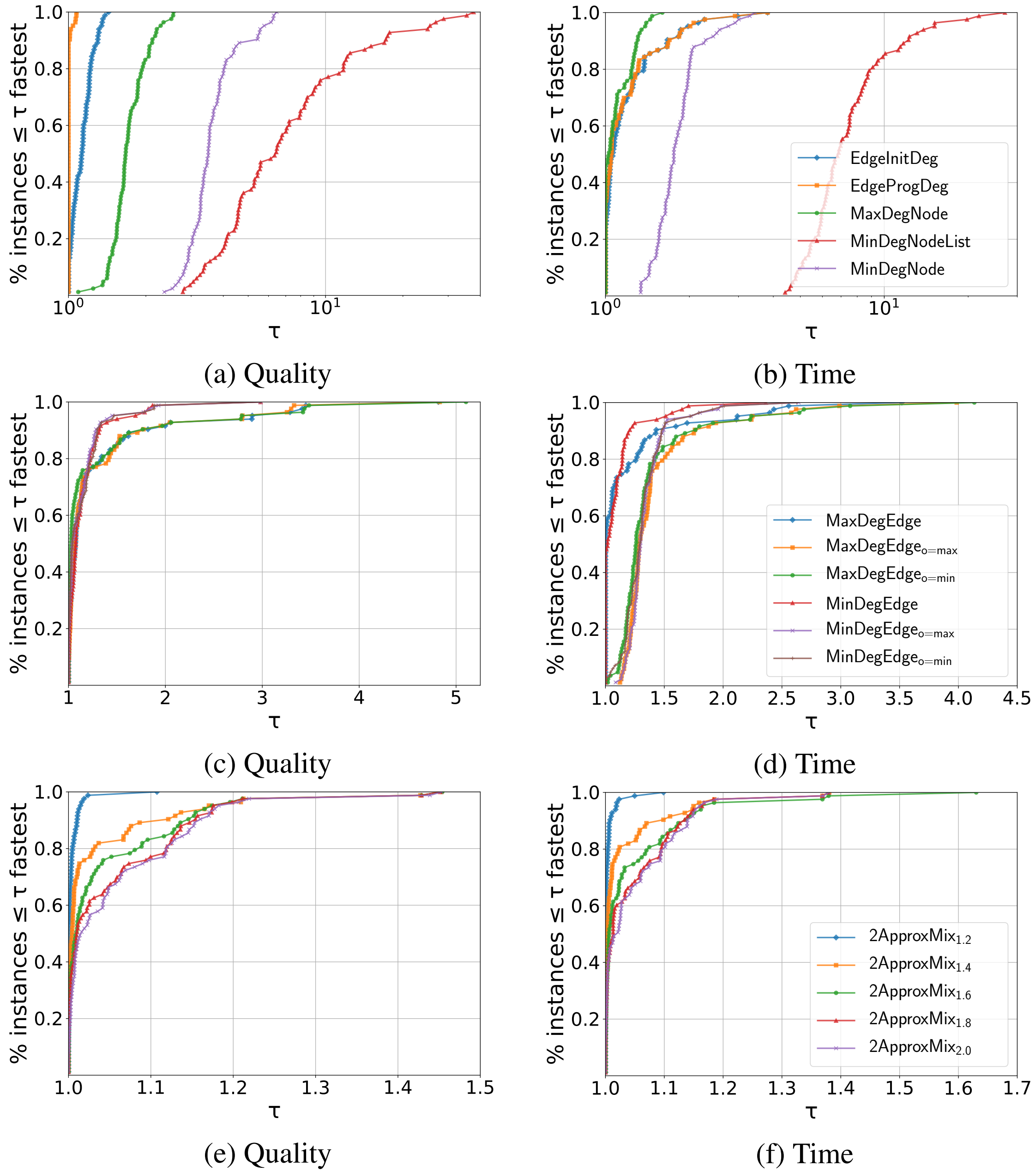
Furthermore, there are many variants of the Descending BFS algorithm that have not yet been experimentally evaluated. One such example is the algorithm also mentioned in Section 4.3.1, which always searches for the smallest node in the d-neighborhood and does not terminate the BFS early. This principle can also be applied to the BFS algorithm from the paper of Borowitz et al. [7].

For the optimal dynamic edge orientation algorithm, it may be beneficial to explore other speedup techniques to further improve the running time. This could be done by implementing more of the acceleration strategies used by Reinstädtler et al. [38] on their static approach for re-solving the orientation in FindOptimal. In addition, it may be beneficial to identify additional invariants that guarantee an optimal solution, whilst requiring the use of alternative algorithms. This approach could facilitate the discovery of algorithms that excel in specific domains, such as reducing the execution time for either insertions or deletions.

APPENDIX A

Appendix

A.1 Further Results



(a) Quality (b) Time

(c) Quality (d) Time

(e) Quality (f) Time

Figure A.1: Performance profiles illustrating solution quality on the left and running time (excluding initialization) on the right side. The rows refer to the categories of algorithms for completely processing nodes, algorithms for processing individual edges, and 2ApproxMix (factor σ) (top to bottom). The dataset consists of static sequences (see Table A.1).

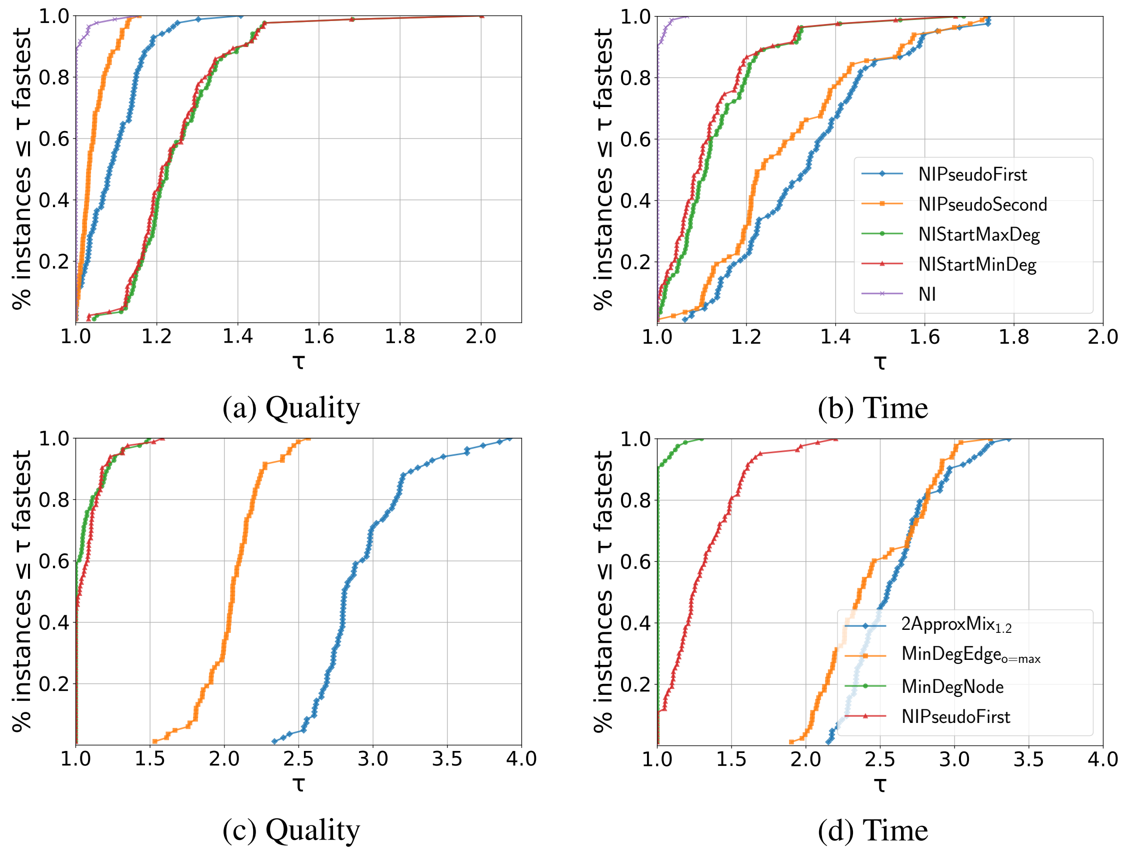
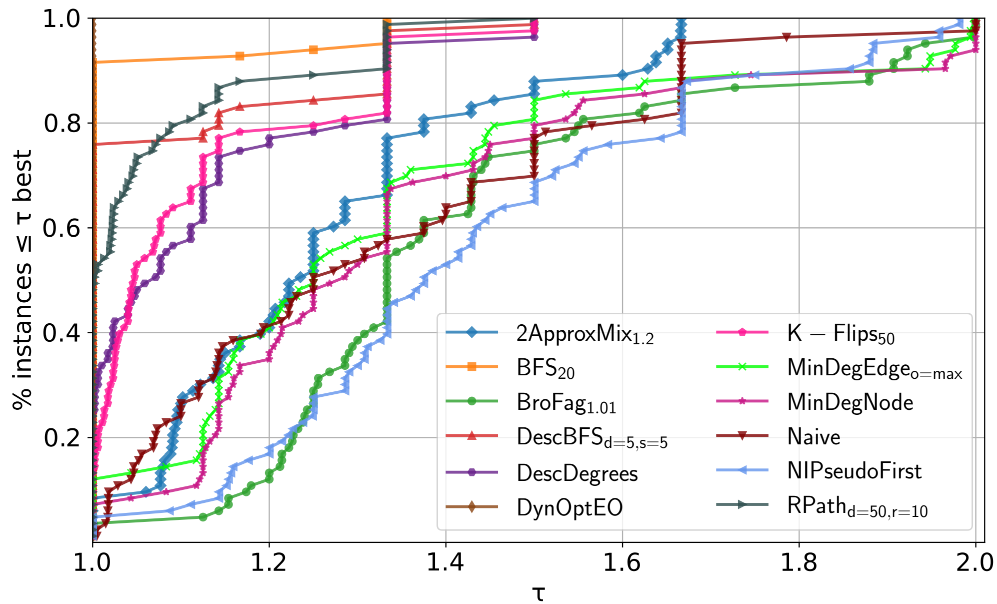
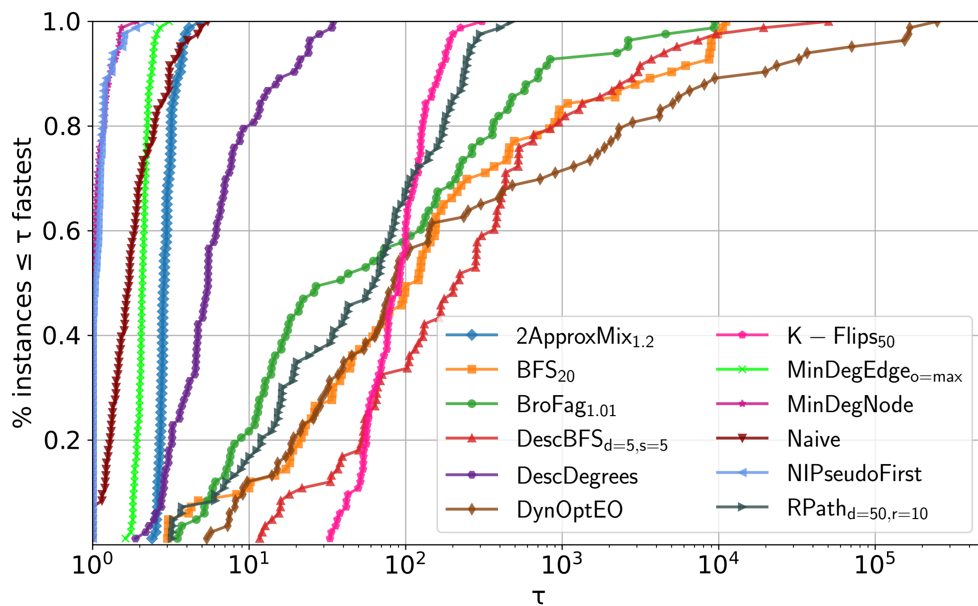


Figure A.2: Performance profiles illustrating solution quality on the left and running time (excluding initialization) on the right side for algorithms based on FOREST [31] and the comparison of best static algorithms (top to bottom). The dataset consists of static sequences (see Table A.1).



(a) Quality



(b) Time

Figure A.3: Performance profiles illustrating solution quality on the top and running time (excluding initialization for static algorithms) on the bottom for the overall comparison. The dataset consists of static sequences (see Table A.1).

A.2 Instances

Table A.1: Benchmark set of static, extended, and dynamic graphs (marked by *) provided by Borowitz et al. [7]. The number of nodes is represented by $|V|$ and in the case of static sequences, $|E|$ displays the number of edges. For real-world dynamic graphs, $\mathcal{U}$ shows the number of updates of the initial state before removing unnecessary updates like parallel edges, self-loops, etc. The total number of insertions and deletions (after adding update operations to sequences that only featured insertions) is indicated by $\mathcal{I}|\mathcal{D}$. If these are equal, only one value is entered, representing both.

Mesh Type Networks

Graph	$\|V\|$	$\|E\|$	$\mathcal{I} \mid \mathcal{D}$
144	144 649	1 074 393	1 127 984
3elt	4 720	13 722	14 380
4elt	15 606	45 878	48 170
598a	110 971	741 934	778 940
add20	2 395	7 462	7 958
add32	4 960	9 462	9 952
auto	448 695	3 314 611	3 480 243
bcsstk29	13 992	302 748	317 885
bcsstk30	28 924	1 007 284	1 057 510
bcsstk31	35 588	572 914	601 552
bcsstk32	44 609	985 046	1 034 183
bcsstk33	8 738	291 583	306 144
brack2	62 631	366 559	384 764
crack	10 240	30 380	31 876
cs4	22 499	43 858	46 047
cti	16 840	48 232	50 624
data	2 851	15 093	15 820
delaunay16	65 536	196 575	206 346
delaunay17	131 072	393 176	412 794
delaunay20	1 048 576	3 145 686	3 302 529
fe_4elt2	11 143	32 818	34 422
fe_body	45 087	163 734	171 885
fe_ocean	143 437	409 593	430 019
fe_pwt	36 519	144 794	152 033
fe_rotor	99 617	662 431	695 444
fe_sphere	16 386	49 152	51 597
fe_tooth	78 136	452 591	475 172

Continuation on the next page

Table A.1: (Continuation)

finan512	74 752	261 120	274 156
m14b	214 765	1 679 018	1 762 927
memplus	17 758	54 196	56 902
rgg15	32 768	160 240	168 214
rgg16	65 536	342 127	359 161
rgg17	131 072	728 753	765 128
t60k	60 005	89 440	93 879
uk	4 824	6 837	7 310
vibrobox	12 328	165 250	173 468
wave	156 317	1 059 331	1 112 189
whitaker3	9 800	28 989	30 430
wing	62 032	121 544	127 594
wing_nodal	10 937	75 488	79 216

Social Networks

graph	$\lvert V\rvert$	$\lvert E\rvert$	$\mathcal{I} \mid \mathcal{D}$
amazon-2008	735 323	3 523 472	3 699 277
as-22july06	22 963	48 436	50 841
as-skitter	554 930	5 797 663	6 086 987
citationCiteseer	268 495	1 156 647	1 214 396
citationCiteseer.ddsg	268 495	1 156 647	1 214 474
cnr-2000	325 557	2 738 969	2 875 510
cnr-2000.ddsg	325 557	2 738 969	2 875 486
coAuthorsCiteseer	227 320	814 134	854 522
coAuthorsCiteseer.ddsg	227 320	814 134	854 634
coAuthorsDBLP	299 067	977 676	1 026 378
coAuthorsDBLP.ddsg	299 067	977 676	1 026 546
coPapersCiteseer	434 102	16 036 720	16 837 972
coPapersCiteseer.ddsg	434 102	16 036 720	16 838 551
coPapersDBLP	540 486	15 245 729	16 007 462
coPapersDBLP.ddsg	540 486	15 245 729	16 007 442
email-EuAll	16 805	60 260	63 248
enron	69 244	254 449	267 156
in-2004	1 382 908	13 591 473	14 271 046
loc-brightkite_edges	56 739	212 945	223 519
loc-gowalla_edges	196 591	950 327	997 585
p2p-Gnutella04	6 405	29 215	30 670

Continuation on the next page

Table A.1: (Continuation)

PGPgiantcompo	10 680	24 316	25 517
rhg1G	1 000 000	10 047 330	10 549 402
rhg2G	1 000 000	19 923 820	20 919 538
soc-Slashdot0902	28 550	379 445	398 410
wordassociation-2011	10 617	63 788	66 924
web-Google	356 648	2 093 324	2 197 929
wiki-Talk	232 314	1 458 806	1 531 555
Real-World Dynamic Networks			
graph	$\|V\|$	$\mathcal{U}$	$\mathcal{I} \mid \mathcal{D}$
amazon-ratings*	2 146 058	5 838 041	476 756 | 920
citeulike_ui	731 770	2 411 819	962 964
dewiki*	2 166 670	86 337 879	55 026 399 | 26 996 658
dewiki-2013	1 532 354	33 093 029	34 747 300
dnc-temporalGraph	2 030	39 264	6 323
facebook-wosn-wall	46 953	876 993	227 197
flickr-growth	2 302 926	33 140 017	24 495 064
haggle	275	28 244	3 490
lastfm_band	174 078	19 150 868	1 851 245
lkml-reply	63 400	1 096 440	214 740
movielens10m*	69 879	10 000 054	384 169 | 416
munmun_digg	30 399	87 627	89 521
proper_loans	89 270	3 394 979	3 499 515
sociopatterns-infections	411	17 298	3 589
stackexchange-stackoverflow	545 197	1 301 942	1 366 705
topology	34 762	171 403	116 233
wiki_simple_en*	100 313	1 627 472	1 124 451 | 433 772
wikipedia-growth	1 870 710	39 953 145	38 529 875
youtube-u-growth	3 223 590	9 375 374	9 844 127

A.3 Implementation Details

A.3.1 Fully Dynamic Δ-Orientation Algorithms

The the pseudocode for the algorithms BFS, RPath, DescDegrees, K-Flips, BroFag from Borowitz et al. [7] is included in the following. To be precise, the pseudocode was adapted to the style used in this work and to match the pseudocode of DescDegrees in Algorithm 12. We also added the parameters used in the experimental evaluation for clarification.

Algorithm 18: Improving u-y-Path Search Algorithm

```
1  global variable: Δ_O
   input: depth d
2  procedure Insertion(O = (V_O, Adj), u, v):
3  Adj[u] = Adj[u] ∪ {v}
4  if odeg(u, O) < Δ_O or Δ_O = 1 then return
5  // find a path p = (u, ..., y), odeg(y, O) < odeg(u, O) − 1
6  p = BFS-Search(u, d) // bounded by depth d
7  if p ≠ ∅ then
8  |  flip all edges of p
9  end if
10 procedure Deletion(O = (V_O, Adj), u, v):
11 Adj[u] = Adj[u] \ {v}
12 Adj[v] = Adj[v] \ {u}
13 procedure Adjacent(O = (V_O, Adj), u, v):
14 return u ∈ Adj[v] or v ∈ Adj[u]
```

Algorithm 19: Random Path Algorithm

```
1  global variable: Δ_O
   input: depth d, # repetitions r
2  procedure Insertion(O = (V_O, Adj), u, v):
3  Adj[u] = Adj[u] ∪ {v}
4  if odeg(u, O) < Δ_O or Δ_O = 1 then return
5  for i = 1 to r do
6  |   // find a path p = (u, ..., y), odeg(y, O) < odeg(u, O) − 1
7  |   p = Random-Path(u)
8  |   if p ≠ ∅ then
9  |   |   flip all edges of p
10 |   |   break
11 |   end if
12 end for
13 procedure Deletion(O = (V_O, Adj), u, v):
14 Adj[u] = Adj[u] \ {v}
15 Adj[v] = Adj[v] \ {u}
```

Algorithm 20: Descending Degrees Algorithm

```
1  global variable: Δ_O
2  procedure Insertion(O = (V_O, Adj), u, v):
3  Adj[u] = Adj[u] ∪ {v}
4  if odeg(u, O) < Δ_O or Δ_O = 1 then return
5  while DescendingDegrees(O, u) ;
6  procedure DescendingDegrees(O = (V_O, Adj), u):
7  v = arg min_{v∈Adj[u]}(odeg(v, O))
8  if odeg(v, O) < odeg(u, O) − 1 then
9  |   Adj[u] = Adj[u] \ {v}
10 |   Adj[v] = Adj[v] ∪ {u}
11 |   DescendingDegrees(O = (V_O, Adj), u)
12 |   return True
13 end if
14 return False
15 procedure Deletion(O = (V_O, Adj), u, v):
16 Adj[u] = Adj[u] \ {v}
17 Adj[v] = Adj[v] \ {u}
```

Algorithm 21: K-Flips [5]

```
   input: maximum flips k
 1 procedure Insertion(𝒪 = (V_𝒪, Q), u, v):
 2 Q_u = Q_u ∪ {(u, v)}
 3 K-Flips(𝒪)
 4 procedure Deletion(𝒪 = (V_𝒪, Q), u, v):
 5 Q_u = Q_u \ {(u, v)}
 6 K-Flips(𝒪)
 7 procedure K-Flips(𝒪 = (V_𝒪, Q)):
 8 for i = 1 to k do
 9 |   u = arg max_{u∈V_𝒪}(odeg(u, Q_u))
10 |   Q_u = Q_u \ {(u, v)}
11 |   Q_v = Q_v ∪ {(v, u)}
12 end for
```

Algorithm 22: Brodal and Fagerberg [8]

```
    input: α̃ a bound on the arboricity α, factor β
 1  procedure Insertion(O = (V, Adj), u, v):
 2  Adj[u] = Adj[u] ∪ {v}
 3  if odeg(u, O) = α̃ + 1 then
 4  |   S := {u}
 5  |   while |S| ≠ ∅ do
 6  |   |   w := pop(S)
 7  |   |   for x ∈ Adj[w] do
 8  |   |   |   Adj[x] = Adj[x] ∪ {w}
 9  |   |   |   if odeg(x, O) = α̃ + 1 then
10  |   |   |   |   push(S, x)
11  |   |   |   end if
12  |   |   end for
13  |   |   Adj[w] = ∅
14  |   end while
15  end if
16  procedure Deletion(O = (V, Adj), u, v):
17  Adj[u] = Adj[u] \ {v}
18  Adj[v] = Adj[v] \ {u}
19  procedure Rebuild(O = (V, Adj)):
20  α̃ = β · α̃
21  A := {(u, v) ∈ V_O × V_O | v ∈ Adj[u]}
22  Adj = [∅] × |V_O|
23  for (u, v) ∈ A do Insertion(O, u, v)
```

Zusammenfassung

Das *edge orientation* Problem besteht darin, den Kanten eines Graphen G Richtungen zuzuweisen, was einen gerichteten Graphen $\mathcal{O}$ erzeugt, wobei der maximale Ausgangsgrad innerhalb von $\mathcal{O}$ minimiert werden soll. Dieses Problem besitzt Anwendungen in verschiedenen Bereichen und kann in polynomialer Zeit optimal gelöst werden. Aufgrund der geringen Anzahl von linearen Ansätzen in der aktuellen Literatur erkunden wir mehrere Strategien zur Approximation der Lösung. Darüber hinaus involviert die dynamische Variante dieses Problems die Erhaltung der Kantenorientierungen über eine Folge von Aktualisierungsoperationen. Hierfür präsentieren wir einen Ansatz, der auf Konzepten von zwei bestehenden Algorithmen basiert und mehrere Iterationen einer Breitensuche für absteigende Knoten durchführt. Darüber hinaus schlagen wir einen neuen Algorithmus zur optimalen Lösung des dynamischen *edge orientation* Problems durch Invarianten vor, welchen wir durch einen Beweis stützen. Aufbauend auf einem bestehenden optimalen statischen Algorithmus manipuliert dieser Ansatz Pfade, um Verbesserungen zu erwirken. Wir werten unserer Implementierungen gegen Algorithmen auf dem aktuellen Stand der Technik aus. Dies zeigt, dass selbst der beste nicht optimale Konkurrent, welcher eine einfache Breitensuche verwendet, im Durchschnitt nur 75.4% schneller als unser dynamischer optimaler Algorithmus ist und dabei 92% der besten Lösungen erreicht.

Bibliography

[1] Oswin Aichholzer, Franz Aurenhammer and Günter Rote. *Optimal graph orientation with storage applications*. Universität Graz/Technische Universität Graz. SFB F003-Optimierung und Kontrolle, 1995.

[2] Srinivasa R. Arikati, Anil Maheshwari, Christos D. Zaroliagis. Efficient computation of implicit representations of sparse graphs. *Discrete Applied Mathematics*, 78(1-3):1-16, 1997. doi: 10.1016/S0166-218X(97)00007-3. URL `https://doi.org/10.1016/S0166-218X(97)00007-3`.

[3] Yuichi Asahiro, Eiji Miyano, Hirotaka Ono, and Kouhei Zenmyo. Graph orientation algorithms to minimize the maximum outdegree. *International Journal of Foundations of Computer Science*, 18(02):197-215, 2007. doi: 10.1142/S0129054107004644. URL `https://doi.org/10.1142/S0129054107004644`.

[4] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. In Reda Alhajj and Jon G. Rokne, editors, *Encyclopedia of Social Network Analysis and Mining, 2nd Edition*. Springer, 2018. doi: 10.1007/978-1-4939-7131-2_23. URL `https://doi.org/10.1007/978-1-4939-7131-2_23`.

[5] Edvin Berglin and Gerth S. Brodal. A simple greedy algorithm for dynamic graph orientation. *Algorithmica*, 82(2):245-259, 2020. doi: 10.1007/s00453-018-0528-0. URL `https://doi.org/10.1007/s00453-018-0528-0`.

[6] Markus Blumenstock. Fast Algorithms for Pseudoarboricity. In M. T. Goodrich and M. Mitzenmacher, editors, *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2016, Arlington, Virginia, USA, January 10, 2016*, pages 113-126. SIAM, 2016. doi: 10.1137/1.9781611974317.10. URL `https://doi.org/10.1137/1.9781611974317.10`.

[7] Jannick Borowitz, Ernestine Großmann, and Christian Schulz. Engineering fully dynamic Δ-orientation algorithms. In *ACDA*. SIAM, 2023.

[8] Gerth S. Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*, pages 342-351. Springer, 1999. doi: 10.1007/3-540-48447-7_34. URL https://doi.org/10.1007/3-540-48447-7_34.

[9] Moses Charikar. Greedy approximation algorithms for finding dense components in a graph. In K. Jansen and S. Khuller, editors, *Approximation Algorithms for Combinatorial Optimization, Third International Workshop, APPROX 2000, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1913 of *Lecture Notes in Computer Science*, pages 84-95. Springer, 2000. doi: 10.1007/3-540-44436-X_10. URL https://doi.org/10.1007/3-540-44436-X_10.

[10] Aleksander B. G. Christiansen, Jacob Holm, Ivor van der Hoog, Eva Rotenberg, and Chris Schwiegelshohn. Adaptive out-orientations with applications. *CoRR*, abs/2209.14087, 2022. doi: 10.48550/arXiv.2209.14087. URL https://doi.org/10.48550/arXiv.2209.14087.

[11] Boliong Chen, Makoto Matsumoto, Jianfang Wang, Zhongfu Zhang, and Jianxun Zhang. A short proof of nash-williams' theorem for the arboricity of a graph. *Graphs Comb.*, 10(1):27-28, 1994. ISSN 0024-6107. doi: 10.1007/BF01202467. URL https://doi.org/10.1007/BF01202467.

[12] Timothy Davis. The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, 2008. URL http://www.cise.ufl.edu/research/sparse/matrices/.

[13] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):303-324, 2002. doi: 10.1007/s101070100263. URL https://doi.org/10.1007/s101070100263.

[14] Zdeněk Dvořák and Vojtěch Tůma. A dynamic data structure for counting subgraphs in sparse graphs. In *Algorithms and Data Structures - 13th International Symposium, WADS*, pages 304-315, 2013. doi: 10.1007/978-3-642-40104-6_27. URL https://doi.org/10.1007/978-3-642-40104-6_27.

[15] David Eppstein. All maximal independent sets and dynamic dominance for sparse graphs. *ACM Transactions on Algorithms*, 5(4), 2009. doi:10.1145/1597036.1597042. URL https://doi.org/10.1145/1597036.1597042.

[16] András Frank and András Gyárfás. How to orient the edges of a graph? In András Hajnal and Vera T. Sós, editors, *COMBINATORICS: 5th Hungarian Colloquium, Keszthely, June/July 1976, Proceedings*, number 2 in Colloquia Mathematica Societatis János Bolyai, pages 353-364. North Holland Publishing Company, 1978.

[17] Daniel Funke, Sebastian Lamm, Ulrich Meyer, Manuel Penschuck, Peter Sanders, Christian Schulz, Darren Strash, and Moritz von Looz. Communication-free massively distributed graph generation. *J. Parallel Distributed Comput.*, 131:200-217, 2019. doi: 10.1016/j.jpdc.2019.03.011. URL `https://doi.org/10.1016/j.jpdc.2019.03.011`.

[18] Harold N. Gabow. Algorithms for Graphic Polymatroids and Parametrics-Sets. *Journal of Algorithms*, 26(1):48-86, 1998. ISSN 0196-6774. Springer, 2006. doi: 10.1006/jagm.1997.0904. URL `https://www.sciencedirect.com/science/article/pii/S0196677497909044`.

[19] Harold N. Gabow and Herbert H. Westermann. Forests, frames, and games: Algorithms for matroid sums and applications. *Algorithmica 7*, pages 465-497, 1992. doi: 10.1007/BF01758774. URL `https://doi.org/10.1007/BF01758774`.

[20] Giorgio Gallo, Michael D. Grigoriadis, and Robert E. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18(1):30-55, 1989. doi: 10.1137/0218003. URL `https://doi.org/10.1137/0218003`.

[21] George F. Georgakopoulos and Kostas Politopoulos. MAX-DENSITY revisited: a generalization and a more efficient algorithm. *The Computer Journal*, 50(3):348-356, 2007. doi: 10.1093/COMJNL/BXL082. URL `https://doi.org/10.1093/comjnl/bxl082`.

[22] Andrew V. Goldberg. Finding a maximum density subgraph. 1984. URL `https://www2.eecs.berkeley.edu/Pubs/TechRpts/1984/CSD-84-171.pdf`.

[23] Meng He, Ganggui Tang, and Norbert Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In Hee-Kap Ahn and Chan-Su Shin, editors, *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*, volume 8889 of *Lecture Notes in Computer Science*, pages 128-140. Springer, 2014. doi: 10.1007/978-3-319-13075-0_11. URL `https://doi.org/10.1007/978-3-319-13075-0_11`0.

[24] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a Scalable High Quality Graph Partitioner. *Proc. of the 24th IEEE International Parallel and Distributed Processing Symposium*, pages 1-12, 2010.

[25] Sampath Kannan, Moni Naor, and Steven Rudich. Implicit representation of graphs. *SIAM Journal on Discrete Mathematics*, 5(4):596-603, 1992. doi: 10.1137/0405049. URL `https://doi.org/10.1137/0405049`.

[26] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573 of *Lecture Notes in Computer Science*, pages 532-543. Springer, 2014. doi: 10.1007/978-3-662-43951-7_45. URL `https://doi.org/10.1007/978-3-662-43951-7_45`.

[27] Łukasz Kowalik. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.*, 102(5):191-195, 2007. doi: 10.1016/j.ipl.2006.12.006. URL `https://doi.org/10.1016/j.ipl.2006.12.006`.

[28] Łukasz Kowalik. Approximation scheme for lowest outdegree orientation and graph density measures. *International Symposium on Algorithms and Computation*, pages 557-566. Springer, 2006. doi: 10.1007/11940128_56. URL `https://doi.org/10.1007/11940128_56`.

[29] Jérôme Kunegis. KONECT: the koblenz network collection. In *22nd World Wide Web Conference, WWW '13*, pages 1343-1350, 2013. doi: 10.1145/2487788.2488173. URL `https://doi.org/10.1145/2487788.2488173`.

[30] Jure Leskovec. Stanford Network Analysis Package (SNAP). URL `http://snap.stanford.edu/index.html`.

[31] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7:583-596, 1992. doi: 10.1007/BF01758778. URL `https://doi.org/10.1007/BF01758778`.

[32] Crispin St.J. A. Nash-Williams. Edge-Disjoint Spanning Trees of Finite Graphs. *Journal of the London Mathematical Society*, s1-36(1):445-450, 1961. ISSN 0024-6107. doi: 10.1112/jlms/s1-36.1.445. URL `https://doi.org/10.1112/jlms/s1-36.1.445`.

[33] Crispin St.J. A. Nash-Williams. Decomposition of Finite Graphs Into Forests. *Journal of the London Mathematical Society*, s1-39(1):12-12, 1964. ISSN 0024-6107. doi: 10.1112/jlms/s1-39.1.12. URL `https://doi.org/10.1112/jlms/s1-39.1.12`.

[34] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proceedings of the 45th ACM Symposium on Theory of Computing, STOC*, pages 745-754, 2013. doi:10.1145/2488608.2488703. URL `https://doi.org/10.1145/2488608.2488703`.

[35] Manuel Penschuck, Ulrik Brandes, Michael Hamann, Sebastian Lamm, Ulrich Meyer, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in scalable network generation. *CoRR*, abs/2003.00736, 2020. URL `https://arxiv.org/abs/2003.00736`.

[36] Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2):141-159, 1982. doi: 10.1002/net.3230120206. URL `https://doi.org/10.1002/net.3230120206`.

[37] Julia Preusse, Jérôme Kunegis, Matthias Thimm, Thomas Gottron, and Steffen Staab. Structural dynamics of knowledge networks. In *Proc. Int. Conf. on Weblogs and Social Media*, 2013.

[38] Henrik Reinstädtler, Christian Schulz, and Bora Uçar. Engineering Edge Orientation Algorithms. *CoRR*, arXiv:2404.13997, 2024. doi: 10.48550/arXiv.2404.13997. URL `https://doi.org/10.48550/arXiv.2404.13997`.

[39] Venkat Venkateswaran. Minimizing maximum indegree. *Discrete Applied Mathematics*, 143(1-3):374-378, 2004. doi: 10.1016/j.dam.2003.07.007. URL `https://doi.org/10.1016/j.dam.2003.07.007`.