Engineering Heuristic Quasi-Threshold Editing

Master Thesis of

David Schmitt

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Jun.-Prof. Dr. Thomas Bläsius
Advisors:	Michael Hamann

Time Period: 1st January 2021 – 30th June 2021

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, June 30, 2021

Abstract

We introduce an algorithm that solves the quasi-threshold graph editing problem. The problem is defined as removing and inserting edges in a graph to reach a quasi-threshold graph. Quasi-threshold graphs are the graphs without P_4 and C_4 as node-induced subgraphs and can also be defined as the transitive closure of a rooted forest. All operations to the edges are called edits, and we examine nonuniform edit costs. Given a graph, the algorithm computes a quasi-threshold graph, which is close to the original regarding the sum of edit costs for the edge insertions and deletions. Building on the previously introduced heuristic Quasi-Threshold Mover (QTM), which is limited to uniform edit costs, we consider edit costs for edge insertions and deletions. QTM represents the graph as a skeleton forest and works in rounds. In each round, QTM heuristically improves the number of edits by trying to move every node to a new position in the forest. The approach is called local move. We use the same concept of local move from QTM to solve the problem while minimizing edit costs. We show that QTM requires only a few changes to be able to solve the problem with nonuniform edit costs. Additionally, we propose improvements in solution quality for uniform and nonuniform edit costs by moving nodes with their subtree instead of the node alone. This optimization is called subtree move. In an extensive experimental evaluation, we apply the algorithms to two large sets of graphs. One set includes edit costs and represent connected components of a COG protein similarity network. The second set visualizes large social networks, and we evaluate uniform edit costs on it. The quality of the solution and running time per round of QTM suffers with edit costs considered in comparison to uniform edit costs. QTM with nonuniform edit costs does, however, on average, converge in fewer rounds than with uniform edit costs. The solution quality can be improved by enabling the subtree move optimization. On the set of COG protein similarity graphs, we compare the solution we find to an exact solver of the problem and are able to match the solution on most graphs. We also show that subtree move can improve the solution quality of the editing problem with uniform edit costs. For both uniform and nonuniform edit costs, though, subtree move comes with an increase in the running time of the algorithm.

Deutsche Zusammenfassung

Wir stellen einen Algorithmus vor, der das Quasi-Threshold Editier Problem löst. Das Problem ist definiert als, Kanten eines Graphen einzufügen oder zu löschen um einen Quasi-Threshold Graphen zu erreichen. Quasi-Threshold Graphen sind Graphen, die weder P_4 noch C_4 als Knoten-induzierten Teilgraphen enthalten und können als transitive Hülle eines Waldes mit Wurzeln definiert werden. Wir bezeichnen Kanteneinfügungen und -löschungen zusammen als Editierungen und untersuchen nichteinheitliche Editierkosten. Für einen gegebenen Graphen findet der Algorithmus einen Quasi-Threshold Graph, der nahe am Orignalgraphen ist, was hier bedeutet dass er Kanten löscht und einfügt, die in Summe niedrige Editierkosten benötigen. Wir bauen auf dem bereits vorgestellten heuristischen Quasi-Threshold Mover (QTM) auf, der sich auf einheitliche Editierkosten beschränkt, und beziehen nun Editierkosten für Kanteneinfügungen und löschungen mit ein. QTM stellt den Graphen als Wald dar und arbeitet in Runden. In jeder Runde verbessert er heuristisch die Anzahl der Editierungen, indem er versucht jeden Knoten an eine neue Stelle im Wald zu bewegen. Das Vorgehen wird als Local Move bezeichnet. Wir benutzen dasselbe

Konzept von QTM um das Problem mit nichteinheitlichen Editierkosten zu lösen. Wir zeigen, dass QTM mit wenigen Änderungen in der Lage ist, das Problem mit nichteinheitlichen Editierkosten zu lösen. Zusätzlich stellen wir Verbesserungen der Lösungsqualität vor, sowohl für einheitliche als auch nichteinheitliche Editierkosten, indem wir das Bewegen von Knoten mit ihrem Teilbaum erlauben. Wir nennen diese Optimierung Subtree Move. In der experimentellen Auswertungen wenden wir dann den Algorithmus auf zwei große Gruppen von Graphen an. Eine Gruppe enthält Editierkosten und stellt Zusammenhangskomponenten eines COG Protein Ähnlichkeitsnetzwerks dar. Die zweite Gruppe visualisiert soziale Netwerke und wir evaluieren einheitliche Editierkosten auf dieser Gruppe. Die Lösungsqualität und Laufzeit von QTM leidet im Vergleich zu einheitlichen Editierkosten, wenn wir Editierkosten betrachten. QTM mit nichteinheitlichen Editierkosten konvergiert jedoch durchschnittlich in weniger Runden als QTM mit einheitlichen Editierkosten. Die Lösungsqualität kann verbessert werden, indem wir die Subtree Move Optimierung aktivieren. Auf den COG Protein Ähnlichkeitsgraphe vergleichen wir unsere Lösung mit einem exakten Algorithmus. Bis auf wenige Ausnahmen erreichen wir die exakte Lösung auf den einzelnen Graphen. Wir zeigen außerdem, dass Subtree Move ebenfalls die Lösungsqualität des Editierproblems mit einheitlichen Editierkosten verbessert. Sowohl für einheitliche als auch für nichteinheitliche Editierkosten geht jedoch die Optimierung mit einer Erhöhung der Laufzeit pro Runde einher.

Contents

1 Introduction **1**

2 Preliminaries **7**

3 QTM with Edit Costs **11**

3.1 Basic Algorithm 11
3.2 Initialization 12
3.3 Local Move 13
3.4 Simple Path Sorting 15
3.5 Edit Costs of Close Children 17
3.6 Max Edit Cost 19
3.7 Edit Cost Ratio 21
3.8 Correctness 22
3.9 Running Time 25

4 Subtree Move **29**

4.1 Basic Subtree Move Algorithm 29
4.2 Subtree Move 30
4.3 Best Parent 32
4.4 Subtree Sort Path 35
4.5 Subtree Move Option 36
4.6 Correctness 37
4.7 Running Time 38

5 Evaluation **41**

5.1 Graph Instances 41
5.2 Solution Quality 42
5.3 Running Time 46
5.4 Exact Solutions 49
5.5 Convergence 51
5.6 Difference in Solutions 54

6 Conclusion **59**

Bibliography **61**

1. Introduction

A lot of structured information from different use cases can be modeled as networks. In the context of biological networks, we can, for example, visualize proteins and their interaction between each other. The network representation of the proteins allows us to visualize complex information that otherwise would be hard to depict. We can annotate the different parts of the protein-protein interaction network with additional attributes. For connections in the network, this includes, for example, the type of interaction or a value for the strength of the interaction between two proteins. We can also argue for directed connections if the interactions are one-sided. We can additionally have properties for the protein, e.g., the primary structure of the protein. The network then allows us to analyze all the information in a structured approach. Protein-protein interaction network analysis can play, for instance, a role in medical research [WFS10].

Another example of networks are social networks. A social network can represent different relations between two people or group of people and a hierarchy for these relations. Example relations are friendships, family bonds, working relations, or even dislike between two people. The connection is not limited to social relation, but can also represent a more abstract concept, such as information flow between people. Depending on the network, it might be possible to give a value to a connection between two people, which shows how strong the connection between them is. The value can, for example, be higher for blood-relation and lower for two people working together.

Within the research area of so-called social network analysis, the focus lies on examining social structures in different contexts with the help of networks. The networks are not only limited to the classic social media networks, but can also include information circulation [Mar17] or knowledge networks [BR17]. While these networks still represent people most of the time, the type of connection between them differs for the different networks. One of the most researched topics for social networks is community detection [Sch07];[For10];[HKW16];[CDVMDW19]. The idea is here to find groups of people in the network that have many connections between them but limited connections to other people in the network. This can, for example, be a close group of friends, an extended family, or members of the same sports club.

To solve a problem like this, we transform the information of the network into a graph and solve the problem on the resulting graph. For social networks, the people are represented as nodes and the connections between them as edges of the graph if a connection exists. A value, how strong the connection is, can be represented as a weight function for the edges of the graph. We can allow negative values for this weight to represent, for example, dislike in a social network. We call this graph then a signed graph. The graph gives us an abstract representation of the social network or other structured information, and we are now able to solve complex problems or run advanced algorithms on it to solve real-world scenarios. Another scenario we look at are protein similarity networks generated from thousands of protein sequences. We can also transform the data set into graphs. The nodes here represent the proteins, and edges between them give the significance of a match between them. A lower value describes that there is a low probability that a match between these proteins can be found by chance [BBBT08].

One problem, which can be solved on these graphs, is the already mentioned community detection. In the context of graph theory, the problem is also often called node clustering, which is looking for clusters of nodes in the graph that are strongly connected to each other and loosely connected to the rest of the nodes of the graph. While this concept is only very loosely defined, many people have tried to formalize a metric for community detection [CDMG17]. Though there is no metric that is generally accepted as it is hard to consider every aspect of the problem. One of the most popular metrics is the measurement modularity introduced by [NG04], which is based on the number of edges between and within communities compared to all edges in the graph. The authors of [BDG$^+$08] show that maximizing the modularity for communities in a graph is NP-complete and can also be extended to weighted graphs. The modularity is then based on the sum of edge weights between and within communities.

The optimal solution for node clustering would be to find a disjoint set of cliques in the graph, so cluster of nodes that have an edge to every node in its cluster but no edges to other nodes in the graph not in its cluster. Such a graph can then be called a cluster graph. It quickly becomes clear that this definition is too strict for most graphs, and we can not find such a set of cliques in most graphs. This can have different reasons for the information we examine. In social networks, for example, a group of friends is not necessarily fully connected on the network. The people in the group can also have additional connections to other groups. The data we collected could also contain errors. For example, two people in the network are not connected, but our data shows a connection. The same thing could happen the other way around, so our data does not show a connection while there is one between two people. While this concept might seem far-fetched for social networks, this is often the case for biology data sets, for example, the COG similarity data set we examine. The edges contain a weight, which shows a probability if the connection is correct. The basic idea for a graph is now to remove and insert edges to reach an optimal solution. We edit the graph to transform it into a disjoint set of cliques. We can call this approach graph editing and in the context of community detection and finding cliques cluster editing. The goal for unweighted graphs is to find a minimum number of edge edits to reach a disjoint set of cliques. We can additionally consider costs for the edit operations. For this, we expand on the idea of edge weights. We allow edge insertion, which means also non-existing edges need a weight. This way, we reach edit costs that are defined for every node pair in the graph and have to be paid if an existing edge is deleted or a non-existing edge is inserted.

While most networks only contain weights for existing edges, there are examples where we have values for all node pairs. One of these are the already mentioned COG protein similarity networks, where we have probability values for each protein pair. We can take these values to model our edit costs. The goal is then to find a set of edge edits with a minimum sum of edit costs. This way, we first edit edges with a low probability that they are correct in our data set and thereby reduce the error we make by editing the graph. While cluster graphs are the optimal solution for node clustering if all edges should exist in the clusters, different use cases can expect different properties for the solution. For social networks, we can allow that a person can be in more than one community. Looking at examples from social networks, this is often the case. A person can be part of a group of friends on one side, but also part of a family on the other side. We have so-called intersecting communities in a network. While [NG13] introduce quasi-threshold graphs as a model for social networks, the authors of [BHHW21] motivate quasi-threshold graphs as an ideal structure to model disjoint sets of intersecting communities.

In this thesis, we examine quasi-threshold graphs for community detection. They can be defined by their forbidden node-induced subgraphs. They do not contain P_4 or C_4, a path of four nodes or a cycle of four nodes, as node-induced subgraph. A straight-forward algorithm for detecting if a graph is a quasi-threshold graph would therefore be to look if the graph has one of the forbidden subgraphs. If we find one of the forbidden subgraphs the graph is not a quasi-threshold graph. They are also called trivially perfect graphs because it is easy to prove for the set of graphs that they are perfect [Gol78]. Another definition we use is that they can be defined as the transitive closure of a rooted forest. Every node in the quasi-threshold graph is adjacent to its ancestors and descendants in the rooted forest. It allows us to focus on the skeleton of the graph, which implicitly defines which edges are included in the graph.

We already gave a basic idea for an algorithm to recognize if a graph is a quasi-threshold graph. We can test for node-induced subgraphs and decide if the graph fulfills the given conditions. For this thesis, we want to look at graph editing for quasi-threshold graphs. We want to transform the graph into a valid quasi-threshold graph and achieve this by editing the graph. We keep all nodes, but allow inserting and removing of edges between the nodes of the graph to reach a valid solution to the problem. In this thesis, we use the term quasi-threshold editing for the problem of transforming a graph into a quasi-threshold graph via editing. Given a graph, we delete and insert edges to transform the input graph into a quasi-threshold graph. We want to find a set of edge edits that minimizes the sum of edit costs for the given graph.

On the one hand, there exist linear-time algorithms for the recognition of quasi-threshold graphs, based on the forest you can construct for a valid quasi-threshold graph. The algorithm, for example, introduced by the authors of [Chu08], can recognize if the graph is a quasi-threshold graph and construct the forest of the graph. It also finds a node-induced P_4 or C_4 if the graph is not a quasi-threshold graph. Quasi-threshold editing, on the other hand, is NP-hard [NG13]. It therefore seems unlikely that it is possible to design an algorithm that solves the problem exact and scales well for large graphs. Even limiting the problem to uniform edit costs for every node pair does not change this fact. For uniform edit costs, the authors of [GHS+20] introduce two algorithms to solve the quasi-threshold editing problem exactly. The author of [Spi19] introduce a fixed parameter tractable algorithm and an integer linear programming algorithm based on the work of [GHS+20] that solve the quasi-threshold editing problem with nonuniform edit costs exactly and find a set of edits with minimum edit costs to transform a graph into a quasi-threshold graph. The authors show that the solution is optimal while considering edit costs. The experimental evaluation shows that while the algorithms work well on small instances with fewer than 100 nodes, the algorithms can not solve larger graphs in a reasonable amount of time. The algorithms were evaluated on a set of protein-protein similarity networks and were not able to solve even instances with more than 40 nodes exactly.

In this thesis, we instead solve the problem heuristically. We cannot guarantee that the solution overall is optimal, but instead improve the solution iteratively in rounds. In every round, we find the optimal solution based on our constraints. We show that we can reach the optimal solution for most of the graphs in the set of protein similarity graphs, where the exact solver finds one.

The problem of heuristic quasi-threshold Editing was already examined by [NG13], but their heuristic was not scalable for large graphs. The authors of [BHSW15] further looked into the problem and introduce Quasi-Threshold Mover (QTM), an algorithm based on heuristic local moving which improves on the initial solution. The algorithm works on the skeleton of the quasi-threshold graph and works in rounds. In every round, QTM tries to move every node once to a new position in the skeleton forest, while minimizing the necessary edits for the edited graph. QTM only considers uniform edit costs, but already stated the possibility to expand the idea on nonuniform edit costs in its conclusion. The algorithm was engineered to solve the problem inclusion-minimal in the initialization and in linear running time per local move in [BHHW21]. For this, the authors introduce reordering of simple paths as the only operation to change the skeleton forest without changing the implied quasi-threshold graph. We can thus sort the neighbors of the node we move to the top of their simple path. They also introduce randomization to the algorithm. We can randomize the choices for the new position of the local move if they are equivalent in the number of edits. The authors show that this simple path sorting step and randomization during the local move improve the solution quality of QTM. Additionally, the authors improved on the running time of the algorithm.

Our contribution is building on QTM and the improvements introduced by [BHHW21]. Instead of uniform edit costs, we consider edit costs for every node pair. This means we have insert edit costs for non-existing edges and delete edit costs for existing edges. We introduce a Quasi-Threshold Mover with edit costs that solves the quasi-threshold editing problem while considering nonuniform edit costs based on the concept of local moving. We show that with few changes, QTM is able to solve the problem with nonuniform edit costs. Furthermore, we present the algorithm in detail and prove its correctness and running time per node and iteration. Thereby, we focus on areas where differences arise due to nonuniform edit costs. We also present optimizations on QTM with the goal of improving solution quality of the heuristic approach at the cost of additional running time. This is achieved through allowing more complex moves in a single run of QTM. This includes moving a node with local move and then moving the node with its newly adopted children to a new position in the forest and again allowing the adoption of new children at its new position. Therefore, we can use a similar algorithm for this subtree move as for the local move. We prove local optimality and running time for the subtree move optimization.

We implemented the algorithm in C++ and evaluated the algorithm and the subtree move optimization on graphs from the use cases of social networks and protein similarity data. For nonuniform edit costs, we used the COG similarity data set from [TKL97] and run QTM on the connected components found by [BBBT08]. We treat every component as a separate graph. We compared our results with the exact solutions from [Spi19] and could match the exact solution on all but three component graphs. Additionally, we were able to solve the quasi-threshold editing problem on all components, but can not give a statement about the quality of the solution because the exact solver was not able to find a solution for 279 of the graphs. We also show that subtree move, on average, finds better solutions than the basic algorithm, but needs more running time per round. Furthermore, we show that subtree move can also improve the solution quality for uniform edit costs. For this, we run experiments on 100 Facebook graphs where no nonuniform edit costs for all node pairs exist. We show that subtree move can find better solutions on average than the QTM algorithm without the optimization, but at the cost of additional running time. As a last part of the evaluation, we examine the effects different edit costs can have on the quasi-threshold graph, QTM finds. We focus as a case study on one of the Facebook graphs.

This thesis is structured as follows: In Chapter 2, we introduce the necessary theory for the problem. This includes descriptions of the quasi-threshold graph and rooted forests. We also introduce edit costs in comparison to simple edge weights. In Chapter 3, we introduce the base QTM algorithm with the changes considering edit costs. We also prove that the correctness of the algorithm is still given when the edit costs are not uniform. Chapter 4 introduces the subtree move optimization on the QTM algorithm and also the proof of correctness and running time of the operation. Experimental evaluation of the algorithm variations follow in Chapter 5. In the end, we reach a conclusion in Chapter 6 and summarize our finding and possible future work.

2. Preliminaries

In this chapter, we introduce the notation used throughout the thesis. We also introduce the foundations of the thesis in graph theory and introduce concepts that are used for designing the algorithm or in the proofs of correctness and running time.

Graph

A Graph is a tuple $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. All examined graphs are undirected unless otherwise stated and do not contain self-loops, edges with both endpoints identical. They also do not contain multiple edges between one node pair. A weighted graph includes a weight function $w : E \to \mathbb{R}$ that gives every edge a weight. Unweighted graphs can be seen as weighted graphs with $w(e) = 1\ \forall e \in E$. For the editing problem, we also need weights for non-existing edges. We define edit costs for all node pairs of the graph with $cost(\{u, v\}) > 0$ if there is an edge between u and v and $cost(\{u, v\}) < 0$ if there is no edge. This way, we can read the edit cost value and simply decide if u and v are neighbors without further information. We define uniform edit costs as $cost(\{u, v\}) = 1$ for existing edges and $cost(\{u, v\}) = -1$ for non-existing edges.
The neighborhood $N(u)$ of a node u is defined as every node in G that shares an edge with u. In the thesis, we use the term u-neighbor to describe a node as neighbor of u or non-u-neighbor if the node is not a neighbor of u. For a set of nodes S, we can define $N(S)$ as the union of the neighborhoods of all nodes in S, excluding S itself: $N(S) := \cup_{u \in S} N(u) \setminus S$. The number of neighbors of a node u is defined as the degree $deg(u) := |N(u)|$ of u. The maximum degree of a Graph G is defined as Δ.

Forest

A rooted forest is a graph consisting of disjoint rooted trees which are, in themselves connected, and cycle-free. This means that for two nodes in the same tree, exactly one path exists that connects them. Each node in a tree has zero or more children and is the parent of its children. We define $p(u)$ as the parent of node u. In the algorithm, we will use a directed forest where the edges of the nodes point towards the parent in the tree. This way, each tree has a node that has no outgoing edges and is defined as the root of the tree. Descendants of a node u are defined as the nodes reachable by repeatedly going from parent to child starting at u. Ancestors of a node u are defined as the nodes reachable by repeatedly going from child to parent starting at u. The root of a tree T is thus an ancestor to all nodes in T. A leaf of a tree has no further children. The depth of a node in

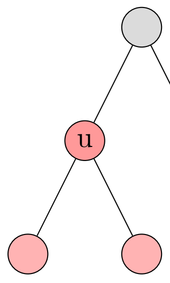
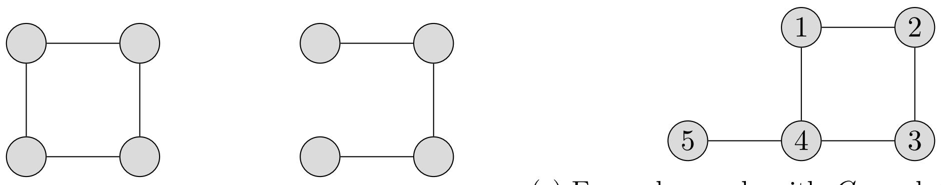


Figure 2.1: Subtree T_u of a tree T.



(a) The basic graph C_4. (b) The basic graph P_4. (c) Example graph with C_4 and C_4 as node-induced subgraphs.

Figure 2.2: Examples for the basic graphs C_4, P_4, and a graph with these graphs as node-induced subgraph.

a tree is defined as the number of edges from the node to the root of the tree. Nodes that share the same depth in a tree are on the same level.

Subtree

A subtree T_u of a tree T is a tree consisting of u and all of its descendants in T. Figure 2.1 show an example for a subtree. It consists of the node u and the descendants of u. A subtree can also contain only one node. If u is the root of the tree T, the subtree T_u is equal to the tree T.

Subgraph

A subgraph H of G is another graph consisting of a subset of nodes from G and a set of edges from G between the nodes. A node-induced subgraph includes all edges in H that connect two nodes of H in G. Figure 2.2 shows examples for the graphs C_4 and P_4, consisting of four nodes and building a cycle or path, respectively. Figure 2.2c contains both graphs as a node-induced subgraphs: C_4 consisting of node 1 to 4 and P_4 consisting of nodes 2 to 5. With the definition of subgraphs, a set of graphs can be defined by their forbidden subgraphs. If a graph does not contain a node-induced subgraph of a set F, the graph is also be called F-free graph.

Quasi-Threshold Graph

With the earlier paragraphs, we can now give a formal definition of a quasi-threshold graph. Quasi-threshold graphs are exactly the $\{P_4, C_4\}$-free graphs. They can also be defined as the transitive closure of a rooted forest. A node u in the quasi-threshold graph is adjacent to all its ancestors and descendants in the rooted forest. A quasi-threshold graph can then implicitly be defined by its skeleton forest. A node in the skeleton forest has an edge to his ancestors and descendants in the graph. Figure 2.3 shows a quasi-threshold graph. The skeleton is represented by the bold edges and the roots by the dark nodes. The transitive closure is depicted with dotted edges. The graph does not have to be connected and can consist of more than one tree.

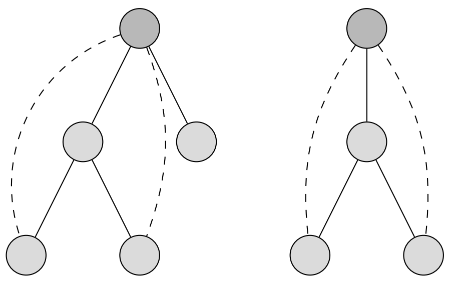


Figure 2.3: Skeleton forest of a quasi-threshold graph and the implied graph.

3. QTM with Edit Costs

This chapter describes the Quasi-Threshold Mover algorithm as introduced by [BHSW15] and the changes to the algorithm that were necessary to allow nonuniform edit costs. In every section, we first introduce the basic behavior of the algorithm and then consider edit costs in the second part.

We introduce the basic idea of the algorithm in Section 3.1, introduce the different initializations possible in Section 3.2, and go into details of the local move in Section 3.3. Then, we outline the simple path sorting step in Section 3.4. We introduce the concept of close children in Section 3.5 and describe how we find the new position for a node in Section 3.6. In Section 3.7, we outline how we can use different edit costs for inserting and removing edges in the algorithm. We then prove the correctness of the algorithm in Section 3.8 and lastly the asymptotic running time of QTM with nonuniform edit costs in Section 3.9.

3.1 Basic Algorithm

QTM transforms a given graph G into a quasi-threshold graph, while heuristically minimizing the necessary edits for this transformation. The algorithm does not change the graph, but instead performs the edits only implicitly. QTM works on the skeleton forest of the graph, which means it first has to generate a rooted forest from the graph. The most trivial initialization for QTM is to simply consider every node as a root of the forest. This implies removing all edges from the graph, which is a valid quasi-threshold graph. QTM then tries to iteratively improve the skeleton in rounds by moving the nodes of the graph in a random order to a new position. Moving a node v_m removes it from its current position in the forest and sets the parent of its children to v_m's current parent. We then insert the node v_m at a new position below a new parent. In every round, we try to move every node once. For each node, we find the best parent that minimizes the necessary sum of edits. If the current position is the best, we find it again and do not move the node. In case there are more than one position that are equal in the necessary edits, QTM has the option to randomize the choice of the parent. QTM also allows adopting children of the new parent. This means we move some of the children of the new parent below the node v_m we moved. Adopting a child can save edits in comparison to not adopting it. If this is the case, v_m always adopts the child. Adopting a child can also make no difference for the edits. Here, we have the option to randomize the choice if v_m should adopt the child.

The algorithm is not optimal as a whole, but is optimal in relation to this local move and finds the optimal parent for the node we move in the given skeleton under the constraint of only moving a single node at once. When considering nonuniform edit costs, the basic approach of local moving can stay the same and still work while considering edit costs. Though, a few changes to the algorithm are necessary to find an optimal solution per local move. Both the decision for the best parent of the local move and which children to adopt have to consider the edit costs for every edge insertion and deletion necessary to perform the move. QTM with edit costs therefore finds for every node an optimal parent that minimizes the necessary sum of edit costs. Additionally, v_m adopts children that save edit costs in comparison to not adopting them. For children where there is no difference in edit costs, if we adopt them or not, we have the option to randomize the choice. The node v_m can adopt a random sample from these nodes. Given that edits are only performed implicitly, the main change of the algorithm is how it calculates edit costs and saved edit costs by a local move, where we will go into detail in the following sections. Furthermore, we introduce metrics similar to [BHSW15] that are based on edit costs, instead of edits where the decisions of QTM are based on. We have to give a note about node pairs $\{u, v\}$ where $\text{cost}(\{u, v\}) = 0$. While we excluded this case for the theory, they exist in practice. We decide to include these node pairs as edges in G and interpret the costs as edit costs for removing these edges. Our algorithm can handle these edges and stay correct. However, we ignore this edge case in the proofs, so we are able to work with a ratio between edit costs.

3.2 Initialization

QTM employs local moving on the skeleton of a graph. The authors of [BHSW15] introduced different initializations to construct an initial skeleton from the input graph. In the experimental evaluation, we show that the initialization influences overall solution quality as well as running time of the algorithm. We used the first possible initialization to introduce QTM. We view every node as a root of a tree in the forest and start the algorithm from there. This corresponds to removing all edges from the graph and implies a valid quasi-threshold graph. The initialization is fast, but, as we will see in the experimental evaluation, often not the best for finding an optimal solution. The second initialization is described as editing and executes an adapted recognition algorithm from [Ham21] on the graph. The algorithm gives back a valid forest of a quasi-threshold graph. The algorithm is based on triangle counting in the graph, which dominates the running time because the rest of the editing heuristic is linear in the graph size. We did not change this editing algorithm to consider nonuniform edit costs. This initialization reaches, in the evaluation often, the best solution, although it is based on uniform edit costs. However, this does come with an increase in running time.

The third initialization inserts the nodes into the graph based on a given order. The node is inserted into the graph by local moving it to its new position while only considering already inserted nodes. Here, different variants exist that vary a lot in solution quality. One variant is, for example, inserting the nodes sorted by their degree in the graph. We insert the nodes of the graph into the forest in ascending order. The best result in the evaluation, however, we reach by randomizing the order of insertion. This initialization is a compromise of the first two. It has a lower running time than the editing initialization while also achieving, on average, better results than simply removing all edges from the graph to reach a skeleton. Another advantage is that this initialization is inclusion-minimal regarding edits. This means that we find a set of edge edits such that it is not possible to reach a valid quasi-threshold graph with a subset of these edits. If a graph is already a valid quasi-threshold graph and can be solved with zero edits, the initialization will always find this solution.

Because we only edit the graph implicitly, we have to calculate the needed edits once for the resulting forest if we want to have the sum of edits as a result. For the functionality of QTM, it is not strictly necessary, but the result would otherwise only include saved edits and edit costs compared to the initial skeleton. During the local move, we simply rely on calculating the edits and edit costs we save by moving a node to its new position. The computation for edits and edit costs can be seen in Algorithm 3.1 as pseudo-code. The algorithm combines both uniform and nonuniform edit costs calculation, as they share the same structure.

We introduce a virtual root r, that is connected to every node in G. The children of r are the roots of the trees in the forest. For uniform edit costs, we perform the depth-first search through each rooted tree of the forest, where we mark a node u on the first visit and count the neighbors of u in the graph that are already marked. These are the neighbors of u in G that are ancestors of it in the same tree and are here called upper neighbors. These edges are necessary for the quasi-threshold graph, but already exist in the graph and do not have to be inserted. Non-existing edges are the difference of upper neighbors and the depth of the node u in the tree. These nodes are not neighbors of u in G, but are ancestors of u in the tree. The edges to these nodes have to be inserted. We sum up the missing and existing edges for all nodes in the tree. When we visit a node again in the DFS, we unmark it and decrease the depth. After the DFS has completed, the sum of missing edges and the difference of number of edges in the graph and existing edges is the number of necessary edits for the given forest. We insert an edge for every missing edge, and remove every edge that does not exist in the transitive closure of the skeleton forest.

For nonuniform edit costs, we can use the same DFS to calculate the deletion edit costs, but simply counting the missing and existing edges is not enough in this case. We instead have to sum up the necessary edit costs for every node pair. When we first visit a node, we add the edit costs of all unmarked neighbors in the graph G to the sum and subtract it for all marked neighbors in G. After the DFS, this way, the sum represents all edges that have to be removed from the graph twice. Simply dividing the sum of edit costs by two gives us the cost for removing these edges. For the cost of inserting edges, all optimizations for uniform edit costs are not correct anymore. We can not use the depth of the node or the number of edges in the graph to calculate the missing edges and insert edit costs. We instead go through every node in the graph and add up insertion edit costs for every ancestor of the node in its tree. If an ancestor of u is not a neighbor in G, we have to insert an edge. In the end, we add the insert edit costs to the remove edit costs. This way, we calculated the sum of edit costs for the initial skeleton forest as a basis for QTM to improve on.

When we consider edit costs for every node pair, it is not necessary to compute the necessary number of edits as well. The decision for the best parent in the forest and which children to attach to v_m, depends only on the current and saved edit costs. In Algorithm 3.1, calculating the edit costs is the more expensive operation, and we only need a constant number of instructions to calculate uniform edits as well. We will see that this is the case for every part of QTM. This is the reason we calculate edits for the transformation, while not strictly necessary. While they are not directly influencing the control flow of the algorithm, they will be valuable for the evaluation in addition to edit costs. They can be used as a further metric to evaluate solutions and, for example, for average edit costs for a given solution.

3.3 Local Move

QTM employs local moving, a concept successfully used in many heuristic algorithms [BHSW15]. Like many of these, QTM works in iterations. In each iteration, all nodes are

Algorithm 3.1: The algorithm for calculating edits and edit costs for the whole initial skeleton forest.

```
1  depth ← 0
2  $\mathrm{cost}_{\mathrm{remove}} \leftarrow 0$
3  $\mathrm{cost}_{\mathrm{insert}} \leftarrow 0$
4  $\mathrm{num}_{\mathrm{missingEdges}} \leftarrow 0$
5  $\mathrm{num}_{\mathrm{existingEdges}} \leftarrow 0$
6  $r$ ← first root
7  while roots not empty do
8      $r$ ← next root                    // start a DFS for every tree in forest
9      $x$ ← first child of $r$
10     while $x \neq r$ do
11         if $x$ not processed then
12             $\mathrm{num}_{\mathrm{upperNeighbors}} \leftarrow 0$
13             for node $v \in N(x)$ do
14                 if $v$ is processed then
15                     $\mathrm{num}_{\mathrm{upperNeighbors}} \leftarrow \mathrm{num}_{\mathrm{upperNeighbors}} + 1$
16                     $\mathrm{cost}_{\mathrm{remove}} \leftarrow \mathrm{cost}_{\mathrm{remove}} - \mathrm{cost}(\{v, x\})$
17                 else
18                     $\mathrm{cost}_{\mathrm{remove}} \leftarrow \mathrm{cost}_{\mathrm{remove}} + \mathrm{cost}(\{v, x\})$
19             $\mathrm{num}_{\mathrm{existingEdges}} \leftarrow \mathrm{num}_{\mathrm{existingEdges}} + \mathrm{num}_{\mathrm{upperNeighbors}}$
20             $\mathrm{num}_{\mathrm{missingEdges}} \leftarrow \mathrm{num}_{\mathrm{missingEdges}} + \mathrm{depth} - \mathrm{num}_{\mathrm{upperNeighbors}}$
21             mark $x$ as processed
22             depth ← depth +1
23             $x$ ← next node in DFS order after $x$ below $r$
24         else
25             unmark $x$ as processed
26             depth ← depth −1
27             $x$ ← next node in DFS order after the subtree of $x$ below $r$
28 for node $u \in G$ do
29     for ancestor $p$ of $u$ do
30         if $p$ is not a neighbor of $u$ then
31             $\mathrm{cost}_{\mathrm{insert}} \leftarrow \mathrm{cost}_{\mathrm{insert}} + \mathrm{cost}(\{p, x\})$
32 $\mathrm{cost}_{\mathrm{all}} \leftarrow \mathrm{cost}_{\mathrm{insert}} + \mathrm{cost}_{\mathrm{remove}} / 2$
33 edits ← $\mathrm{num}_{\mathrm{missingEdges}} + |V| - \mathrm{num}_{\mathrm{existingEdges}}$
```

touched in a random order and moved from their position in the skeleton forest to a new position. With the goal in mind of minimizing necessary edits, the new position has to reduce or at least not change the number of edits needed to transform G in the implied quasi-threshold graph.

For this, the algorithm calculates relative saved edits for every move. When moving v_m, the algorithm has to find a new parent u for v_m and calculate the currently needed edits at the current position to get the saved edits later. In Algorithm 3.2 we see the algorithm for the current edits. Again, we combine the computation of edits and edit costs because the algorithms share the same structure. At first, for uniform edit costs, the algorithm initializes the edits as the number of neighbors of v_m to simulate isolating v_m in the forest. We remove all edges adjacent to v_m. QTM uses a loop through the neighbors of v_m to mark them for later use. Here, we simply can add one to the current edits for every neighbor. Then we subtract one edit for all ancestors and all nodes we find in a DFS starting from v_m that are neighbors of v_m and add one edit for every node that is not a neighbor of v_m. This way, we revert the edits for removing edges that are still used and add edits for inserting needed edges. At the end, we have summed up all edits adjacent to v_m, which are the only ones that can change when moving v_m to a new position.

When considering edit costs for all node pairs, we also have to sum up the currently needed edit costs for v_m to calculate the saved edit costs by the local move. Algorithm 3.2 shows the algorithm for the current edit costs. We have to sum up the edit cost of all neighbors of v_m. The basic QTM algorithm already features a loop that goes through all neighbors to get the number of neighbors and mark them for later use. We simply can reuse this loop to sum up the edit costs for removing all edges adjacent to v_m. For the descendants of v_m we start a DFS at v_m through its subtree and then loop through all ancestors of v_m. We subtract the edit costs of every node we find this way from the sum of edit costs, and save the adjacency test that is needed for uniform edit costs. The way we defined the edit costs, this reverts the edit costs for removed edges that are still used and adds the edit costs for edges to insert. The edit costs are negative for insertions and positive for deletions. Subtracting the edit costs is, then for both scenarios, correct. We use this approach more than once in the algorithm and have to add or subtract the edit costs depending on scenario. After the Algorithm 3.2 we have computed the current edit costs adjacent to v_m.

QTM then isolates v_m as the first step of the algorithm. We move v_m below the virtual root r. The children of v_m get the parent of v_m as new parent, and v_m is the root of its own tree in the skeleton forest. This way, we avoid edge cases where v_m would be considered for its own new parent.

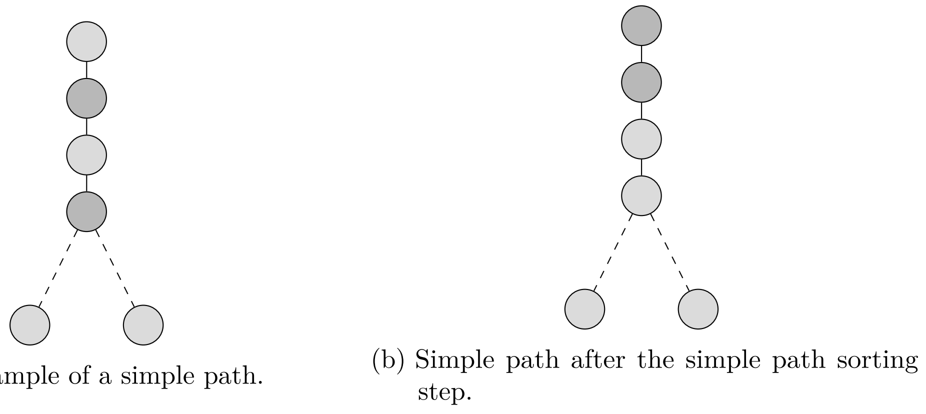
3.4 Simple Path Sorting

The authors of [BHHW21] already engineered QTM for uniform edit costs. One of the optimizations is an additional simple path sorting step. In Figure 3.1, we see an example of sorting a simple path. This step is executed before the node is moved to a new location with the local move. Figure 3.1a shows an example of a simple path before it is sorted. The path consists of four nodes. Every node in the path, excluding the lowest one, only has one child in the tree. The children of the lowest node are not part of the simple path, but are instead representatives for the simple paths below the one shown. A simple path can also only consist of a single node. It is easy to see that we can represent the skeleton forest of a quasi-threshold graph as a set of simple paths.

The authors of [BHHW21] show that it is possible to change the order of simple paths without changing the implied quasi-threshold graph. The idea of simple path sorting is

Algorithm 3.2: The algorithm for calculating current edits and edit costs for v_m so we are able to calculate saved edits and edit costs for the local move.

```
1  $\text{edits}_{\text{current}} \leftarrow 0$
2  $\text{cost}_{\text{current}} \leftarrow 0$
3  for node $u \in N(v_m)$ do
4      $\text{edits}_{\text{current}} \leftarrow \text{edits}_{\text{current}} + 1$
5      $\text{cost}_{\text{current}} \leftarrow \text{cost}_{\text{current}} + \text{cost}(\{u, v_m\})$
6      mark $u$ as neighbor
7  $x \leftarrow$ first child of $v_m$
8  while $x \neq v_m$ do
9      if $x$ not processed then
10         if $x$ is neighbor of $v_m$ then
11             $\text{edits}_{\text{current}} \leftarrow \text{edits}_{\text{current}} - 1$
12         else
13             $\text{edits}_{\text{current}} \leftarrow \text{edits}_{\text{current}} + 1$
14         $\text{cost}_{\text{current}} \leftarrow \text{cost}_{\text{current}} - \text{cost}(\{x, v_m\})$
15         mark $x$ as processed
16         $x \leftarrow$ next node in DFS order after $x$ below $v_m$
17     else
18         unmark $x$ as processed
19         $x \leftarrow$ next node in DFS order after the subtree of $x$ below $v_m$
20 for ancestor $p$ of $v_m$ do
21     if $p$ is neighbor of $v_m$ then
22         $\text{edits}_{\text{current}} \leftarrow \text{edits}_{\text{current}} - 1$
23     else
24         $\text{edits}_{\text{current}} \leftarrow \text{edits}_{\text{current}} + 1$
25     $\text{cost}_{\text{current}} \leftarrow \text{cost}_{\text{current}} - \text{cost}(\{p, v_m\})$
```


(a) Example of a simple path.

(b) Simple path after the simple path sorting step.

Figure 3.1: Sorting of a simple path of the skeleton forest. Neighbors of v_m are colored in dark-gray.

now to sort v_m-neighbors to the top of their respective simple paths in the forest. For every neighbor u of v_m we check the simple path of u. We move u to the highest position in its simple path that is not a neighbor of v_m. We can achieve that by simply switching the nodes in the simple path. Figure 3.1b shows the simple path after the sorting step. Neighbors of v_m are shown in a darker color and are moved to the top of the simple path. A potential parent now has the maximum number of neighbors in its ancestors, without changing the forest. For uniform edit costs, this step improves the quality of solutions, as shown by [BHHW21]. QTM finds for v_m the position in the skeleton forest that has the minimum number of edits adjacent to v_m. While the authors did not examine nonuniform edit costs, it is easy to see that this step makes sense when considering edit costs. Sorting simple paths does not change the implied quasi-threshold graph. We do not insert or remove an edge in this step. Thus, the step does not need edit costs. We want as many neighbors to v_m as ancestors for a potential parent as possible. Sorting simple paths improves the solution quality for nonuniform edit costs. It is not necessary to change this step because it does not consider edit costs, but only relies on adjacency information for nodes. Preliminary experiments show that this step also reaches the best solutions with nonuniform edit costs. We use the simple path sorting step for all experiments in the evaluation.

3.5 Edit Costs of Close Children

When moving v_m, we have to find the optimal position in relation to edits necessary to move v_m to. Then we decide if we want to adopt children of the new parent u, which means placing them below v_m instead of below the parent u. We adopt children if we can save edits compared to not adopting them. For this decision, the authors of [BHSW15] introduce a metric called child closeness as $\text{child}_{\text{close}}(c)$, which considers if a node c has more neighbors than non-neighbors of v_m below it. Nodes with a positive child closeness are called close children and should be adopted if v_m chooses their parent as the new parent. Nodes with a child closeness of zero do not impact the edits necessary. While [BHSW15] ignored these nodes in the first version of QTM, the authors of [BHHW21] introduce randomization for the so-called indifferent children as an optimization. QTM randomly samples from indifferent children to reach better solutions. [BHHW21] shows in their experiments that randomization finds better solutions for uniform edit costs. Again, preliminary experiments show that this is also the case for nonuniform edit costs. We will use randomization in all experiments in our evaluation.

QTM uses a priority queue to compute child closeness values and find the best parents for v_m. We use the tree depth of the node as the key for the priority queue and process the nodes in descending order. A potential parent is either a neighbor of v_m or has a close child himself. We first add all neighbors of v_m to the queue. We process the nodes in a bottom-up approach and propagate positive child closeness values to the top. First, we add parents of close children to the queue. We then calculate the child closeness values with many small DFSs. To avoid visiting a subtree twice, we save DFS pointers as DFS_{next} to the last visited nodes to skip already performed DFSs. Because we propagate positive child closeness values of processed nodes to the top, and we already processed all neighbors of v_m below a non-processed node u, child closeness of u can only decrease in the DFS. We can abort the DFS if we ever reach a negative child closeness value and save a pointer in DFS_{next}.

The algorithm for processing a node u in the priority queue is shown in Algorithm 3.3. Here, we again calculate both edit costs and the edits for uniform edit costs. We first focus on uniform edit costs. To calculate child closeness for uniform edit costs, QTM starts a DFS for every potential close child in the queue. For every node x we find in the DFS, we

Algorithm 3.3: The algorithm for calculating $\mathrm{child}_{\mathrm{close}}(u)$ and $\mathrm{child}_{\mathrm{cost}}(u)$. We will use $\mathrm{child}_{\mathrm{cost}}(u)$ to decide if we want to adopt u.

```
1  $\mathrm{child}_{\mathrm{close}}(u) \leftarrow \sum$ over $\mathrm{child}_{\mathrm{close}}$ of close u-children
2  $\mathrm{child}_{\mathrm{cost}}(u) \leftarrow \sum$ over $\mathrm{child}_{\mathrm{cost}}$ of close u-children
3  if $u$ is $v_m$-neighbor then
4      $\mathrm{child}_{\mathrm{close}}(u) \leftarrow \mathrm{child}_{\mathrm{close}}(u) + 1$
5  else
6      $\mathrm{child}_{\mathrm{close}}(u) \leftarrow \mathrm{child}_{\mathrm{close}}(u) - 1$
7  $\mathrm{child}_{\mathrm{cost}}(u) \leftarrow \mathrm{child}_{\mathrm{cost}} + \mathrm{cost}(\{u, v_m\})$
8  if $\mathrm{child}_{\mathrm{cost}}(u) \geq 0$ and $u$ has children then
9      $x \leftarrow$ first child of $u$
10     while $x \neq u$ do
11         if $x$ not processed or $\mathrm{child}_{\mathrm{cost}}(x) < 0$ then
12             if $x$ processed then
13                 $\mathrm{child}_{\mathrm{close}}(u) \leftarrow \mathrm{child}_{\mathrm{close}}(u) + \mathrm{child}_{\mathrm{close}}(x)$
14                 $\mathrm{child}_{\mathrm{cost}}(u) \leftarrow \mathrm{child}_{\mathrm{cost}}(u) + \mathrm{child}_{\mathrm{cost}}(x)$
15                 $x \leftarrow \mathrm{DFS}_{\mathrm{next}}(x)$
16             else
17                 $\mathrm{child}_{\mathrm{close}}(u) \leftarrow \mathrm{child}_{\mathrm{close}}(u) - 1$
18                 $\mathrm{child}_{\mathrm{cost}}(u) \leftarrow \mathrm{child}_{\mathrm{cost}}(u) + \mathrm{cost}(\{x, v_m\})$
19             if $\mathrm{child}_{\mathrm{cost}}(u) < 0$ then
20                 $\mathrm{DFS}_{\mathrm{next}}(u) \leftarrow x$
21                 break
22             $x \leftarrow$ next node in DFS order after $x$ below $u$
23         else
24             $x \leftarrow$ next node in DFS order after the subtree of $x$ below $u$
```

can decrease the child closeness by one if the node was not processed because the child is not a neighbor of v_m. If the node is already processed, we can simply add the child closeness of the child to the child closeness of u. The child closeness is always negative in that case, which is equivalent to decreasing the child closeness. We can abort the DFS for a node u if its child closeness ever decreases below zero, because the DFS can only decrease the child closeness of u. In this case, we can store a pointer for the unfinished DFS to avoid running a DFS through the same nodes twice. $\mathrm{DFS}_{\mathrm{next}}$ points in this case to the last visited node. If a DFS finishes, it means that $\mathrm{child}_{\mathrm{close}}(u) \geq 0$ and u is a close or indifferent child. We then propagate positive child closeness to the parent of u and add the node to the priority queue as a potential parent.

For uniform edit costs, counting neighbors and non-neighbors is correct, but for nonuniform edit costs, this does not consider the edit costs for every node pair. So we have to introduce a metric that sums up the edit costs we save for a child if it is moved below v_m compared to not adopting it and leaving u as its parent. The node keeps all its ancestors and children, which means we only have to consider the edit cost to v_m. We introduce the metric as child cost or $\mathrm{child}_{\mathrm{cost}}(c)$ for a node c and the edit cost of a node c to v_m as $\mathrm{cost}(\{c, v_m\})$. Analog to the uniform case, we call nodes with a positive child cost close children of v_m and children c with a $\mathrm{child}_{\mathrm{cost}}(c) = 0$ indifferent. We calculate the child cost of close children similar to the child closeness in [BHSW15]. We start with all neighbors of v_m in a priority queue, with the tree depth of the nodes as key in a bottom-up approach. Then we process all nodes in the queue and start many depth-first searches. Furthermore, we store pointers for each node for the last processed node by a DFS to skip already-performed DSFs as $\mathrm{DFS}_{\mathrm{next}}$. The algorithm for processing a node u is shown in Algorithm 3.3, where we again calculate uniform edits, although not strictly necessary for the QTM when considering edit costs. Every non-processed node u in the priority queue is either a neighbor of v_m or has a child with positive child costs. In both cases, we add the edit costs of u to v_m to the child costs of u. We again can save the test if u is a neighbor of v_m because of our definition of edit costs. If the child cost is still positive after this, we start a DFS at u. Because of the bottom-up approach and the saved DFS pointers, we will skip already-explored subgraphs with positive edit costs. The child cost of u can thus only stay the same or decrease because there are no neighbors of v_m below u that have not been already processed by the priority queue.

In the DFS, we add child costs of a processed child x to the child costs of u. The child cost of x is, in this case, always negative, and this is equivalent to decreasing the child cost of u. We can skip the already explored subtree with $\mathrm{DFS}_{\mathrm{next}}(x)$. For non-processed children, we add the edit cost of the child to v_m to the child cost of u. We can here skip a test if the child is a neighbor of v_m because this is already given through the sign of the edit costs. We decrease $\mathrm{child}_{\mathrm{cost}}(u)$ here because we already processed all neighbors of v_m. Again, we can abort the DFS if the child costs of u ever goes below zero and save the pointer $\mathrm{DFS}_{\mathrm{next}}$ in this case. The DFS only finishes when $\mathrm{child}_{\mathrm{cost}}(u) \geq 0$ and u is close or indifferent child. We then propagate positive child costs to the parent of u and add the node to the priority queue as a potential parent. We now have computed $\mathrm{child}_{\mathrm{cost}}$ values for all close children in the skeleton forest. The child closeness calculation is necessary so that we can also get uniform edit as a result of QTM that can be used for evaluation, while the child costs are used for the decision if we adopt a child. The $\mathrm{child}_{\mathrm{cost}}$ values also play a role when we look for the best parent in the algorithm.

3.6 Max Edit Cost

To find the best parent for the local move, [BHSW15] introduces $\max_{\mathrm{score}}$ to recursively calculate the edits saved if v_m would choose a node as parent. For a potential parent p

Algorithm 3.4: The algorithm for calculating $\max_{\text{cost}}$ and $\max_{\text{score}}$ and finding potential parents for the local move.

```
for $v_m$-neighbor $u$ do
    push u
while queue not empty do
    $u \leftarrow$ pop
    determine $\text{child}_{\text{cost}}(u)$ and $\text{child}_{\text{close}}(u)$ by DFS
    $a \leftarrow$ max over $\max_{\text{score}}$ of reported $u$-children
    $b \leftarrow \sum$ over $\text{child}_{\text{close}}$ of close $u$-children
    $x \leftarrow$ max over $\max_{\text{cost}}$ of reported $u$-children
    $y \leftarrow \sum$ over $\text{child}_{\text{cost}}$ of close $u$-children
    if $x = y$ then
        coin $\leftarrow$ randomization
        if coin then
            $\max_{\text{score}}(u) \leftarrow b$
            $\max_{\text{cost}}(u) \leftarrow y$
        else
            $\max_{\text{score}}(u) \leftarrow a$
            $\max_{\text{cost}}(u) \leftarrow x$
    else
        $\max_{\text{score}}(u) \leftarrow max\{a, b\}$
        $\max_{\text{cost}}(u) \leftarrow max\{x, y\}$
    if $u$ is $v_m$-neighbor then
        $\max_{\text{score}}(u) \leftarrow \max_{\text{score}}(u) + 1$
    else
        $\max_{\text{score}}(u) \leftarrow \max_{\text{score}}(u) - 1$
    $\max_{\text{cost}}(u) \leftarrow \max_{\text{cost}}(u) + \text{cost}(\{u, v_m\})$
    if $\text{child}_{\text{cost}}(u) > 0$ or $\max_{\text{cost}}(u) > 0$ then
        report $u$ to $p(u)$
        push $p(u)$
```

$\max_{\text{score}}$ is the sum of positive child closeness over the children of p and the edits saved by p if p is the best parent in its subtree. If p is not the best parent in its subtree, it saves a reference to the best parent and $\max_{\text{score}}$ is the saved edits for this best parent.

Similar to this, we introduce the maximum edit costs saved if node u is chosen as parent as $\max_{\text{cost}}(u)$. Given through the definition of quasi-threshold graphs, choosing node w as parent for v_m requires the quasi-threshold graph to have edges from v_m to all ancestors of w and to all descendants of close children of w. We define this set of nodes as X_w. For uniform edit costs, we simply can maximize the number of neighbors minus non-neighbors of v_m in X_w. While considering edit costs, we instead have to maximize the edit costs to v_m in X_w. It is, for example, possible to have positive edit costs with more non-neighbors than neighbors of v_m in X_w. We can calculate the max cost recursively. For this, we first only examine edit costs and best parents in a subtree T_u restricted to u and its descendants. If we now want to decide on the best parent of a subtree T_a of an ancestor a of u all best parents in the subtree share the edit costs from a to v_m.

Algorithm 3.4 shows the algorithm for calculating $\max_{\text{cost}}$ for the local move. As calculating max cost and child cost share the same bottom-up approach, we can combine them. We use the same priority queue for both computations. We start by adding all neighbors of v_m

to the queue. When we process a node, we first sum up the child cost of u-children, that are not negative. If a node u is chosen as parent, we only adopt children when their child costs is not negative. We therefore can limit us to positive child costs. As a next step, we test if the sum of child cost is greater than all max costs of u-children. If this is the case, u is the best parent in its subtree, and we initialize max cost of u with the sum of child costs and add the edit costs of u to v_m. Otherwise, we take the maximum of max cost over all children and initialize max cost of u with it. Then we add the edit costs of u to v_m to it. One special case we have here is when $x = y$. In this case, it does not matter if we choose u or the best parent in T_c of child c as best parent. We can randomize the choice of the parent here. If the child cost or max cost of u is positive, the parent of u is a potential parent for the local move. We therefor report u to its parent and add p to the priority queue.

After the queue is empty, we have found the optimal parent. Max cost in the root represents max costs for the best parent in the skeleton. We have to simply compare the value to the current edit costs to decide if QTM should move the node to the new position. There, we can use the child cost for the adoption decision. The node v_m adopts all close children of the best parent. This saves edit costs compared to not adopting them. We also do not adopt children c with $\text{child}_{\text{cost}}(c) < 0$. It saves edit costs when we do not adopt them. We can then randomly choose from the indifferent children to adopt at least two children. Adopting one child is equivalent to choosing the child as parent. We want to avoid equivalent solutions for the randomization so that we can reach a point where QTM can again find an improvement. In the case where we adopt all children of the best parent, the position of v_m in its new simple path does not matter. QTM moves v_m by default to the top of its simple path, which does not make a difference here, but that changes when we look at the optimizations for QTM.

3.7 Edit Cost Ratio

We already introduced in the previous sections how we have to adopt QTM to be able to solve the quasi-threshold editing problem with nonuniform edit cost. In this case, we can have different edit costs for every node. One scenario we can also easily represent with this approach is different edit costs for removing and inserting an edge. This means $\text{cost}(\{u, v\}) = r$ if $\{u, v\}$ is an edge in G and $\text{cost}(\{u, v\}) = -i$ if $\{u, v\}$ is not an edge in G. The variables i and r represent positive edit costs for inserting and removing an edge, respectively. One advantage of this approach is that we can define edit costs for graphs, where we do not have edit costs. We can evaluate different edit costs for graphs where we before could only evaluate uniform edit costs. The absolute values do not matter as much as does the ratio of the edit costs, as we show in Section 5.6.

While a ratio for the edit cost seems like a special case of nonuniform edit costs, we can implement the ratio in QTM close to the uniform case. During the QTM algorithm, we know for every edit if we remove or insert an edge. For the edit costs of these edits, we then only multiply the edits by i or r, depending on if we remove or insert an edge. During the implementation, we can simply check if we have edit costs for every node or edit costs for removing and inserting edges and then use the correct calculation. As these changes are only multiplications at a few lines in the algorithm, we did not include them in the pseudo-code. QTM is thus able to handle different edit costs for inserting and removing edges, which we use in Section 5.6 to compare different ratios of edit costs to each other. We show that the ratio has an influence on the result of QTM. For the following proofs, we handle the edit cost ratio as a special case of nonuniform edit costs. The proofs do not need to be adapted to prove that QTM can solve the quasi-threshold editing problem with an edit cost ratio. The proof of the running time already uses a ratio for the edit cost, which in this case is already fixed to a certain value. We can prove the same running time for the case of an edit cost ratio.

3.8 Correctness

We have to show that the algorithms find a correct solution to the quasi-threshold editing problem while minimizing the necessary edit costs per local move, and thus is able to find a solution to the problem that is close to optimal. We show that QTM for every move finds the parent with the lowest possible edit costs necessary to move v_m there, while considering the edit costs we save by adopting children of the parent. [Ham21] proves this for uniform edit costs, while we show it still holds true when considering different edit costs for every node pair. We follow the structure of the proofs in [Ham21] very closely, and show very similar theorems are correct when considering nonuniform edit costs. We show that the algorithm processes all potential parents and finds all close children for v_m in the graph. For this, we show that we compute the $\text{child}_{\text{cost}}$ and max_{cost} values correctly for all nodes in the graph. We argue, with structural induction, that because of the bottom-up approach of QTM nodes below a non-processed node u are already correctly processed.

We additionally show that a stronger theorem from [BHHW21] holds true: QTM finds the optimal edits adjacent to v_m with the local move. The authors examine uniform edit costs, but this specific theorem does not rely on the uniform edit costs. It is easy to show that the proof from [BHHW21] is also correct for nonuniform edit costs.

For proof of correctness, we first show that the algorithm processes all close children and calculates correct child costs. As a reminder, edit costs for non-existing edges are negative. If $\text{cost}(\{u, v\}) = 0$ for a node pair $\{u, v\}$ there exists an edge for the nodes u and v in the graph G. We also use the virtual root r in this proof, which has an edge to all nodes in the graph.

Theorem 3.1. *For a node u, $\text{child}_{\text{cost}}(u)$ is either the sum of edit costs to v_m in the subtree of u, or the sum of edit costs to v_m is negative for the nodes in the subtree of u. In the second case, if u has been processed by Algorithm 3.4, then $\text{child}_{\text{cost}}(u) < 0$ and $\text{DFS}_{\text{next}}(u)$ points to a node in the subtree of u and child$_{cost}(u)$ is the sum of edit costs to v_m over all nodes in DFS order between u and $\text{DFS}_{\text{next}}(u)$. The sum includes edit costs to v_m of u, $\text{DFS}_{\text{next}}(u)$ and the edit costs of subtrees of close children in DFS order between u and $\text{DFS}_{\text{next}}(u)$*

Proof. We first show that all close or indifferent children of the graph are processed by Algorithm 3.4. A node u can only have $\text{child}_{\text{cost}}(u) \geq 0$ if it is a neighbor of v_m or one of its children is close. For the second case, one of the descendant has to be a neighbor of v_m. We push all neighbors in the queue, and they are therefore processed. If the node u is not a neighbor, one of its children c has to be close. The node c then pushes the parent of c into the queue and is thus also processed. After the queue is empty, all nodes u with $\text{child}_{\text{cost}}(u) \geq 0$ are processed.

We now have to show that the Algorithm 3.3 correctly computes $\text{child}_{\text{cost}}(u)$. With the same basic idea as [Ham21], we show this by structural induction. If no node below u has been processed, it means that u is a neighbor of v_m, because we first push neighbors and parents of processed nodes into the queue. We initialize $\text{child}_{\text{cost}}(u)$ as $\text{cost}(\{u, v_m\})$ as there are no close children of u. We do not need a DFS if there are no nodes below u and $\text{child}_{\text{cost}}(u)$ is correct. If there are nodes below u we start the DFS. For every node x in the DFS, we add $\text{cost}(\{x, v_m\})$ to $\text{child}_{\text{cost}}(u)$ (with $\text{cost}(\{x, v_m\})$ negative as x is not a neighbor of v_m) as they are not processed. For non-processed nodes, $\text{DFS}_{\text{next}}(x)$ pointers are not changed, which is why we can ignore line 15. We abort the DFS if $\text{child}_{\text{cost}}(u) < 0$ and set $\text{DFS}_{\text{next}}(u)$ to the current node x. We correctly summed up the edit costs for all non-neighbors below u including x.

Now we assume there are nodes below u that are already processed. For these nodes, Theorem 3.1 holds true. We only insert parents of nodes and neighbors of v_m into the queue, and then all nodes with $\text{child}_{\text{cost}}(u) \geq 0$ below u are already processed. We first sum up the $\text{child}_{\text{cost}}$ of close children and the edit costs of u to v_m. If $\text{child}_{\text{cost}}(u)$ is now negative, we can skip the DFS because all nodes below u we did not consider have a negative $\text{child}_{\text{cost}}$.

If $\text{child}_{\text{cost}}(u) \geq 0$, we perform the DFS where we have three different cases for the visited node x:

1. For node x, $\text{child}_{\text{cost}}(x) \geq 0$. The node was already processed and the $\text{child}_{\text{cost}}(x)$ is correct. We already summed up the edit costs in the algorithm and can directly skip to the next node in DFS order.

2. Node x has not been processed. Then x is not neighbor of v_m and we add $\text{cost}(\{x, v_m\})$ to $\text{child}_{\text{cost}}(u)$.

3. Node x has been processed and $\text{child}_{\text{cost}}(x) < 0$. For this node, the second part of Theorem 3.1 holds true. We add $\text{child}_{\text{cost}}(x)$ to $\text{child}_{\text{cost}}(u)$ which correctly considers all nodes between x and $\text{DFS}_{\text{next}}(x)$. We then visit the next node after $\text{DFS}_{\text{next}}(x)$ in DFS order, because we already considered $\text{DFS}_{\text{next}}(x)$

We have two situations that can end the DFS. On the one hand, we have visited all nodes in the DFS and the $\text{child}_{\text{cost}}(u)$ is correct. On the other hand, we aborted the DFS because we reached $\text{child}_{\text{cost}}(u) < 0$ at some point. Then we set the DFS pointer of u accurately and $\text{child}_{\text{cost}}(u)$ correctly represents the sum of edit costs for the nodes between u and the DFS pointer. We thus have shown that the Theorem 3.1 is correct for u. □

The following theorem from [Ham21] holds true for nonuniform edit costs and shows that we do not have to consider every node as a parent for v_m:

Theorem 3.2. *Only nodes with close children and neighbors of v_m need to be considered as parents of v_m.*

Proof. We show this by contradiction and assume that the best parent u has no close children and is not a neighbor of v_m. This means $\text{cost}(\{u, v_m\}) < 0$, and we can not attach a child to save edit costs. Then we can choose the parent of u as parent and save edit costs because $\text{cost}(\{u, v_m\}) < 0$. $P(u)$ is then a better parent for v_m than u and u is not the best parent. This is a contradiction to our assumption. □

We now have shown that we process all potential parents for v_m and have considered the edit costs below a potential parent u. We can save the sum of $\mathit{child}_{\mathit{cost}}$ over all close children of u. Now we have to examine the edit costs for ancestors of the node u. We show the following theorem that is close to one proven by [Ham21] for uniform edit costs.

Theorem 3.3. *For the subtree T_u of a node u, u has been processed by Algorithm 3.4 and $\text{max}_{\text{cost}}(u)$ is correctly calculated or $\text{max}_{\text{cost}}(u) < 0$ for the subtree T_u.*

Proof. As a start, we show that if $\max_{\text{cost}}(u) \geq 0$, u has already been processed. With $\max_{\text{cost}}(u) \geq 0$, u is either the best parent or there is a node below u that is the best parent. If u is the best parent, we processed u because of Theorem 3.2. If u is not the best parent, we have a best parent b below u. The $\max_{\text{cost}}(u)$ is then the sum of edit costs for v_m to X_b restricted to T_u and there is no ancestor of b in T_u with a better $\max_{\text{cost}}$. As b is the best parent, Theorem 3.2 tells us that b is processed. We then also process the parent $p(b)$ and $\max_{\text{cost}}(p(b)) \geq \max_{\text{cost}}(b) + \text{cost}(\{p(b), v_m\})$. Here, $\text{cost}(\{p(b), v_m\})$ can be negative if $p(b)$ is not a neighbor of v_m. We continue with the next parents until we are at u and are finished or reach a node where $\max_{\text{cost}}(a) \leq 0$ and $a \neq u$ and $a \neq b$. Then $p(a)$ is at least as good as b as parent. This is the case because the sum of edit costs of v_m to X_b restricted to T_a is equal or lower than zero, as seen by $\max_{\text{cost}}(a) \leq 0$. Therefore, $\max_{\text{cost}}(p(a))$ is at least as good. This contradicts our assumption that there is no ancestor of b with a better $\max_{\text{cost}}$. We consequently processed all nodes between b and u.

Similar to [Ham21] and $\max_{\text{score}}$ we prove that we calculate $\max_{\text{cost}}$ correctly by structural induction. If no node below u has been processed, u is a neighbor of v_m and has no close children. As x and y are 0 in line 25, $\max_{\text{cost}}(u)$ is $\text{cost}(\{u, v_m\})$ as we only save the deletion of edge $\{u, v_m\}$ for u. Theorem 3.3 holds true for all processed nodes below u. As all nodes in T_u share u as a common ancestor and therefore $\text{cost}(\{u, v_m\})$ influences them the same way, we simply have to decide if u is the best parent or the best parent for T_u is the best parent of T_c of a child c of u. For the first case, we compute the sum of $\text{child}_{\text{cost}}$ over all close children. For the second case, we take the maximum of $\max_{\text{cost}}$ over all reported children and the respective parent that belongs to the $\max_{\text{cost}}$. The maximum of x and y in line 25 thus correctly chooses the $\max_{\text{cost}}$ and best parent. Additionally, we add the edit costs of u to $\max_{\text{cost}}$ and have proven Theorem 3.3 for node u. □

We continue the calculation until we reach the virtual root r, which is processed as a neighbor of v_m. As T_r consists of the whole graph, we have the best parent of the graph at r. We consequently have shown that QTM finds the optimal parent for the local move and children to adopt while considering nonuniform edit costs.

Furthermore, we can show a stronger theorem that was introduced by [BHHW21]. QTM finds the optimal edits adjacent to the node we move for the given forest. While the authors assumed uniform edit costs, we can easily show that this holds true for nonuniform edit costs. This is only the case when QTM sorts simple paths. We follow the proof of [BHHW21] closely and consider edit costs where necessary.

Theorem 3.4. *Consider a graph $G = (V, E)$, a node $u \in V$ and a skeleton forest T. T^- is T with u removed and simple paths reordered such that neighbors of u are at the top of their simple paths. Executing QTM on T^- minimizes the edit costs incident to u.*

Proof. We start with Q as the quasi-threshold graph, with the minimum edit costs incident to u and T_Q as its skeleton forest. T_Q^- is the forest T_Q with u removed and u's children are below u's parent. We keep every other edge, and every node keeps its ancestors and descendants. QTM finds in this situation the best parent and adopted children while considering edit costs, as we have shown before. This means QTM finds a set of edits with a minimum of edit costs incident to u in T_Q^-. The authors of [BHHW21] show that the trees T_Q^- and T^- only differ in the ordering of their simple paths. We now have to show analogous to [BHHW21] that the orderings of T_Q^- and T^- are equally good, while considering edit costs. We closely follow the proof of [BHHW21].

We now look at the parent p and the children C of u in T_Q. Assuming p is the lower end of its simple path in T_Q^-, we get the same ancestors with the lowest node in p's simple path in T^-. The same is the case for descendants when we adopt a child c. We get the same descendants if we adopt the highest node in c's simple path in T^-. Then the order of the simple paths does not matter in this case. If p is not the lower end of its path, we have to distinguish between two cases: p only has one child, which we adopted, or u is a leaf in T_Q. In the first case, the order of p's simple path does not matter. All different orders and positions for p in the simple path yield the same neighbors and thus the same edit costs. In the second case, the order of the simple paths does matter. As u is a leaf in T_Q and p is not the lower end of its simple path, u needs an edge to p and p's ancestors, but not to the nodes below p in p's simple path in T_Q. We call p's simple path P_p. We have to insert an edge for every non-u-neighbor in p's ancestors in P_p and p itself, and we have to remove an edge for every u-neighbor below p in P_p. All this edits imply edit costs. If we sort all neighbors to the top of P_p and choose the lowest u-neighbor in P_p as parent, we minimize the edit costs among all orders of P_p as this implies no edits in P_p. QTM thus finds the best parent and adopted children in T^- to minimize the edit costs incident to u. $\square$

3.9 Running Time

In the previous section, we have shown that QTM solves the quasi-threshold editing problem with nonuniform edit costs. We now show that the running time of the algorithm is $\mathcal{O}(m\eta \log(\Delta))$ with η as a factor depending on the edit costs of the graph and Δ as the maximum degree of the graph G. We define η as follows:

$$\eta = \frac{\max\limits_{u,v\in V}(|cost(\{u,v\})|)}{\min\limits_{u,v\in V}(|cost(\{u,v\})|)} \; \forall u,v \in V : \; \text{cost}(\{u,v\}) \neq 0$$

For this proof, we assume $\text{cost}(\{u,v\}) \neq 0$ for all node pairs in G to avoid edge cases and dividing by zero for the edit cost fraction. The algorithm has a running time per node of amortized $\mathcal{O}(\eta d \log(d))$ with d as the degree of the node. The running time differs from uniform edit costs that are shown by [Ham21]. For uniform edit costs, the authors show a running time of $\mathcal{O}(d \log(d))$ per node, which is the special case of uniform edit costs and $\eta = 1$. We also consider a different approach for processing the nodes in a bottom-up approach. The authors of [BHHW21] show for uniform edit costs that we can use a bucket queue limited to the depth of $2 \cdot deg(u)$ as we can ignore deep neighbors. This is not the case for nonuniform edit costs. We can instead use a different approach where we use a dynamic array of nodes per level in the tree and sort the initial array of the neighbors only once. This results in a running time of $\mathcal{O}(\eta d + d\log(d))$.

As the factor of edit costs depends on the bit length representing the edit costs, which means the running time of the algorithm is only pseudo-polynomial, we additionally show that the algorithm has a worst-case running time of $\mathcal{O}(n \log(d))$ per node and $\mathcal{O}(n^2 \log(\Delta))$ per iteration. Using the dynamic array approach to process the nodes we can lower this to $\mathcal{O}(n + d\log(d))$ per node and $\mathcal{O}(n^2 + n\Delta \log(\Delta))$ per iteration. The edit costs in practice, though, have a low η, and we are below the worst-case running time.

For the whole QTM algorithm, we keep tree depth values for every node in the forest. They show the depth of the node in their respective tree. In the algorithm, we update them when we move a node v_m. We have to decrease the depth of the descendants of v_m at the original position and increase the depth of the descendants of v_m at the new position after the move. A node has to be adjacent to all its ancestors and descendants in the result graph. Every edge of these that does not exist has to be inserted. For uniform edit costs, this means that half of the descendants and ancestors at the new position have to be adjacent to v_m. Otherwise, it is better to choose r as parent. For nonuniform edit costs, this does not hold true. We introduce the fraction between the maximum and minimum of the absolute value for the edit cost over all edit costs as η. At the new position, a minimum of $\frac{1}{2\cdot\eta}$ of the ancestors and descendants have to be adjacent to v_m. Otherwise, r is the better parent for v_m. We can update the depth values at the new position in $\mathcal{O}(\eta d)$ because they are at most $2 \cdot \eta \cdot d$ nodes below v_m at the new position. For the special case of uniform edit costs, η is 1, which is what [Ham21] have shown for QTM.

For the original position, we will do an amortized analysis. The initial edit costs can never exceed the number of edges times $\max\limits_{u,v\in V}(|\text{cost}(\{u,v\})|)$ which is equivalent to removing all edges. We scale the edit cost with $\min\limits_{u,v\in V}(|cost(\{u,v\})|)$ to only consider η for the rest of the proof. We give every node tokens for every neighbor in the edited graph. As the minimum edit costs for all edges is now 1, we cannot insert more edges than the total edit costs. The total edit costs are maximal ηm, and so we initially distribute $\mathcal{O}(\eta m)$ tokens. When we move a node, it generates tokens for new neighbors and itself, in total, worst case $2 \cdot \eta \cdot deg(v_m)$ tokens. The node can thus account for updating the depth of descendants at its original position with tokens. We generate $\mathcal{O}(\eta m)$ tokens per round. We can also account for calculating needed and saved edit costs and updating pointers to parents and children with the same argument. Likewise, we can do all this in amortized $\mathcal{O}(\eta d)$ per node and $\mathcal{O}(\eta m)$ per iteration. Now we only have to show we can find the new parent and children in time $\mathcal{O}(\eta d \log(d))$ per node. We initially insert d nodes in the queue and over the course of the algorithm, we only insert $\mathcal{O}(\eta d)$ in the queue and need amortized constant time when processing a node. The standard priority queue we use needs $\mathcal{O}(\log(n))$ time per operation, where n is the number of nodes in the queue. The proof of running time follows closely the proof of [Ham21]. The main difference is that we insert more than $\mathcal{O}(d)$ nodes in the queue.

The basic idea for the proof is that we give $2 \cdot \eta$ tokens to a node, which the node can use to visit descendants in the DFS or pass up to ancestors. The tokens are represented by the initialization of $\max_{\text{cost}}$ and $\text{child}_{\text{cost}}$ where η is an upper bound. Tokens are consumed if we process a node u that is not a neighbor of v_m. The tokens were passed up by close children of u or children with $\max_{\text{cost}} > 0$. After processing u, we pass up the rest of the tokens to the parent of u. First, we examine Algorithm 3.4 and ignore the DFS for $\text{child}_{\text{cost}}(u)$. We set $\max_{\text{cost}}(u)$ as the maximum of x and y and add $\text{cost}(\{u, v_m\})$ to it. We also initialize $\text{child}_{\text{cost}}(u)$ as $y + \text{cost}(\{u, v_m\})$ and only decrease it afterwards. Then x is an upper bound for y and $\text{child}_{\text{cost}}(u) \leq \max_{\text{cost}}(u)$. The tokens represented in $\max_{\text{cost}}(u)$ are thus always passed up by $\max_{\text{cost}}(c)$ of a child c of u.

The last thing missing is now the computation of $\text{child}_{\text{cost}}(u)$ in Algorithm 3.3 and the DFS. If we do not start a DFS Algorithm 3.3 only does constant work. We consume the token for this by adding $\text{cost}(\{u, v_m\})$ to $\text{child}_{\text{cost}}(u)$. In this case, $\text{cost}(\{u, v_m\})$ is negative and $\text{cost}(\{u, v_m\}) \leq -1$. For the DFS, we need constant work per node. We visit a node at most twice and can attribute the work for the second visit already to the first visit of a node. When we visit a node x there are a few different situations: the node has not been processed or $\text{child}_{\text{cost}}(x) < 0$, we consume a token by decreasing $\text{child}_{\text{cost}}(u)$ by at least 1. In the second case, we visit a node x that has been processed and $\text{child}_{\text{cost}} \geq 0$. For this, we account the constant work to the processing of the node x, and we skip the node with the DFS pointer. We skip x only once. As we finish processing u either $\text{child}_{\text{cost}}(u) > 0$, and we do not descend into the DFS again, or we aborted the DFS and set $\text{DFS}_{\text{next}}(u)$ to a node below x. We will thus never visit x again.

This means that we process only $\mathcal{O}(\eta d)$ nodes in Algorithm 3.4 and have amortized constant work per node.

We now show that changing the approach, how we process the nodes in a bottom-up approach can further reduce the running time. The authors of [BHHW21] argue for uniform edit costs, that we can ignore deep neighbors at a depth greater $2 \cdot deg(u)$ as otherwise, making u a root is better than choosing a deep neighbor as parent. We therefore can use a bucket per depth of the remaining neighbors in the priority queue, which removes the log factor in the running time. As we already argued for updating the depth values, this is not the case for nonuniform edit costs. Depending on the edit costs, deep neighbors are still potential parents. Instead we can consider a different approach that does not change the asymptotic running time with uniform edit costs, but changes it for nonuniform edit costs. We use an initial array for the neighbors and sort this array by tree depth of the nodes. We can do this in $\mathcal{O}(d\log(d))$ per node. After this, we keep two dynamic arrays of nodes for the current and next level of the tree. We process nodes in the current level as long as it is not empty. The parent of a processed node is pushed into the array for the next level. As soon as the current level runs empty, we additionally process the neighbors on the current level and then process the array for the next level. We can do this by simply swapping the arrays, as the current level is now empty. In addition, we skip empty levels and only process neighbors of this level if there are any. With this dynamic array approach, we have constant running time for inserting a node in the array and processing it. This way, the running time for the algorithm is $\mathcal{O}(\eta d + d\log(d)$ per node as we still process ηd nodes. Per iteration, we have a running time of $\mathcal{O}(m\eta + n\Delta\log(\Delta))$ as we sort, in the worst case, a array of Δ nodes.

Based on the previous assumptions, we can show the worst-case running time for QTM is $\mathcal{O}(n\log(d))$ per node, with n as the number of nodes in the graph. This means the worst-case running time per iteration is $\mathcal{O}(n^2\log(\Delta))$ when using a priority queue for the nodes. As we update the depth of the descendants of a node u, at the new position, we have to update at most n nodes. We do this once and at the new position, u can have a maximum of n descendants. As updating the depth values takes constant time, QTM has a running time of $\mathcal{O}(n)$ for updating the depth values at the new position. We also do not need an amortized analysis for the original position. We have the same situation as for the new position. A node can have a maximum of n descendants, and we can update the depth values in $\mathcal{O}(n)$ per node and $\mathcal{O}(n^2)$ per iteration, as we move a node at most once in one iteration. We can compute the current edit costs and update pointers for children and parents in the same running time. Missing from our proof is now that QTM can find the best parent and close children in $\mathcal{O}(n\log(d))$. We show that we process at most n nodes and initially add d nodes to the queue. For the algorithm, we first add all neighbors of v_m to the priority queue. As we only add one node to the queue for every node we process, there are never more than d nodes in the queue at the same time. We also process nodes ordered by depth, and thus never insert a node into the queue that was already processed. As shown before, we process each node at most once in the priority queue and visit the node at most twice in the DFS and skip the node otherwise with the DFS pointers. We do constant work in the DFS per node and when processing a node. Therefore, we have a worst-case running time of $\mathcal{O}(n\log(d))$ per node and $\mathcal{O}(n^2\log(\Delta))$ per iteration while using a priority queue. We can again lower this running time with the dynamic array approach to $\mathcal{O}(n + d\log(d))$ per node and $\mathcal{O}(n^2 + n\Delta\log(\Delta))$ per iteration. This running time is polynomial, while the earlier running time depends on the edit costs of the graph and is only pseudo-polynomial.

4. Subtree Move

QTM minimizes the edits and edit costs for the quasi-threshold editing with local moving of a single node. The algorithm finds the best parent for the node and adopts its close children. Under these circumstances, QTM finds an optimal solution for every local move. Here, we show that extending the local move to allow moving more than one node can reach different solutions that are not possible with local move. We introduce an optimization on QTM that allows moving of nodes with their subtree. During this algorithm, we again minimize the edits and edit costs needed for this move. The subtree move reuses a lot of components of the basic QTM algorithm which allows it to reach a good asymptotic running time, however, it is still worse than a simple local move.

We introduce the basic idea in Section 4.1 and more details on the algorithm in Section 4.3. Furthermore, we add a simple path sort step based on QTMs simple path sorting in Section 4.4. In Section 4.5, we introduce different options for the subtree move, for example, only moving one child together with v_m. We also prove the correctness of the algorithm in Section 4.6 and its running time in Section 4.7.

4.1 Basic Subtree Move Algorithm

The basic idea of subtree move is instead of moving a single node, we move its whole subtree, including all nodes below it. For QTM, we combine both local move and subtree move. We first move the node v_m to its optimal position with a local move, and then adopt all close children of its parent. Then we look at the subtree of the node v_m. When the subtree of v_m does not only consist of a single node, we try to move v_m and its subtree to a new position. We find the optimal position for the subtree, which can differ from only moving v_m alone. At the new position, we find close and indifferent children for v_m and adopt all close children in addition to its subtree. We randomize the choice for equivalent best parents in relation to edit cost, and can also sample randomly from indifferent children. We can also adopt them, as they do not change the edit costs. In comparison to the local move, we do not isolate the node first because the children of v_m will move with it.

Subtree move is close to implicitly contracting the subtree below v_m onto v_m and finding a new parent for the subtree of v_m. The only time it differs from contracting is when we adopt children. Instead of moving them below the subtree, we move them directly below v_m. This way, we can again use child closeness or child cost values to decide if we should adopt a child. We can reuse the computation from the local move.

Algorithm 4.1: The algorithm for calculating the size and sum of degree for the subtree of v_m.

```
1  $m \leftarrow$ number of edges in graph
2  $n \leftarrow$ number of nodes in graph
3  $\text{size}_{\text{subtree}} \leftarrow 0$
4  $\text{degree}_{\text{subtree}} \leftarrow 0$
5  $\text{degree}_{\text{avg}} \leftarrow (2 \cdot m)/n$
6  $x \leftarrow$ first child of $v_m$
7  while $x \neq v_m$ do
8      if $x$ not processed then
9          mark $x$ as processed
10         mark $x$ as as part of subtree
11         $\text{size}_{\text{subtree}} \leftarrow \text{size}_{\text{subtree}} + 1$
12         $\text{degree}_{\text{subtree}} \leftarrow \text{degree}_{\text{subtree}} + \text{degree}(x)$
13         if $\text{size}_{\text{subtree}} > max\{50, \sqrt{n}\}$ or $\text{degree}_{\text{subtree}} > \text{degree}_{\text{avg}} \cdot max\{50, \sqrt{n}\}$
            then
14             break
15         $x \leftarrow$ next node in DFS order after $x$ below $v_m$
16     else
17         unmark $x$ as processed
18         $x \leftarrow$ next node in DFS order after the subtree of $x$ below $v_m$
```

4.2 Subtree Move

For the first step of the subtree move algorithm, we start a DFS at v_m through its subtree. We calculate the size of the subtree including v_m and the sum of node degree in the subtree. We want to decide if it makes sense to move v_m with its subtree. Likewise, we only continue if the subtree has at least two nodes, including v_m. Otherwise, we only move v_m and this does not differ from the local move. We also want to limit the maximum size of the subtree we move. It is easy to see that the running time of the subtree move depends on the size of the subtree we move. For example, for a node u we have to sum up the edit cost to every node in the subtree to u to decide if u is a potential parent. We only allow a maximum subtree size of $\sqrt{n}$. We prove in Section 4.7 that this influences the running time. For small graphs, we can allow more nodes because the running time is lower. We execute the subtree move if the subtree is smaller than 50 or $\sqrt{n}$ whichever is higher. We can also make an argument for only allowing subtrees with a low average degree. As we have to visit every neighbor of the subtree in the algorithm, we can reduce the running time by allowing subtrees with a low average degree. For this, we allow a threshold of subtree size times average degree of the graph. This way, we ignore subtrees that have a large number of nodes and a high average degree.

The algorithm for the subtree size is shown in Algorithm 4.1. We initialize the average degree as two times the number of edges divided by the number of nodes. We then start a DFS from v_m. For every node we visit for the first time, we increase the subtree size by one and add the degree of the node to the subtree degree. If the subtree size or subtree degree are above our threshold for the subtree size or size threshold times average degree, respectively, we abort the DFS early and do not execute the subtree move. Otherwise, we continue and have the subtree size at the end of the DFS.

As a next step for the subtree move, we have to calculate the edits and edit costs we currently need for the subtree. This way, we can compute the relative saved edits and edit

Algorithm 4.2: The algorithm for calculating the external degree and external edit costs for the subtree of v_m.

```
1  $\text{cost}_{\text{neighbors}} \leftarrow 0$
2  $\text{degree}_{\text{extern}} \leftarrow 0$
3  $u \leftarrow$ first child of $v_m$
4  while $x \neq v_m$ do
5      if $x$ not processed then
6          for $v \in N(u)$ do
7              if $v$ not in subtree then
8                  $\text{num}_{\text{neighbors}}(v) \leftarrow \text{num}_{\text{neighbors}}(v) + 1$
9                  $\text{cost}_{\text{neighbors}} \leftarrow \text{cost}_{\text{neighbors}} + \text{cost}(\{v, x\})$
10                 $\text{degree}_{\text{extern}} \leftarrow \text{degree}_{\text{extern}} + 1$
11         mark $x$ as processed
12         $x \leftarrow$ next node in DFS order after $x$ below $_v m$
13     else
14         unmark $x$ as processed
15         $u \leftarrow$ next node in DFS order after the subtree of $x$ below $v_m$
```

costs for the new best parent. In comparison to the basic QTM, it is not enough to get all edits adjacent to v_m. We instead need all edits and edit costs adjacent to all nodes in the subtree. We also can focus on the edits and edit costs for ancestors of v_m. The node v_m is moved with its descendants, and thus no edits will change here.

Algorithm 4.2 shows the preparation for the current edits and edit costs of the subtree, which is closely related to the first step of the basic QTM where we remove all edges of v_m. For the subtree move, we remove all external edges of the subtree. This means we remove all edges of nodes in the subtree, excluding edges that connect two nodes in the subtree. For uniform edit costs, we start a DFS at v_m. When we first visit a node u we loop through its neighbors in G. For every neighbor of u that is not in the subtree, we increase the external degree of the subtree by one. We also keep for every node in G how many neighbors it has in the subtree, which we will need later in the algorithm. We increase the value of every neighbor in the loop by one. At the end of the DFS, we have summed up the external degree of the subtree and for every node in G we know how many neighbors it has in the subtree. For nonuniform edit costs, we can use the same DFS as for uniform edit costs. When we first visit a node u, we also loop through all neighbors of u. For every neighbor v not in the subtree, we have to sum up the edit costs of u to the neighbor v. After the DFS, $\text{cost}_{\text{neighbors}}$ is the sum of edit costs for all external edges of the subtree, which we will need for calculating the current edit costs of the subtree. The next step of the algorithm differs a lot between uniform and nonuniform edit costs and is the reason we can not reach similar running times for the two different cases for this optimization. While we can use the number of neighbors in the subtree for every node for calculating edits in the rest of the algorithm, we can not do the same for nonuniform edit costs. We have to add up all edit costs for every node in the subtree to a specific node. This step alone has a running time of $\mathcal{O}(n\sqrt{n})$ with $\sqrt{n}$ as the subtree size. It is the main reason we limit the maximum size of the subtree.

Algorithm 4.3 shows the calculation of the subtree edit costs. We loop through all nodes of the graph G. For every node v in the graph, we add up all edit costs to nodes in the subtree, including v_m. We will use this value to find the best parent for the subtree and will call it $\text{cost}_{\text{subtree}}(v)$ for a node v. This step offers potential for improvement. While it

Algorithm 4.3: The algorithm for calculating the $\text{cost}_{\text{subtree}}$ for every node in the graph.

```
1 for $v \in V$ do
2     for node $u$ in subtree do
3         $\text{cost}_{\text{subtree}}(v) \leftarrow \text{cost}_{\text{subtree}}(v) + \text{cost}(\{u, v\})$
```

Algorithm 4.4: The algorithm for calculating current edit costs for the subtree of v_m so we are able to calculate saved edit costs for the subtree move.

```
1 $\text{edits}_{\text{subtree,current}} \leftarrow \text{degree}_{\text{extern}}$
2 $\text{cost}_{\text{subtree,current}} \leftarrow \text{cost}_{\text{neighbors}}$
3 for ancestor $p$ of $v_m$ do
4     $\text{edits}_{\text{subtree,current}} \leftarrow \text{edits}_{\text{subtree,current}} + \text{size}_{\text{subtree}} - 2 \cdot \text{num}_{\text{neighbors}}(p)$
5     $\text{cost}_{\text{subtree,current}} \leftarrow \text{cost}_{\text{subtree,current}} - \text{cost}_{\text{subtree}}(p)$
```

might be useful to parallelize this step in practice, we noticed that the graphs we evaluated on were too small. In preliminary experiments, we see that for graphs up to 8800 nodes, a vectorized operation to add up the edit costs is faster than a parallel approach.

With the preparation out of the way, we can now determine the current subtree edit costs. Algorithm 4.4 again shows the algorithm for both edits and edit costs. For uniform edit costs, we initialize the current subtree edits with the external degree of the subtree. As mentioned earlier, this is equivalent to removing all edges from the subtree to other nodes in the graph. We then have to consider all ancestors of v_m. For every ancestor p, we check how many neighbors p has in the subtree. We subtract the number of neighbors we calculated in the preparation from the current subtree edits. This way, we revert removing external edges of the subtree that we still need. The number of non-neighbors of p in the subtree is the difference between subtree size and the number of neighbors of p in the subtree. We add this difference to the current subtree edits. For every non-neighbor, we have to insert an edge and thus increase the current subtree edits by one. After looping through all ancestors, we have considered all current subtree edits. It is not necessary to start a DFS because v_m will keep its subtree and therefore the edits to its descendants do not matter as they do not change.

We initialize the current subtree costs for nonuniform edit costs with the cost of all neighbors of the subtree. This is the sum of edit costs for all external edges of the subtree, which again implies removing all these edges. We already calculated this value in the preparation. To get the sum of current subtree costs, we also have to go through all ancestors of v_m and can ignore the edit costs for descendants because v_m will keep them. For every ancestor p, we subtract the edit cost of p to the subtree from the current subtree cost. We already calculated this value in the preparation, so we do not have to loop through all nodes in the subtree and add up the edit costs. The definition of edit costs saves us again an adjacency check for the nodes. The subtraction reverses removed edges and adds the edit costs for inserted edges. After the computation, we now have the edits and edit costs that are currently needed for the subtree, which can change if we move the subtree. The next step is now to find close children and the best parent for the whole subtree.

4.3 Best Parent

Like the basic QTM algorithm and its local move, we have to find the optimal parent for the subtree move. We also allow adopting children of the new parent. The decision on the best parent depends, therefore, on the one hand on the subtree costs of the parent and on the other hand on the edits and edit costs we can save by adopting close children. Subtree move uses child closeness and costs to make the adoption decision for close children of the new parent. This calculation does not differ from the local move and will be executed as described in Section 3.5. We simply have to make sure that v_m and the nodes in the subtree are never processed because we do not isolate it compared to the local move. It is enough to make sure they are not inserted in the priority queue, and the DFS for the child closeness and child cost is aborted when we reach v_m. Because of the bottom-up approach, this way, we ignore the subtree for the adoption and best parent decision.

For the subtree move, we have to find the best parent for the subtree of v_m. We can again find the best parent recursively. As described in Section 3.6, it is possible to find the best parent for a subtree T_u constrained to a node u. We propagate these values up until we have the best parent for the whole forest in the virtual root. Here again, helps the view of contracting the nodes in the subtree of v_m onto v_m. Instead of finding the best parent based on neighbors and non-neighbors of v_m we use neighbors and non-neighbors of the subtree of v_m. For nonuniform edit costs, we use subtree edit costs, which we calculated in the preparation step.

Similar to the local move, we can use the same bottom-up approach for finding close children of v_m and finding the best parent. We thus can use the same priority queue for both computations. We again will describe the algorithm with uniform edit costs first and then go into detail where edit costs have to be considered. In comparison to the local move we consider, for every node we process, the edits to every node in the subtree. The edits can also be zero. This is the case when there are as many u-neighbors as non-u-neighbors in the subtree. We computed these values in preparation for the algorithm, and can calculate the edits for the subtree from the number of neighbors of a node in the subtree and the subtree size.

Algorithm 4.5 shows how we fill the queue for uniform edit costs. We loop through all subtree neighbors. A potential parent u needs at least one close child or at least one neighbor in the subtree. Because of the bottom-up approach, we initially only need to add subtree neighbors to the priority queue to process every potential parent. We add a node to the priority queue if it is a neighbor of v_m. This is necessary for determining child closeness values that do not depend on the subtree. We also add nodes to the queue that have at least as many neighbors as non-neighbors in the subtree. We already have prepared these values. The number of non-neighbors for a node u is the difference between the subtree size and $\text{num}_{\text{neighbors}}(u)$. We can check the number of edits for the subtree of a node u in constant time. The same loop is shown in Algorithm 4.6 for nonuniform edit costs. We again loop through all subtree neighbors and add neighbors of v_m to the queue because of the already mentioned reasons. Additionally, we add nodes to the queue that have a positive subtree edit cost or zero. These nodes have the potential to be parents for the subtree. We again have prepared these values, so they do not have to be determined here.

Algorithm 4.7 shows how we determine the best parent for the subtree. We again combine uniform and nonuniform edit costs and will use max_{score} and max_{cost} to find the best parent. As in the local move, we use the tree depth of the nodes as keys for the priority queue to process nodes in a bottom-up approach. For uniform edit costs, we first determine child closeness values for a node u. Then, for each node u, we check if the sum of child closeness over its close children is greater than the maximum score max value of its children. If this is the case, u is the best parent for its subtree or otherwise the best parent is below u. If the two values are equal, we randomize the choice. Independently of that, we add the difference between the number of neighbors and non-neighbors of the node in the subtree to

Algorithm 4.5: The algorithm for finding potential parents for the subtree in the subtree neighbors with uniform edit costs and pushing them in the queue to be processed later.

```
for neighbor u of the subtree do
    if u not in subtree then
        if u is neighbor of $v_m$ or $\mathrm{num}_{\mathrm{neighbors}}(u) \cdot 2 - \mathrm{size}_{\mathrm{subtree}} \geq 0$ then
            push u
```

Algorithm 4.6: The algorithm for finding potential parents for the subtree in the subtree neighbors with nonuniform edit costs and pushing them in the queue to be processed later.

```
for neighbor u of the subtree do
    if u not in subtree then
        if u is neighbor of $v_m$ or $\mathrm{cost}_{\mathrm{subtree}}(u) \geq 0$ then
            push u
```

Algorithm 4.7: The algorithm for calculating $\mathrm{max}_{\mathrm{cost}}$ and $\mathrm{max}_{\mathrm{score}}$ for the subtree move. The algorithm finds the best parent for the subtree of v_m.

```
while queue not empty do
    $u \leftarrow$ pop
    determine $\mathrm{child}_{\mathrm{cost}}(u)$ and $\mathrm{child}_{\mathrm{close}}(u)$ by DFS
    $a \leftarrow$ max over $\mathrm{max}_{\mathrm{score}}$ of reported $u$-children
    $b \leftarrow \sum$ over $\mathrm{child}_{\mathrm{close}}$ of close $u$-children
    $x \leftarrow$ max over $\mathrm{max}_{\mathrm{cost}}$ of reported $u$-children
    $y \leftarrow \sum$ over $\mathrm{child}_{\mathrm{cost}}$ of close $u$-children
    if $x = y$ then
        coin $\leftarrow$ randomization
        if coin then
            $\mathrm{max}_{\mathrm{score}}(u) \leftarrow b$
            $\mathrm{max}_{\mathrm{cost}}(u) \leftarrow y$
        else
            $\mathrm{max}_{\mathrm{score}}(u) \leftarrow a$
            $\mathrm{max}_{\mathrm{cost}}(u) \leftarrow x$
    else
        $\mathrm{max}_{\mathrm{score}}(u) \leftarrow max\{a, b\}$
        $\mathrm{max}_{\mathrm{cost}}(u) \leftarrow max\{a, b\}$
    $\mathrm{max}_{\mathrm{score}}(u) \leftarrow \mathrm{max}_{\mathrm{score}}(u) + \mathrm{num}_{\mathrm{neighbors}}(u) \cdot 2 - \mathrm{size}_{\mathrm{subtree}}$
    $\mathrm{max}_{\mathrm{cost}}(u) \leftarrow \mathrm{max}_{\mathrm{cost}}(u) + \mathrm{cost}_{\mathrm{subtree}}(u)$
    if $\mathrm{child}_{\mathrm{cost}}(u) > 0$ or $\mathrm{max}_{\mathrm{cost}}(u) > 0$ then
        report $u$ to $p(u)$
        push $p(u)$
```

the score max value. If the score max value stays positive or the child closeness is positive, we add the parent of u to the queue.

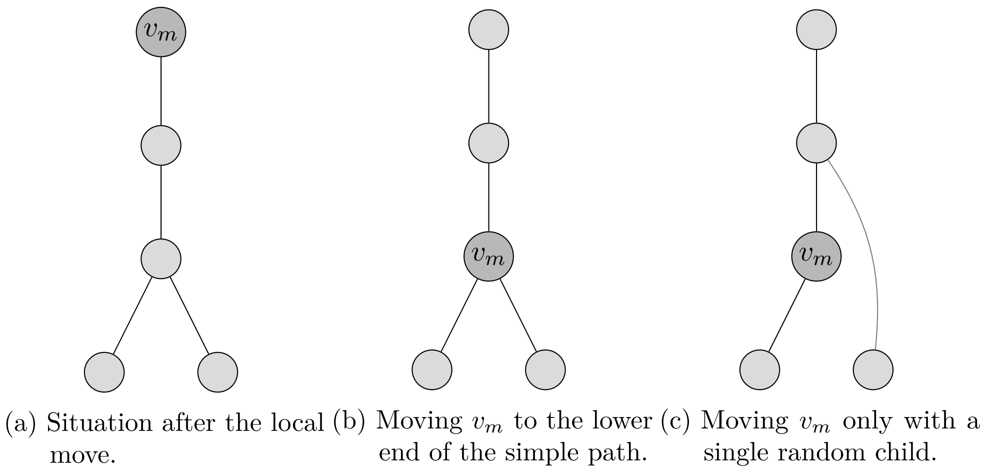
When considering nonuniform edit costs, we first sum up the child costs of close children. If the sum of child costs is equal to the maximum max cost value of its children, we randomize if we choose the u or the best parent below u as the best parent. Otherwise, we choose the maximum of both values. Independently of the choice, we add the subtree edit costs of node u to the max cost value of u. When the child cost or max cost value of u is positive, we add the parent of the node to the queue and report u to its parent. After the queue is empty, we can find the best for the subtree and its associated max score or max cost in the virtual root.

We now can use the child cost values to decide which child v_m additionally should adopt. We adopt all close children of the best parent because they save edits or edit costs in comparison to not adopting them. One special case is children of the best parent with a child cost or closeness value of zero. We can call them indifferent children, because for the edit costs, it does not make a difference if we adopt them or not. The authors of [BHHW21] only allowed adopting of at least two children to avoid equivalent solutions for the local move. We can here simply sample randomly from the indifferent children. Because we only do a subtree move when v_m already has a child, we always have at least two children when we adopt one. This way, all edge cases from [BHHW21] can be ignored, and we take a random sample from the indifferent children of the best parent to adopt. We now can move v_m, with its subtree, to the new position, and adopt all close children of the new parent and the chosen indifferent children. Because the algorithm is building on local move, subtree move also finds the original position again if it was the optimal position for the subtree.

4.4 Subtree Sort Path

The authors of [BHHW21] show that it is possible to change the order of simple paths in a rooted forest without changing its implied quasi-threshold graph. The authors use this property to sort simple paths within QTM and move neighbors of v_m to the top of their respective simple paths. The authors show in their experiments that this can reduce the number of edits that are needed for the local move. In Section 3.4, we examined the step for nonuniform edit costs and have seen that this step does not require additional edit costs, either. This means we can sort simple paths for the local move, and it is also possible for the subtree move.

Simply moving neighbors of v_m to the top of their simple paths is not enough in this case. As we have to consider all nodes in the subtree for the best parent, all neighbors of the subtree can have an influence on $\max_{\text{cost}}$ for a node u. We thus sort all subtree neighbors with a positive subtree edit cost to the top of their simple path. For every neighbor, we switch its position with the first node in the simple path that is not a neighbor of the subtree. We sort a node u to the top when $\text{cost}_{\text{subtree}}(u) \geq 0$. For the local move, it is not too hard to see why this step can save edits or edit costs. Every neighbor in the ancestors of a potential parent saves edit costs. For the subtree move, we can have the situation for a node u where $\text{cost}_{\text{subtree}}(u) = 0$. We might move nodes to the top that do not have an influence on the edit costs. Compared to the local move, we also can not prove that subtree move finds the minimum edit costs incident to the subtree. We can even give a basic idea why this is not the case. The minimum edit costs incident to the subtree, for example, can expect children to be attached to a different location in the subtree than to v_m. We do not allow this operation for our version of QTM. The expectation is that the sorting step for subtree move can save edit costs in certain cases, but does not make



(a) Situation after the local move. (b) Moving v_m to the lower end of the simple path. (c) Moving v_m only with a single random child.

Figure 4.1: Examples for the different subtree options.

a difference for others. We evaluate subtree move simple path sorting as an additional variant of QTM in Chapter 5.

4.5 Subtree Move Option

When we move more than one node together with v_m, moving the node together with its whole subtree is the most trivial option and is the option we used to introduce the subtree move optimization. Considering we execute the subtree move after the local move, we already know that all adopted children of v_m are close children or, in case of randomization, at least, indifferent children. They have a positive child closeness or child cost, depending on edit costs, and it makes sense to move them together. Looking at the way the local move is executed, we notice a detail that might be a problem for the subtree move. As described in Section 3.6 local move moves the node as a default option to the top of its new simple path. If v_m is not alone in its simple path, v_m therefore only has one child. Other nodes in its simple path also do not have to be its close children.

Figure 4.1a shows an example situation with v_m. The local move placed v_m at the top of its new simple path. The close children that v_m adopted are below the lowest node in the simple path. Our subtree move, as it was described before, will now try to move v_m together with its simple path and all descendants of it. This specific situation does not allow a different subtree move without changing v_m.

Fortunately, as proven by the authors of [BHHW21], we are able to reorder the nodes in the simple path of v_m without changing the implied quasi-threshold graph. We can use this to allow more options when moving the subtree of v_m.

The first option we have is moving v_m and its subtree without reordering the simple path. We simply take the situation after the local move and start our algorithm. This is shown in Figure 4.1a. No further operations are necessary to use this option for the subtree move.

The second option we have is moving v_m to the lower end of its simple path before starting a subtree move. We switch the position of v_m with the current lower end of the path. The situation after moving v_m to the lower end is shown in Figure 4.1b. This has more than one advantage. First, we can say that all children of v_m are now close or indifferent children of v_m. Given that local move does not allow the adoption of a single child, local move will not move v_m to a position in a simple path that is not the lower end and the child below v_m is one of the adopted children. Thus, the adopted children are always the children of the lower end of the path. We can now move v_m together with all of its adopted children.

Secondly, we have now the option to not move the whole subtree, but instead move a subset of the adopted children together with v_m. While it was still possible beforehand, moving only a part of the subtree is not the same as reordering a simple path and therefore implies edits and edit costs. We have to correct these if we want to move anything else than the whole subtree.

The third option we now have, as already mentioned, is not moving the whole subtree. The idea is here to move v_m with only a subset of its children, which is only possible if v_m is the lower end of its simple path. Every child that is not moved gets the parent of v_m as its new parent. One example of this is given in Figure 4.1c, where we only move v_m together with one child and the second child gets a new parent. While it would be possible to find the optimal subset of children to move, this is no trivial task. We thus decided on randomly sampling one child to move together with v_m. As already mentioned, in this case, we have to correct edits and edit costs for every child we do not move together with the subtree. Given that every child here was processed during the local move, we can simply subtract the child closeness and child cost values from the current subtree edits and edit costs.

In the algorithm for the subtree move, we start with the first option and try to move the subtree. In case we find a best parent, we move the subtree to the new position. Otherwise, we continue with the next option and try to move the subtree again. We go through all options for the subtree move in the order we listed them, and resume with the local move of the next node if we executed one subtree move or tried every option.

4.6 Correctness

In this section, we show that the subtree move algorithm correctly finds the best parent for the subtree of v_m in regard to edit costs, while considering the edit costs we will save by adopting children of the best parent. We start by showing that the $\text{child}_{\text{cost}}$ values are calculated correctly. The subtree move algorithm borrows the $\text{child}_{\text{cost}}$ calculation from the local move and uses these values to decide on children for v_m to adopt. We already have shown in Section 3.8 that the $\text{child}_{\text{cost}}$ values are correctly calculated by the algorithm. As the children are directly adopted by v_m instead of the subtree, we have the same situation as for the local move and the proofs from Section 3.8 for $\text{child}_{\text{cost}}$ still hold true

We now have to show that the algorithm finds the best parent for the subtree. The proof is close to the proof for the local move in Section 3.8. The main difference is that for a node u we have to consider $cost_{subtree}(u)$ instead of $cost(\{u, v_m\})$. First, we show that we do not have to add all nodes to the queue and consider as parent for the subtree:

Theorem 4.1. *Only nodes with close children and nodes with $cost_{subtree} \geq 0$ need to be considered as parents of v_m and its subtree.*

Proof. We can prove this theorem analog to Theorem 3.2 for the potential parent for the local move. We simply adapt the proof by considering nodes with $cost_{subtree} \geq 0$ as neighbors of the subtree. The same is true for neighbors of v_m and $\text{cost}(\{u, v_m\}) \geq 0$. □

We have shown that the edit costs below a node u is equivalent to the local move. Now, we have to consider the edit costs above a potential parent u:

Theorem 4.2. *Consider the subtree T_u of a node u. Then for the subgraph of T_u, either $\max_{\text{cost}}(u) < 0$ or u has been processed and $\max_{\text{cost}}(u)$ is correctly computed by Algorithm 4.7.*

Proof. We use $\max_{\text{cost}}(u)$ again to find the best parent for the subtree move. Algorithm 4.7 uses the same structure as Algorithm 3.4 simply with $\text{cost}_{\text{subtree}}(u)$ instead of $cost(\{u, v_m\})$. The proof can be adapted analogously. We can simply consider $\text{cost}_{\text{subtree}}(u)$ as edit costs for every node u in the proof in Section 3.8 and correctly consider all nodes in the subtree. We do not repeat the proof here. Subtree move therefore finds the best parent to move v_m and its subtree to. □

We have shown that subtree move finds the optimal parent for the subtree of v_m and children for v_m to adopt. Furthermore, we have shown that the concept of the local move for a single node can be adapted without too many changes to work for a subtree. We can reuse most components of the algorithm.

4.7 Running Time

We have already shown that the subtree move algorithm correctly finds the parent for v_m and its subtree. We also have shown that we can use the same basic idea of adopting children as for the local move. The running time of the subtree move optimization is $\mathcal{O}(n\theta + n\log(|N(S_u)|))$ per node. S_u is the set of nodes in the subtree T_u. Subtree move also increases the running time per iteration to $\mathcal{O}((n^2\theta + n^2\log(|N(S)|))$. S is the set of nodes in the subtree with the largest neighborhood we move. For this proof, we use θ as the number of nodes in the largest subtree we move:

$$\theta = \max_{u \in V}(|V_{T_u}|)$$

With the same idea as in Section 3.9 we can further reduce the running time by using dynamic arrays instead of the priority queue when processing the nodes. This way we reach a running time of $\mathcal{O}(n\theta + |N(S_u)|\log(|N(S_u)|))$ per node and $\mathcal{O}(n^2\theta + n|N(S)|\log(|N(S)|))$ per iteration.

We now have to show that QTM can find the best parent for the subtree and children to adopt in the mentioned running time.

As $\theta = n$ in the worst case, we have a quadratic running time per node. This is the reason we limit subtree size in the algorithm to $\sqrt{n}$. So for the following, we assume $\theta = \sqrt{n}$. We calculate $\text{cost}_{\text{subtree}}$ when we execute a subtree move in Algorithm 4.3. This preparation is in running time $\mathcal{O}(n\theta)$ per node and in $\mathcal{O}(n^2\theta)$ per iteration. We simply loop through all nodes in the graph and add up edit costs for all nodes in the subtree. This preparation allows us to do constant work for every node we process and when we start a DFS for a node. This computation is also fast in practice, as it is a linear and thus cache-efficient scan over the edit costs values of all nodes. It allows us to use the basic idea from the running time proof of the basic QTM algorithm in Section 3.9. An amortized analysis is made difficult by the fact that $\text{cost}_{\text{subtree}}$ can be zero, and we thus do not always consume a token when we process a node. We instead limit us to the worst-case running time.

As for the local move, we have to update depth values at the new position. We can only have n descendants at the new position, not depending on any factors. We update at most depth values for n nodes and can do this in constant time per node. Therefore, we can update depth values in $\mathcal{O}(n)$ per node. The same is true for the original position. We update at most n nodes as descendants of the node we move. We can update all depth values in $\mathcal{O}(n)$ per node and $\mathcal{O}(n^2)$ per iteration. Because of the preparation, we also can calculate the current edit costs for the subtree in the same running time. We visit each node at most once for the computation. The same is true for updating pointers for children and parents when we move a node.
We now have to prove that we can find the best parent for the subtree and children to

adopt in $\mathcal{O}(n\log(|N(S_u)|))$ per node. We process a maximum of n nodes and initially add all subtree neighbors $|N(S_u)|$ to the queue. Likewise, we only add one node to the queue for every node we take from the queue. Thus, there are never more nodes in the queue than there are initially. We also process every node in the queue only once and visit a node at most twice in the DFS and skip it otherwise, as shown in Section 3.9. We need constant time per node for processing it and visiting it in the DFS. In the worst case, we process every node in the graph. So we can find the best parent in $\mathcal{O}(n\log(|N(S_u)|))$ per node. When considering the worst-case running time for QTM, we have a running time of $\mathcal{O}(n\theta + n\log(|N(S_u)|))$ per node and $\mathcal{O}(n^2\theta + n^2\log(|N(S)|))$ per iteration.

As for the local move, we can use the dynamic array approach to process the nodes for the subtree move. We initially insert a maximum of $|N(S_u)|$ nodes into the array and thus can sort the array in $\mathcal{O}(|N(S_u)|\log(|N(S_u)|))$ per node and $\mathcal{O}(n|N(S)|\log(|N(S)|))$ per iteration. The running time for updating the depth values stays the same, but is in this case dominated in the $\mathcal{O}$-notation by the $\text{cost}_{\text{subtree}}$ computation. The running time for the algorithm using dynamic arrays is then $\mathcal{O}(n\theta + |N(S_u)|\log(|N(S_u)|))$ per node and $\mathcal{O}(n^2\theta + n|N(S)|\log(|N(S)|))$ per iteration.

We have proven the asymptotic running time for nonuniform edit costs. Calculating $\text{cost}_{\text{subtree}}$ is one part of the algorithm that differs from uniform costs. For uniform edit costs, we can save the computation of $\text{cost}_{\text{subtree}}$. We still have to loop all neighbors of the nodes in the subtree, and can do this in $\mathcal{O}(|N(S_u)|)$. We also have to do this for nonuniform edit costs, though there it is overshadowed by the $\text{cost}_{\text{subtree}}$ computation. While the subtree move with uniform edit costs is faster than nonuniform edit costs, we still have an increase to the running time compared to the local move. We are not linear in the number of edges per iteration anymore.

5. Evaluation

In this chapter, we evaluate the Quasi-Threshold Mover algorithm with nonuniform edit costs and the subtree move optimization on it. Additionally, we evaluate subtree move on different sets of graphs while using uniform edit costs or different edit costs for removing and inserting edges. The algorithm was implemented in the programming language C++ within the framework NetworKit [SSM16], and based on the Quasi-Threshold Mover by [BHSW15]. We used the Python interface for the experiments. All experiments were executed on a Supermicro Superserver SYS-5018R-MR with an Intel Xeon E5-1630v3 CPU and 128GB(8x16GB) DDR4 RAM.

In this chapter, we first describe the graph instances we used in Section 5.1. Then we look at the solution quality in Section 5.2, especially for nonuniform edit costs, and compare these against exact solutions where known in Section 5.4. We also evaluate the solution quality with uniform edit costs on a bigger set of graphs. In Section 5.3, we evaluate the running time of the algorithm with nonuniform edit costs. Additionally, we examine the effects of the subtree move optimization on running time for nonuniform as well as nonuniform edit costs. In Section 5.5, we examine convergence of the algorithm with nonuniform edit costs compared to uniform edit cost and the subtree move optimization. As a last section, we examine the difference in solutions in Section 5.6 on one graph of the Facebook graph set.

5.1 Graph Instances

We apply the Quasi-Threshold Mover to a few different graph instances with varying sizes and properties. Given that we want to examine the algorithm with nonuniform edit costs, the choice of graphs is limited. A simple weighted graph is not enough because we need edit costs for all node pairs. For nonuniform edit costs, we therefore examine protein similarity data obtained from a COG data set, and introduced in [RWB^{+}07, BBBT08]. The authors generated a graph from protein sequences where the edit costs of two nodes are the difference between a threshold of -10 the proteins E-value from bidirectional BLAST hits. Details can be read in the mentioned paper. We additionally scale the edit costs with a factor of 100 and round up the absolute value. This way, we can use integer values for the edit costs in the algorithm. The resulting graph consists of 3964 connected components with different sizes, starting at below 10 nodes and up to 8836 nodes for the largest component. We view every connected component as a separate graph. The set of graphs has edit costs for every node pair included. The same set was also used to evaluate an exact solver for weighted quasi-threshold editing by [Spi19] and thus we have exact solutions for these graphs to compare our results to. We can use the same set for uniform edit costs. The authors of [GHS+20] introduce an exact solver for quasi-threshold editing. They evaluate the solver on the same graph set, and we then have exact solutions for the graph set with uniform edit costs. We can give an answer on how well the subtree move works for uniform edit costs.

We also use a set of Facebook graphs that were given out by Facebook in the beginning, when Facebook still was mostly used by universities [TMP12]. The set consists of graphs of 100 universities and colleges and is thus also referred to as Facebook 100. The nodes of the graphs represent people from the respective universities. An edge between two people exists if they are connected on Facebook. Additionally, the graphs include information for every person, for example their dorm, or their year of graduation at the university. We can match these properties to the found communities. The set does not include edit costs. Still, we can evaluate uniform edit costs and different edit costs for inserting and removing edges on it. The graph set also includes larger graphs, and we can see how well subtree move scales in practice.

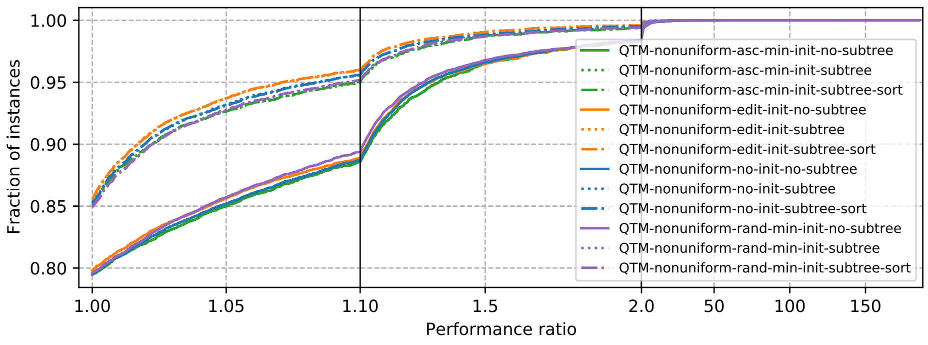
5.2 Solution Quality

In this section, we first evaluate the quality of the solution QTM finds. Our metric is here the sum of edit costs for nonuniform edit costs or the number of edits for uniform edit costs. For the set of protein similarity network graphs, we let the algorithm run with ten different seeds for the randomization and use the given edit costs to find a solution for the editing problem. We run the algorithm for a maximum of 400 iterations and allow a plateau where the move does not change the edit costs of 100 iterations. We also compare different initialization for the algorithm and show that these have an influence on the result. Based on the results of [BHHW21], we do not examine the sort path and randomization optimization. The authors show clearly that these optimizations find better results, and preliminary experiments suggest that this is also the case for nonuniform edit costs. For all experiments, we thus sort simple paths before the local move and randomize the choice for equal best parents and adoption of indifferent children.

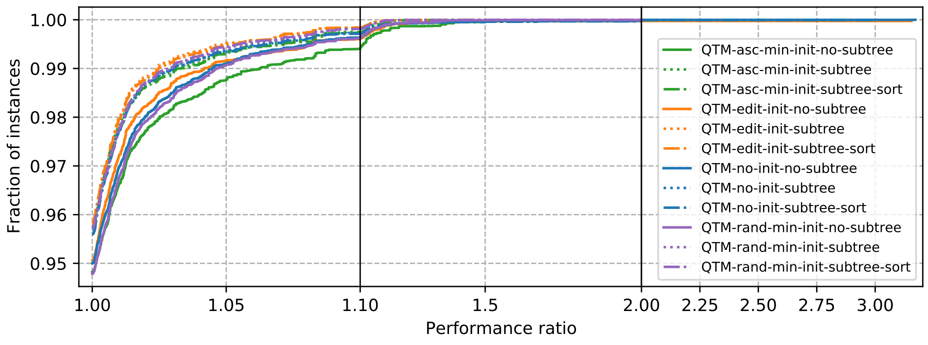
For all plots, we show different variants of the algorithm and compare them. The labels for the different variants of QTM are constructed as follows. **QTM** uses uniform edit costs and **QTM-nonuniform** nonuniform edit costs. Next, we have different intializations: the editing algorithm **-edit-init**, no initialization **-no-init** and the two inserting initializations in randomized order **-rand-min-init** and ordered ascending by node degree **-asc-min-init**. We denote the subtree move optimization by **-subtree** or **-no-subtree** if not enabled. The simple path sort step for the subtree move is denoted by **-sort** for the subtree move variants.

For evaluating the solution quality, we use performance profiles for the different variants of the algorithm. On the x-axis, we see the performance ratio, which compares the found edit costs or edits for a variant of the algorithm with the minimum edit costs or edits found for a given graph. A performance ratio of 1.1 means that the algorithm variant needs 10% more edit or edit costs than the best variant for this graph. The y-axis shows a cumulative fraction of graph instances that reach a certain performance ratio. A graph instance is here a graph with a given seed. For each graph, we have 10 different seeds and graph instances. We use bucket graphs, where every bucket has linear scaling with different bucket borders to increase readability.

Figure 5.1 shows two performance profiles over all 3964 protein similarity graphs for uniform and nonuniform edit costs. We compare different variants of QTM with each other. Both



(a) Comparison of the different variants of QTM on the COG protein similarity graph set using nonuniform edit costs.



(b) Comparison of the different variants of QTM on the COG protein similarity graph set using uniform edit costs.

Figure 5.1: Performance profiles for the COG protein similarity graph set, comparing different variants of QTM with different initializations or no initialization. We focus on the subtree move optimization and simple path sorting for the subtree move.

profiles share the same scaling for the first two buckets, but differ in the last to include the maximum performance ratio. For nonuniform edit costs in Figure 5.1a, we can see that the variants of QTM that only differ in the initialization are close together. The subtree move matches the best solution on a larger fraction of the graph instances than variants without it. QTM matches the minimum edit costs on about 80% of the graph instances, independent of the initialization without subtree move. Most of these instances do not require a single edit, which is one reason for the high fraction. The different initializations influence the overall solutions QTM finds. Inserting nodes in ascending order by node degree reaches the best solution on a smaller fraction of graph instances than the editing initialization. Without subtree move, randomizing the order needs at most 10% more edit cost on 89% of the graph instances. Other initializations are close to it, but reach this performance ration on fewer graph instances. While most graph instances for all variants are at most two times worse than the best solution, we have one outlier at 187 times worse. This is given through the wide range of edit costs. It is possible for two solutions to differ in one edit, but a multiple of that in edit costs.

Examining the subtree move optimization shows us that it plays a bigger role for the solution quality than the initialization. All variants with the optimization activated perform better than their counterpart. QTM matches the best solution on 85% of the graph instances. For subtree move, the difference between the initializations also increases slightly. While without subtree move, no initialization is above the rest for the whole plot, we can now say that the editing initialization performs the best with subtree move. Editing matches the best solution on a larger fraction of the graph instances and is also at most 10% worse than the best solution on a larger fraction compared to other variants. We can also see that sorting simple paths for the subtree move does not always improve the solution of the subtree move. While sorting simple paths for the subtree move is not clearly better than a simple subtree move, the best solution for a few graphs is only reached with sorting before the subtree move. No other variants of QTM could match these few solutions in our experiments. One note that has to be given is that the set of graphs has a lot of graphs that can be solved with zero edits. As mentioned in Section 3.2 all initializations of QTM excluding no-init are inclusion-minimal. QTM overall is not inclusion-minimal, however, the local move only improves the number of edits, and we thus find solutions with zero edits with these initializations guaranteed. A lot of the instances that were solved identically fall on these graphs.

Figure 5.1b shows performance profiles for QTM using uniform edit costs on the protein similarity graphs. We compare the same variants of QTM as for nonuniform edit costs. The first thing to notice is that QTM has less variance in the solutions, with uniform edit costs compared to nonuniform edit costs. QTM matches the best solution for a graph on at least 95% of the graph instances for all variants. We also do not have any big outliers. The worst ratio between the best solution and the solution for a graph instance of a QTM variant is 3.21. As we show in Section 5.4, the best solution of QTM is also the exact solution for most graphs. QTM performs better using uniform edit costs than nonuniform edit costs.

While the performance for uniform edit costs was already examined by the authors in [BHHW21], we examine the subtree move optimization. For uniform edit costs, subtree move finds better solutions on a larger fraction of graph instances than QTM without it. The difference is not as big as for nonuniform edit costs, but it still plays a bigger role for the solution quality than the initialization. The worst initialization with subtree move performs better than the best initialization without subtree move. The initializations still differ with subtree move, but the difference between them is smaller. The editing initialization finds good solutions on a slightly larger fraction of graph instances than the other initializations. For sorting simple paths for the subtree move, the same thing as for

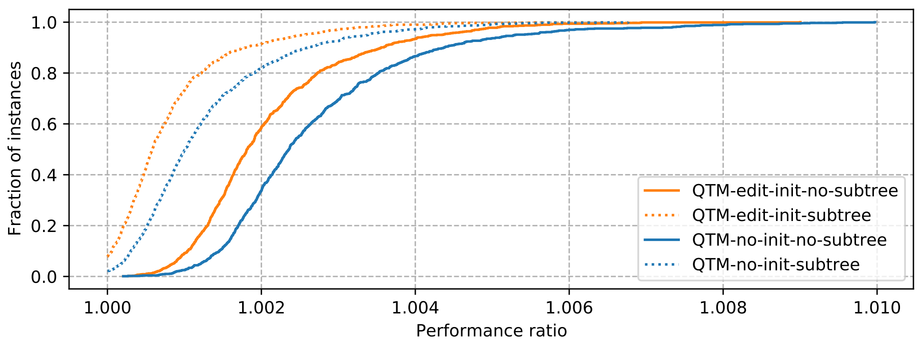


Figure 5.2: Performance profile for the Facebook 100 graph set comparing different variants of QTM with different initializations or no initialization. We focus on the subtree move optimization.

nonuniform edit costs can be said. While it is not clear-cut better to do it, there are a few best solutions in our experiments that were only reached with this variant of the algorithm.

We evaluated QTM with its subtree move optimization on the Facebook 100 graph set, where we used uniform edit costs. We allow, similar to the COG graph set, a maximum of 400 iterations and a maximum plateau size of 100. Furthermore, we limited us to two different initializations to allow the experiments to run in a limited time. These initializations are the editing initialization and no initialization. We also do not use simple path sort for the subtree move. We have seen that it is not clearly better, and preliminary experiments have shown that this is also the case for the Facebook graph set.

Figure 5.2 shows a performance profile for the Facebook 100 graph set, which compares the four variants of QTM we used to each other. We can already see that without subtree move, no graph instance matches the best solution. For most graphs, only a single variant and seed can find the best solution, and for the Facebook graph set, the variant for the best solution always includes subtree move. All graph instances are also, at most, 1% worse than the best solution. This mostly stems from the fact that the graphs need a lot of edits, and QTM comes close to the best solution in very few iterations. The initialization plays a role for the solution quality. As for the COG protein similarity data set, the editing initialization wins and finds better solutions on a larger fraction of graph instances.

Subtree move clearly improves the performance of QTM and plays a bigger role for the solution quality than the initialization. As already mentioned, the best solution for every Facebook instance we found uses subtree move. One fact that should be mentioned here is that most of the Facebook instances use all 400 iterations and do not converge beforehand. As subtree move can move a node more than once per iteration, it can be expected that the solution is better after the same number of iterations for subtree move compared to simple local move. To exclude this factor, we allow different number of iterations for the two variants in Section 5.5. We show that subtree move with 20 iterations achieves similar results to a variant of QTM without subtree move enabled with 400 iterations.

In addition to the performance profiles, Table 5.1 shows the edit costs for the two largest graphs in the COG protein similarity graph set. We can clearly see that subtree move improves the solution QTM finds for these graphs. The same is not true for sorting simple paths for the subtree move, compared to not enabling the sorting step. The mean of the edit costs increases for the second graph, but we have a lower minimum. The standard deviation depends on the graph. While the subtree move decreases the variance for the first graph, it increases the variance for the second graph. The influence of sorting simple

Table 5.1: Comparison of the different variants of QTM regarding edit costs. We focus on the two largest graphs in the COG protein similarity graph set and show n and m for the graphs. We show the minimum and mean of the edit costs over 10 seeds for different variants of QTM on these graphs. Additionally, we show the standard deviation of the edit costs. All variants use the editing initialization.

Number	n	m	QTM Variant	Edit Costs		
				min	mean	std
19	2 362	72 501	nonuniform, no-subtree	2 845 760	2 878 585	24 862
			nonuniform, subtree	2 641 808	2 654 650	8 803
			nonuniform, subtree-sort	2 627 677	2 654 145	12 265
21	8 836	283 778	nonuniform, no-subtree	54 706 928	55 169 078	369 327
			nonuniform, subtree	52 779 871	53 559 498	542 712
			nonuniform, subtree-sort	52 704 149	53 590 736	651 699

paths depends on the graph, while subtree move improves the quality for both graphs, which also can be seen in the performance profiles.

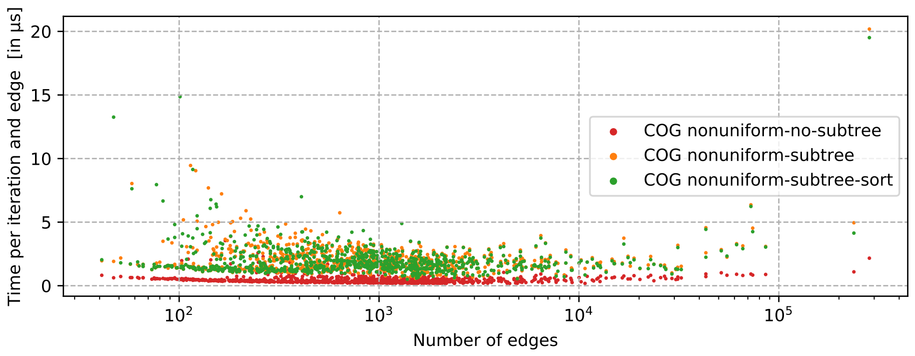
Overall, we can conclude that the solution quality of QTM for nonuniform edit costs is not as good as it is for uniform edit costs. Fewer variants of the algorithm find the best solution for a graph. This can be mitigated through the subtree move optimization, which improves the solution quality of QTM with nonuniform edit costs by a lot. We also have shown that subtree move is a good idea for uniform edit costs. QTM performs better on the COG protein similarity graph set as well as the Facebook 100 graph set when using subtree move. Sorting simple paths for the subtree move on the other side does not always improve the result. However, it can find the best solution on a few graphs where subtree move alone is not able to.

5.3 Running Time

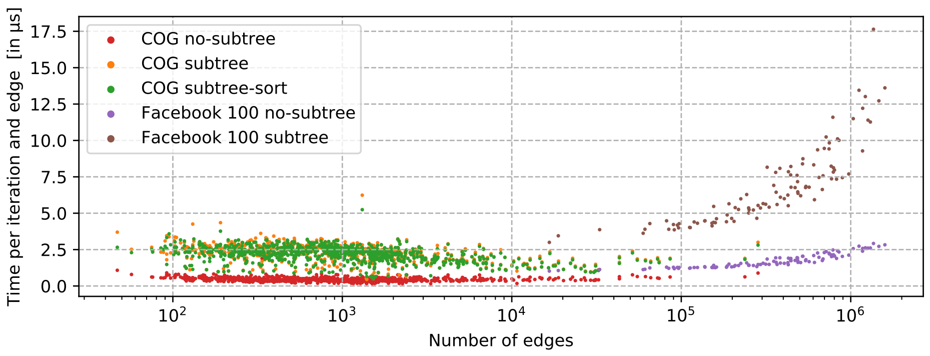
In this section, we evaluate the running time of QTM with nonuniform edit costs in practice. We also evaluate the running time of the subtree move optimization. We do this for nonuniform as well as uniform edit costs. For uniform edit costs, we have access to bigger graphs in the form of the Facebook graphs, which give us a good idea how the optimization scales in practice. We again run the experiments for 400 iterations and average over the different seeds.

We notice big differences in the initializations. Only looking at the initialization, we see that while the initializations that insert nodes with local move are close to the running time of the rest of the iterations, this is not the case for no-init and editing. For very small graphs, editing is a lot slower than an iteration with local move. For one of the smallest graphs in the COG graph set, the initialization is, for example, 1371.58 times slower than the average of the iterations following it. On the other side, no-init is, for example, 17.1 times faster than the local move on the largest graphs of the COG protein similarity graph set. To avoid the influence of the initialization, we exclude the first iteration from the calculation.

For the evaluation of running time, we plot the time per iteration. We also divide through the number of edges because basic QTMs running time is linear in the number of edges per iteration. While this is not the case for nonuniform edit costs and the subtree move, it turns out to be a valid scaling factor in practice, at least for small graphs. The plots show the number of edges on the x-axis. The axis is in logarithmic scale because the number of edges for the different graph sets range from below 100 to over one million edges.



(a) COG protein similarity graph set with nonuniform edit costs.



(b) COG protein similarity and Facebook 100 graph set with uniform edit costs.

Figure 5.3: Running time per edge and iterations plotted versus the number of edges for different variations of QTM on the two examined graph sets.

The y-axis shows the time per iteration and edge in microseconds. We average over all initializations and seeds for a graph and exclude the first iteration that is dominated by the chosen initialization.

Figure 5.3 shows the plots for the time per iteration and edge for uniform and nonuniform edit costs. The uniform edit costs plot includes the Facebook graph set in addition to the COG protein similarity graph set. We plotted different variants of QTM and excluded graphs that took fewer than 20 edits to solve to reduce outliers and increase readability of the plot. For nonuniform edit costs on the COG protein similarity graph set in Figure 5.3a, we can see that basic QTM is linear in the number of edges per iteration. While this does not match the theory, we can see that QTM performs well in practice. We are here limited to the COG protein similarity graph set, which does not include many large graphs. Examining nonuniform edit costs on larger graphs can be a topic of future work.

Subtree move slows QTM down, which is to be expected. The variance in time per iteration and edge increases. This can be explained by the fact that the subtree move is not always executed in every round. We also notice a few outliers where the time per iteration and edge reaches nearly 15 microseconds. This can be explained by, on average, low number of iterations. In the case of the outlier, we see three iterations that are a factor 78 slower than the average and drive up the mean for this graph. We can not give a reasonable explanation for these outliers because they seem very rare and not limited to a single graph, seed or variant. The set does not include very many large graphs, but the ones we do have make a trend upwards visible. Subtree move does not scale well for large graphs. Sorting simple paths does not impact the running time of subtree move by a lot. For most graphs, the difference between sorting and not sorting is not visible.

Table 5.2: Comparison of the running time for different variants of QTM on the two graph sets. We focus on the largest graphs in the graph sets and show n and m of the graphs. We show the minimum and mean of the running time in seconds over 10 seeds for QTM with and without the optimizations enabled. Additionally, we show the standard deviation of the running time. All variants use the editing initialization.

Name	n	m	QTM Variant	Time[s]		
				min	mean	std
Texas84	36 371	1 590 655	uniform, no-subtree	1 777.89	1 791.93	9.66
			uniform, subtree move	8 605.14	8 673.39	30.19
Penn94	41 554	1 362 229	uniform, no-subtree	1 546.23	1 582.53	20.01
			uniform, subtree	9 519.27	9 573.04	39.55
COG Nr. 21	8 836	283 778	uniform, no-subtree	68.19	89.78	11.64
			uniform, subtree	221.42	284.75	43.82
			uniform, subtree-sort	177.80	246.81	56.79
			nonuniform, no-subtree	64.29	68.94	2.12
			nonuniform, subtree	601.85	649.67	37.32
			nonuniform, subtree-sort	595.12	648.90	28.18

Figure 5.3b compares different variants of QTM with uniform edit costs on the two graph sets. Here we have the Facebook graphs to see how subtree move scales on large graphs. Basic QTM was already evaluated by [BHHW21] for uniform edit costs, and we focus here on changes by the subtree move. When only looking at the COG protein similarity graph set, subtree move is close to linear in the number of edges and the running time per edge even decreases for higher number of edges. This observation was also made by the authors of [BHHW21]. We can not give a reasonable explanation why this is the case. Subtree move slows QTM down, but follows the trend of basic QTM closely. The subtree move optimization performs well on small graphs, and the simple path sorting step before the subtree move does not have an influence on the running time.

When looking at the Facebook 100 graph set, the results come closer to the theory. We can also see that the graph set is overall harder for community detection, which comes with increased running times. As expected, subtree move is not linear in the number of edges per iteration for larger graphs. Compared to nonuniform edit costs, we also do not see as many outliers for subtree move. On average, a higher number of iterations lowers the effect a single iteration with a high running time can have on the mean.

In addition to the running time per edge, we compare the absolute running time for the two largest Facebook graphs and the largest bio graph. We want to give more perspective on how subtree move scales in practice. Table 5.2 shows minimum and mean running time for QTM with and without subtree move enabled. All variants of QTM use the editing initialization. For the bio graph, we also include the simple path sorting step and running times for QTM using nonuniform edit costs. The subtree move optimization slows down QTM by a factor of about six for the largest Facebook graph, Penn94. Without subtree move, QTM takes about 26 minutes while subtree move takes two hours and 40 minutes on average. We can also see that the optimization increases the standard deviation of the running time. For the COG protein similarity graph number 21, we can see that using nonuniform edit cost has a lower running time without subtree move than using uniform edit costs. This stems from the fact that QTM using uniform edit costs needs over 350 iterations for this graph. On the other side, using nonuniform edit costs QTM only needs about 110 iterations. We can also see that subtree move is slower with nonuniform edit cost. The mean running time on the bio graph increases by a factor of 9.42. The simple path sorting step speeds up the subtree move for uniform edit cost. Looking at the iterations, this is mostly achieved through reducing the number of used iterations. On running time per iteration, no large influence is visible.

To conclude, we can say that QTM with nonuniform edit costs scales well in practice and can match the asymptotic running time of uniform edit costs. The subtree move optimization performs well for small graphs with a reasonable increase to the running time. Though, we can see that subtree move does not scale well for large graphs. For uniform edit costs on the Facebook 100 graph set, we can see that the optimization takes a lot of time per iteration. As we show in Section 5.5, it is still a valid option by reducing the number of iterations when we are using subtree move. For nonuniform edit costs, the graph sample is too small to give a definitive answer, but the subtree move does not scale well on the two largest graphs of the COG protein similarity graph set.

5.4 Exact Solutions

We picked the COG protein similarity graph not only because they include edit costs, but also because for some of them, there are exact solutions available. For uniform edit costs, the authors of [GHS$^+$20] introduce an exact solver for the problem. The authors compare the basic QTM algorithm with the exact solution for this graph set. We focus on the engineered variant of QTM from [BHHW21] with simple path sorting and randomization and compare our subtree move optimization to it. The author of [Spi19] implements an exact solver for weighted F-free editing with a focus on quasi-threshold editing. The solver is thus considering nonuniform edit costs. For both cases, we have access to the exact result for the COG protein similarity graph set. For uniform edit costs, we additionally have access to a lower bound for the edits needed for the graph set. We compare QTM to the exact results or lower bounds where known. For nonuniform edit costs, graphs exist where we have neither an exact solution nor a lower bound. For these graphs, we simply give statements about the minimum of edit costs QTM found.

When looking at uniform edit costs, we have exact solutions to 3836 of the 3964 graphs. For the remaining 128 graphs, we have a lower bound for the edits. QTM, with all different initializations and optimizations, over 10 seeds for the randomization was able to find the exact solution on all 3836 graphs where the best solution is known. For most of the graphs, more than one variant of QTM could find the exact solution. However, out of these exact solutions, 18 were only found with subtree move enabled and seven of these were only found when using sorting of simple paths for the subtree move.

For the graphs with only a lower bound, we did not match the lower bound with any variant of QTM. We can therefore not say if any of the lower bounds is also the exact solution for the graph. Out of the graphs with only a lower bound for 44 graphs, the minimum was only found by QTM when using subtree move. For 25 of these, the minimum was only found when using simple path sort for the subtree move. However, this does not mean that the minimum for the other graphs was not matched by a variant with subtree move. The opposite is the case. For most graphs, a variant of QTM with subtree move also reached the minimum. The maximum ratio of QTMs solution to the lower bound of a graph is 1.7646. While this is quite far away, we do not have a metric for the quality of the lower bounds. We can not say how good the lower bounds are. QTM comes close to the lower bound on very few graphs, where we know that the lower bound is at least close to the exact solution. For the rest of the graphs, we can not make a statement about the quality of the lower bounds or the solution of QTM.

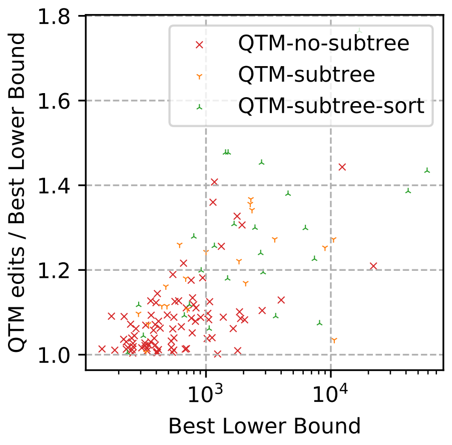
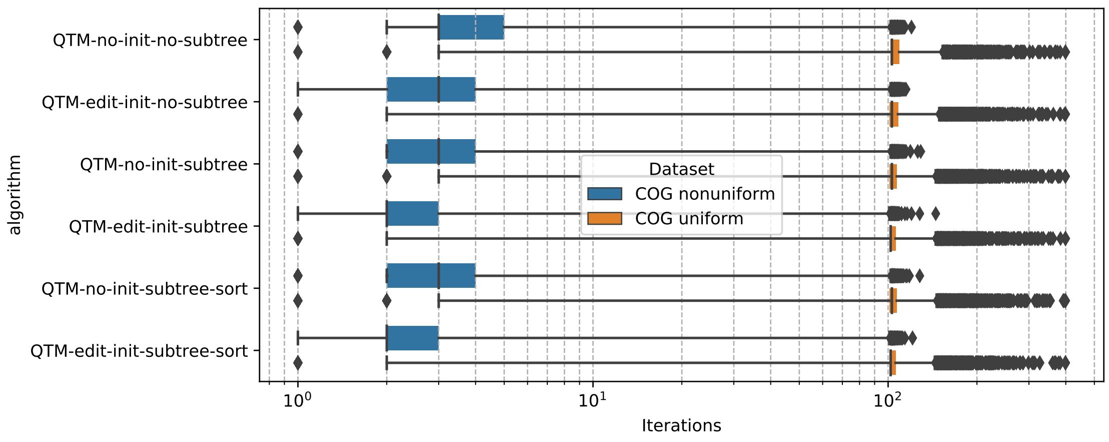


Figure 5.4: Plot for the graphs of the COG protein similarity graph set, which were not exactly solved. Comparison of the solution of the different variants of QTM to the lower bound.

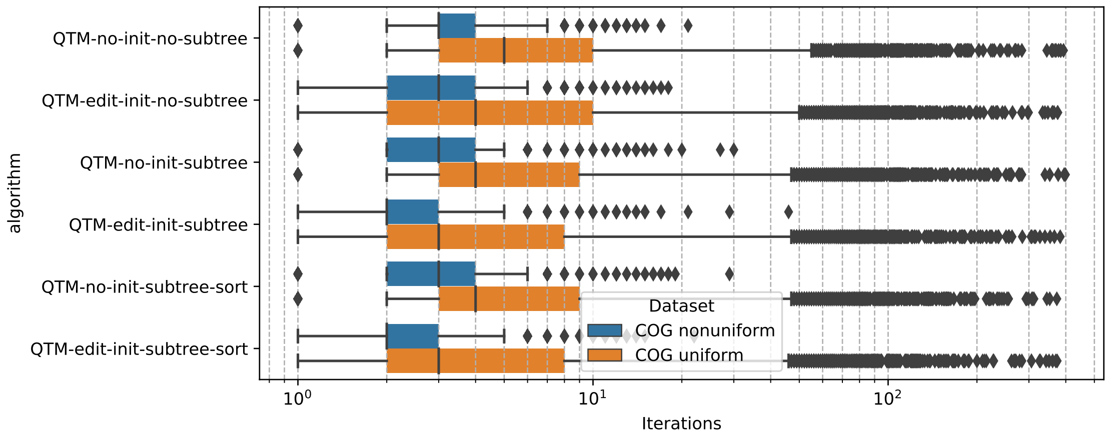
Figure 5.4 shows a plot for the graphs with only a lower bound and compares the minimum solution of QTM to the lower bound. The x-axis shows the given lower bound for the COG graph set in a logarithmic scale, and the y-axis shows the ratio of QTM's solution to the lower bound. QTM is close to the lower bound for graphs, where the lower bound is small. We can also see that for large lower bounds, often a variant of QTM with subtree move finds the minimum solution. On the one hand, QTM can be close to the lower bound. For example, for one graph, the lower bound is 1240 edits, while QTM solved it with 1243 edits. The ratio is then about 1.0016. Here we know that the lower bound and the solution of QTM are very close to the exact solution. On the other hand, we have a graph where the lower bound is 16930 edits. QTM found a minimum solution of 29868 edits with subtree move enabled, where we reach the maximum ratio of 1.7646. We can not say how far away the solution of QTM is from the exact solution.

For nonuniform edit costs, we have exact solutions to 3685 of the 3964 graphs. Here, we simply do not have any information about the solution of the remaining 279 graphs, and so have no way of knowing how close QTM is to the exact solution. QTM finds the exact solution for 3681 of the 3865 graphs where we have an exact solution. This includes all different variants of QTM, over 10 different seed for the randomization. For the remaining four graphs, QTM is at least very close to the exact edit costs. The maximum ratio of QTMs best solution to the exact solution cost is 1.00344. The best solution of a variant of QTM is at most 0.4% worse than the exact solution. For all graphs where QTM solved the quasi-threshold editing problem exactly, for 180 graphs, QTM was only able to reach the solution when using subtree move. 49 of these 180 graphs were also only exactly solved by using simple path sort for the subtree move. Overall, for 347 of the 3964 graphs, QTM found the best solution only by using subtree move. This also means that 167 of the 279 graphs where we do not know the exact solution the best solution for the graph was only found by a QTM variant with subtree move enabled. This is nearly 60% of these graphs. Especially for hard graph instances, subtree move can improve the solution quality of QTM.

We can conclude that QTM with nonuniform as well as uniform edit costs already performs well without optimization and is able to find the exact solution for a lot of graphs and is at least close to it on the remaining graphs. Enabling subtree move allows QTM to find the exact solution on a few graphs where QTM, without it, is not able to. Especially for hard instances and nonuniform edit costs, the advantages of subtree move become clear. Sorting simple paths for the subtree move is better for a select few graphs.



(a) Iterations used by QTM until the algorithm is finished.



(b) Iterations used by QTM cutoff when no changes follow in later iterations.

Figure 5.5: Number of iterations used by QTM for the COG protein similarity data set. We compare uniform and nonuniform edit costs and different variants of QTM to each other. We plot the mean, 5th to 95th percentile and, in addition, outliers.

5.5 Convergence

We already examined the time per iteration and solution quality of QTM. Another factor that influences performance is how fast QTM converges on the best solution. While the algorithm plays a role here, convergence is also influenced by the graph we examine. We also show that nonuniform edit costs allow for a faster convergence. Randomization also plays a big role, which is why we also examine QTM cut off after the last change to edit costs happened. This way, we ignore large plateaus that do not change anything. Additionally, we examine QTM after 20 iterations. So we can tell if the solution of QTM is already close to the best solution, it will find after only a few iterations.

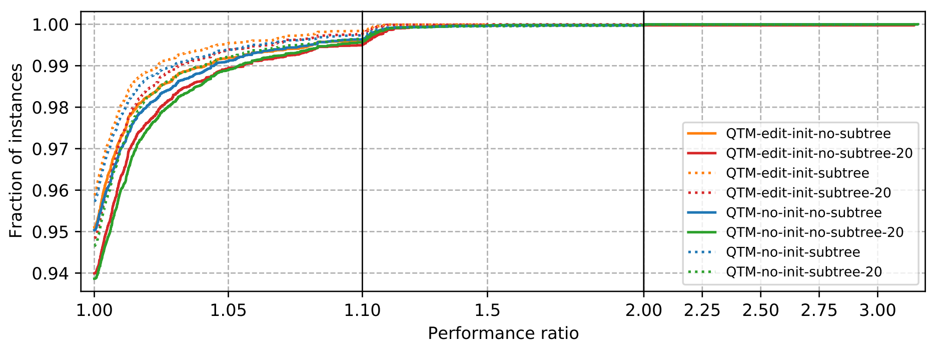
Figure 5.5 shows the iterations QTM needs for different variants of the algorithm on the COG protein similarity graph set. We focus on the editing and no initialization to increase readability. We compare variants with and without subtree move and simple path sort for the subtree move. The plots also compare uniform to nonuniform edit costs. The x-axis shows the iterations needed on a logarithmic scale. The y-axis lists the different variants of QTM. The whiskers extend to the 5th and 95th percentile, and we additionally plot outliers outside these percentiles. The vertical line on the box shows the median for the iterations. For every QTM variant, we show uniform and nonuniform edit costs.

In Figure 5.5a, we see the plot for the used iterations for different variants of QTM using nonuniform and uniform edit costs. We allowed a maximum of 400 iterations and a plateau size of 100 iterations for the randomization, where the edits or edit costs can stay the same. We see that for nonuniform edit costs, all QTM variants on average need fewer iterations to find the best solution. The median is below five for every variant. For every QTM variant, the 95th percentile extends to 100 iterations. Which means there are still graphs where QTM needs more iterations to finish. However, this does not mean that QTM converges slower for these graphs. Looking at iterations where the edit costs still change, we see that QTM reaches the solution for most of these graphs very quickly and simply uses the full plateau size for randomization.

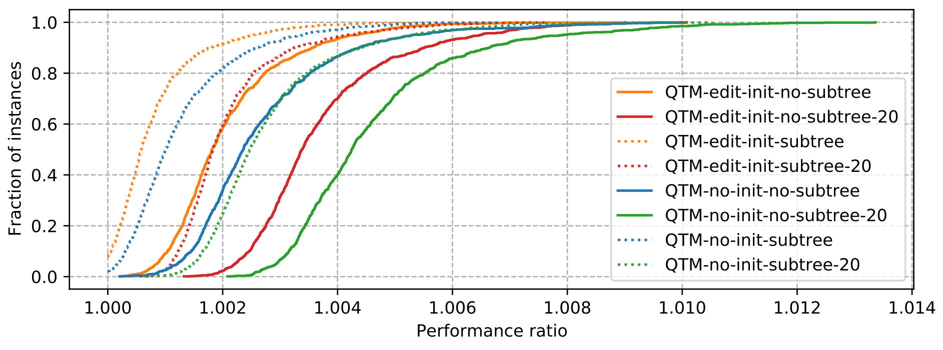
The subtree move does not have a large visible influence on the iterations but moves at least the median for the editing initialization by one. We also have a few outliers that need more iterations with subtree move enabled. For uniform edit costs, QTM needs more iterations on average. Some graphs even need the full 400 iterations we allowed. We also can not make out any visible influence of the subtree move. The median lies at close to 100, which shows us that QTM more often than not needs the full plateau size we gave, which is not changed by the subtree move optimization. This stems from the fact that for this graph set, there exist a lot of equivalent solutions in the number of edits for uniform edit costs. The authors of [Ham21] show there exists more than one solution for most graphs. For some of the graphs, there are more than one million equivalent solutions. The randomization only stops if it has no choices anymore or the plateau for iterations with no changes is reached, in this case, 100. For nonuniform edit costs, most graphs have only one valid, best solution. QTM reaches here, for most graphs, a point where no equivalent choices in relation to edit costs exist and therefore does not need the full plateau size. QTM finds the best solution or a local minimum it can not escape. This also explains why QTM, using nonuniform edit costs, does match the best solution on a smaller fraction of graph instances.

To reduce the influence of the randomization on the results, we cut off the iterations for QTM as soon as the end result is reached and excluded iterations that did not change the edits or edit costs. Figure 5.5b shows these results. For nonuniform costs, the algorithms need fewer than 10 iterations for most graphs to reach the end result. Even the furthest outlier only needs below 50 iterations. The influence of the subtree move is again minimal. For uniform edit costs, a big difference to the plot in Figure 5.5a is noticeable. The median is now below 10 iterations, and all algorithms need fewer than 100 iterations on most graphs to reach the end result. Though we still have a lot of outliers that need the maximum of 400 iterations or come close to it. We also can now see a small influence of the subtree move. Similar to nonuniform edit costs, the median for the iterations is moved by one by enabling subtree move.

To give more insight into how subtree move influences the convergence of QTM, we again look at performance profiles in Figure 5.6. The profiles show the edit costs for different variants of QTM after up to 20 iterations, or after a maximum of 400 iterations. We again only include the editing initialization and no initialization to reduce the number of algorithm variants. In these plots, we only show uniform edit costs for both COG and Facebook graph set. For nonuniform edit costs on the COG graph set, we instead found that with very few exceptions, there is no difference between the edit costs after 20 or 400 iterations. For this graph set, it is enough to run QTM with 20 iterations to reach the best solution. Looking back at Figure 5.5b, we can see that there are only eight graph instances where the edit costs still change after 20 iterations. For uniform edit costs on the same graph set, we can see in Figure 5.6a that there exist a difference between 20 and 400 iterations. The number of iterations matters more than the chosen initialization. We can also see that subtree move plays a bigger role for the solution quality than the number



(a) Variants of QTM on the COG protein similarity graph set using uniform edit costs.



(b) Variants of QTM on the Facebook 100 graph set using uniform edit costs.

Figure 5.6: Performance profiles for the COG protein similarity and Facebook graph set comparing different initializations of QTM and the subtree move optimization after 20 iterations and at a maximum of 400 iterations.

of iterations. While more iterations still improve the result, choosing to enable subtree move can mitigate the disadvantage of fewer iterations. We can say that while the number of iterations makes a difference, QTM already performs well with fewer iterations. For uniform edit costs, it is a good option to enable subtree move with 20 iterations, instead of not enabling it and allowing up to 400 iterations.

We also evaluate the convergence of QTM on the Facebook graph set with uniform edit costs in Figure 5.6b. The set includes harder and larger graphs, and QTM needs more iterations to solve them. As expected, the number of iterations makes a difference for the solution quality of QTM on this graph set. The differences are still very small. The worst variant of QTM with 20 iterations for a graph instance is only 1.4% worse than the best solution for this graph. We also can see that subtree move can save iterations. QTM, with the editing initialization and a maximum of 400 iterations, is close in performance to the same initialization with subtree move enabled and up to 20 iterations. While subtree move does not scale well, in practice it might find use for large graphs when we only want very few iterations.

In conclusion, we can say that QTM converges very quickly. Often, 20 iterations are enough to reach good solutions. Especially for nonuniform edit costs, few iterations are enough. For most graphs in our graph set, more iterations do not make a difference. Subtree move also improves the convergence for uniform edit costs. For graphs in the sizes we examined, it can be a good choice to enable subtree move instead of allowing more iterations for QTM.

5.6 Difference in Solutions

We can look at the quasi-threshold editing problem from another perspective. Before, we have looked either at uniform edit costs or nonuniform edit costs, where every node pair had its own edit costs. We can compromise this approach and have different edit costs for removing and inserting edges. As we can define them ourselves, we do not need them as part of the graph set. We can evaluate this approach on a graph set where we do not have access to edit costs, for example, the Facebook 100 graph set. We executed the variants of QTM again with a maximum of 400 iterations and a plateau size of 100. As the absolute value of the edit costs does not matter for the solutions of QTM in the end we used different ratios for the edit costs. The ratio is defined as the cost of inserting an edge divided through the cost of removing an edge of the graph: $ratio = \frac{i}{r}$.

We first evaluated a ratio of 2. This means inserting an edge is twice as expensive as removing an edge. We notice that on average, 98.51% of the edits QTM finds for the Facebook 100 graph set are removing an edge. This also means that we still insert a few edges, although the operation is twice as expensive. We overall need, on average, 1.23% more edits to solve the graph set with the ratio of 2 compared to uniform edit costs. QTM also needs fewer iterations, on average, to reach the best solution. While we need 383 iterations for the uniform case, we need 358 iterations on average for this ratio of edit costs.

As a second approach, we evaluated a ratio of 0.5. Removing an edge is now twice as expensive as inserting an edge. Surprisingly, most of the edits are still removing an edge, although the edit cost is twice as high. Of all edits, on average, 87.61% are still removing an edge. Overall, QTM needs, on average, 2.35% more edits to solve the graph set compared to uniform edit costs. While a few edits are replaced by inserting edges, we can say that for the Facebook graph set, on average, more edges are removed than inserted when solving the quasi-threshold editing problem. This ratio of remove to insert edit costs also reduces the used iteration for the graph set to 301 iterations on average, compared to 382 iterations for uniform edit costs.

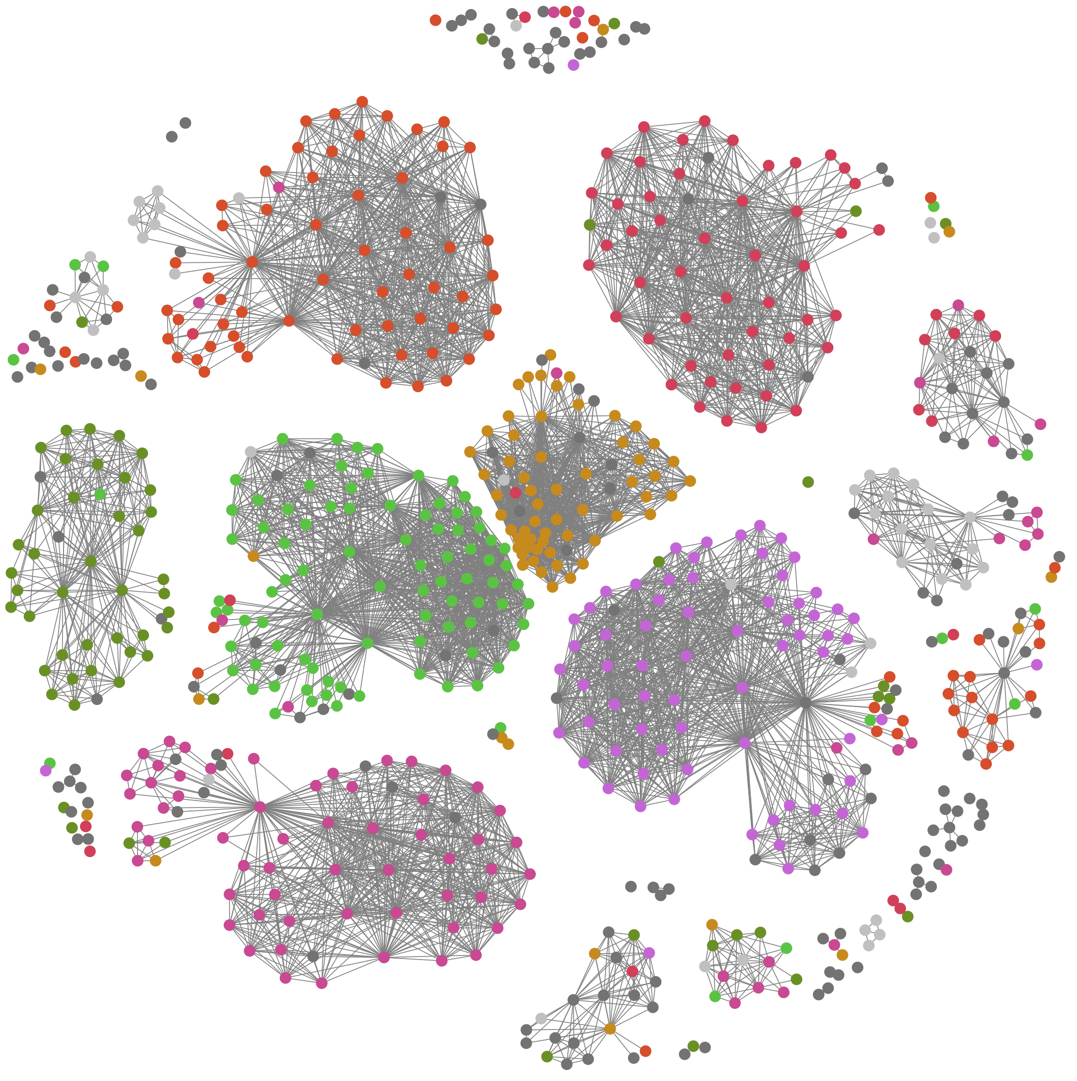


Figure 5.7: Edited graph of Caltech36, one graph of the Facebook 100 graph set. The solution was reached by QTM with a seed of 0, no initialization, and the subtree move optimization disabled. The colors mark different dorms for the people. We have no information about the dorms for dark-gray nodes. We use 1 as the edit cost for removing an edge and 2 for inserting an edge.

To give more insight into different solutions of QTM, we examine one graph of the Facebook 100 graph set and show the result for the different edit costs ratios. We compare different ratios for the edit costs on this graph. We also examine the influence of the subtree move optimization on this graph. The graph we examine is one of the smaller graphs in the graph set. It shows the Facebook network of the California Institute of Technology in the USA and is called Caltech36 in our experiments. The graph has 769 nodes and 16656 edges. We have information about the dorm for most of the people and use this information to evaluate the effectiveness of QTM in the context of community detection.

For all examples on this graph, we use a seed of zero for the randomization, no initialization and the subtree move optimization disabled. We choose this variant because the difference for the different edit costs is clearly visible. We have QTM using uniform edit costs as a base case for this graph. QTM uses 11425 edits for this graph, consisting of 10272 edge removals and 1153 edge insertion. For the solution of the graph using uniform edit costs, QTM needs 275 iterations and 4.51 seconds. The edited graph only has 7537 edges, which is fewer than half of the original graph.

We now execute the same QTM variant on the graph using a different edit cost ratio. Figure 5.7 shows the quasi-threshold graph QTM found for this graph using an edit cost ratio of 2: the edit cost for edge insertions is 2 and the edit cost for edge removals 1. The figure additionally shows the dorm for every person coded by color, to give an informal argument for the quality of the community detection. For dark-gray nodes, we have no information about the dorm. Most nodes in a cluster share the same color and are therefore part of the same dorm. We can also find a few small examples of overlapping communities in the clusters. One example is the cluster consisting of nodes in orange and bright-gray in the top left of the graph. One node connects two communities of different dorms. The variant of QTM we used for this example figure of the graph used 11736 edits to solve the quasi-threshold editing problem. For a ratio of 2, over 96.3% of the edits are removing an edge. QTM removes 11305 edges while only inserting 431 edges. Overall, we need 2.72% more edits than with uniform edit cost. We also remove 67.7% of all edges from the graph. The shown edited graph only has 5782 edges remaining. Changing the ratio of edit costs reduces the used iterations to 190, with a running time of 3.28 seconds for this graph.

In Figure 5.8, we see the edited graph after using the same variant of QTM with the opposite ratio of edit costs. Removing an edge is now twice as expensive as inserting an edge. Just from a first look, we can tell that QTM connects more of the communities. We have more than one person that connects two dorms. The clusters thus consist of nodes in different colors. The edited graph also has a lot more edges than the graph in the previous figure. To put this into numbers, QTM needs 12114 edits to solve this graph, consisting of 2995 edge insertions and 9119 edge removals. The edited graph then has 10532 edges, 82% more than the result, using an edit cost ratio of 2. QTM also uses 6.03% more edits than the solution with uniform edit costs, and 3.22% more edits than the solution with an edit cost ratio of 2. QTM with this edit cost ratio is again faster than QTM using uniform edit costs while using 156 iterations and 3.96 seconds to edit this graph.

In conclusion, we can say that the edit cost ratio influences the result QTM finds, though we still remove a lot more edges than insert them. This is even the case when the remove operation is twice as expensive. In our example graph, we can see that QTM is able to find communities given by the data. We also can find intersecting communities in the quasi-threshold graph.

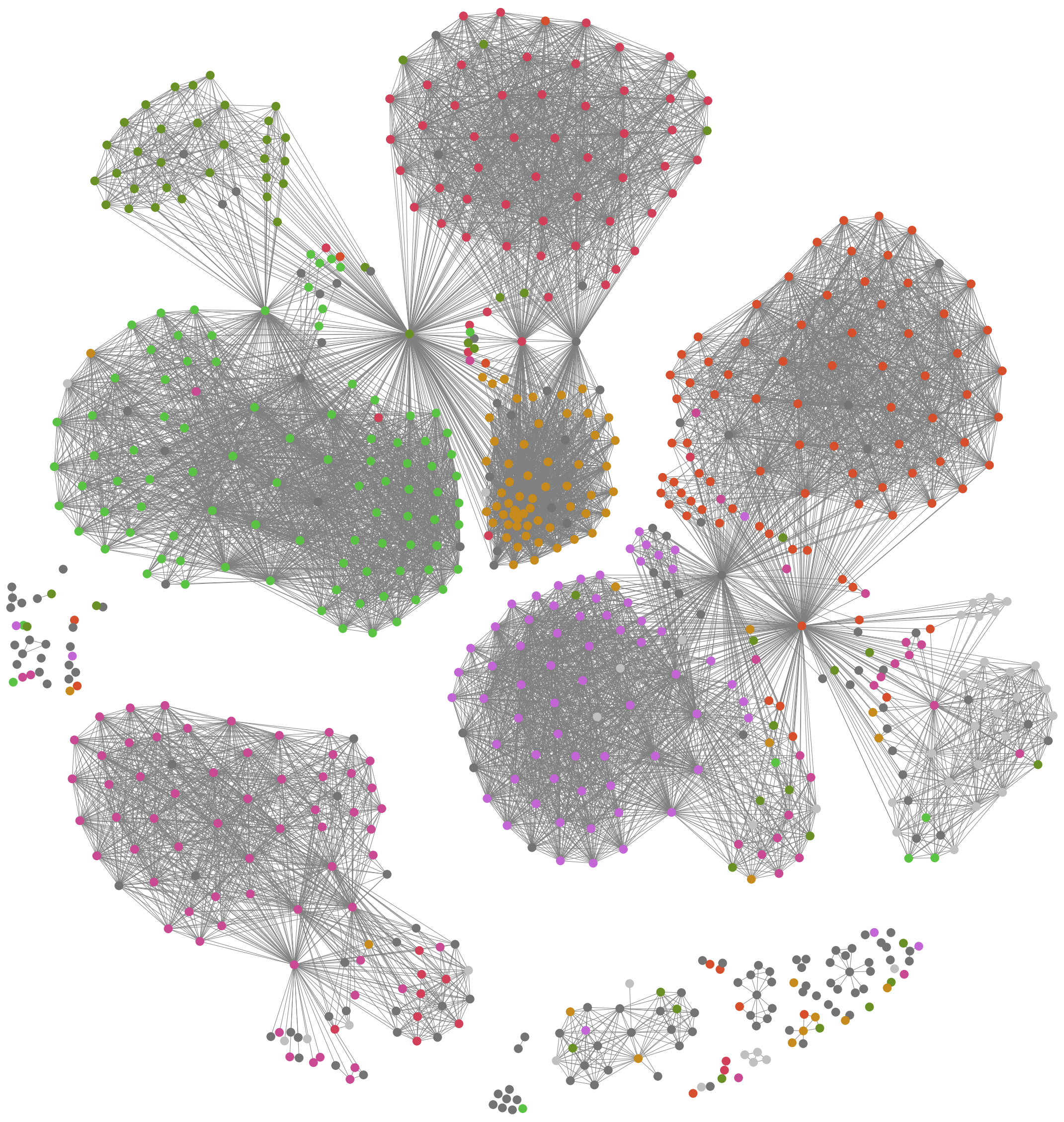


Figure 5.8: Edited graph of Caltech36, one graph of the Facebook 100 graph set. The solution was reached by QTM with a seed of 0, no initialization, and the subtree move optimization enabled. The colors mark different dorms for the people. We have no information about the dorms for dark-gray nodes. We use 2 as the edit cost for removing an edge and 1 for inserting an edge.

6. Conclusion

We have extended the heuristic Quasi-Threshold Mover to consider nonuniform edit costs. We have shown that with few changes, the concept of local moving is a good approach to solve the quasi-threshold editing problem with nonuniform edit costs. Furthermore, we have shown that our algorithm correctly solves the problem and that these changes come with an increase in running time depending on the edit costs. As this running time is only pseudo-polynomial, we additionally show a quadratic worst-case running time per round for the algorithm. We introduced an optimization for QTM in the form of subtree move that improves the solution quality for nonuniform as well as uniform edit costs. We use a similar concept to local move in this optimization and have shown that our algorithm executes the subtree move correctly. The running time, although, depends on the size of the subtree.

We evaluated the extended QTM and our subtree move optimization on the COG protein similarity and the Facebook 100 graph set. The COG protein similarity graph set includes edit costs for every node pair to allow evaluating nonuniform edit cost. The solution quality of QTM suffers when considering nonuniform edit costs. The variants of QTM match the best solution QTM finds on a smaller fraction of graphs. We also have more outliers where a variant of QTM is a lot worse than the best solution on this graph. The performance in solution quality can be improved by subtree move. We are able to match the exact solution on more graphs in the COG graph set. The subtree move comes with an increase in running time and does not scale well for large graphs. Nevertheless, the optimization might be an option if we only use a small number of iterations for large graphs. We also have shown that subtree move can improve the solution of QTM for uniform edit costs. QTM already performs very well using uniform edit cost, which reduces the influence subtree move has on the solution of QTM. Experiments on the large Facebook 100 graph set show that subtree move can be an option to reduce the necessary iterations on large graphs. Our experiments showed that QTM converges very quickly for nonuniform edit costs. Compared to QTM using uniform edit costs, where QTM converges on average in 100 iterations, we reach the result in 10 or fewer iterations for most graphs. This is a sign that QTM can get stuck in local minima it can not escape. For uniform edit costs, this is less often the case.

Furthermore, we evaluate different ratios for edit costs on the Facebook 100 graph set. The experiments showed that the quasi-threshold problem is often solved by removing more edges than inserting edges. When making inserting edges cheaper, most of the edits we find are still removing an edge. However, we insert more edges than QTM using uniform edit costs. Looking at an example graph, the edited graph can change a lot by using different edit costs. For example, we see more intersecting communities with cheaper edge insertions, and the edited graph overall has more edges.

Future Work

The topic of heuristic quasi-threshold editing allows different paths for future research. One topic is to find a way to escape the local minima where QTM gets stuck while using nonuniform edit cost. Currently, QTM only executes a move if it saves edit costs or randomizes the move if it is equivalent in edit costs. One approach would be to allow a temporary increase of the edit costs. For this, the basic idea would be to save a history of local moves and allow moves that are not optimal. We decide if the last moves overall saved edit costs and need a way to reverse local moves in case we did not save edit costs. This approach still has open questions, for example, how a history like that can be constructed. In addition to the move, we also have to save which simple paths we sorted during the local move, as we also have to reverse the simple path sorting step. Another topic for future work is the subtree move optimization. Currently, we move the whole subtree or randomly sample one child to move with the subtree. One approach to engineer the subtree move would be to find an optimal subset of the children to move together with the node we move. For every potential parent, we need a metric to decide which children we move together with the subtree, similar to $\text{child}_{\text{cost}}$ for adopting children. We can use a similar bottom-up approach to find these values. Open questions are here if this is possible in a reasonable running time, as we need these values for every subset of children. One last topic of future research would be further evaluation of nonuniform edit costs. As our experiments were limited to one graph set with nonuniform edit costs, there are still open questions about the performance of QTM while using nonuniform edit cost. For example, a graph set from a different use case might give additional valuable insight on the QTM algorithm and the subtree move optimization.

Bibliography

[BBBT08] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truß. A fixed-parameter approach for weighted cluster editing. In Alvis Brazma, Satoru Miyano, and Tatsuya Akutsu, editors, *Proceedings of the 6th Asia-Pacific Bioinformatics Conference, APBC 2008, 14-17 January 2008, Kyoto, Japan*, volume 6 of *Advances in Bioinformatics and Computational Biology*, pages 211–220. Imperial College Press, 2008.

[BDG$^+$08] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Trans. Knowl. Data Eng.*, 20(2):172–188, 2008.

[BHHW21] Ulrik Brandes, Michael Hamann, Luise Häuser, and Dorothea Wagner. Skeleton-based clustering by quasi-threshold editing, 2021.

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast quasi-threshold editing. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 251–262. Springer, 2015.

[BR17] Julia Brennecke and Olaf Rank. The firm's knowledge network and the transfer of advice among corporate inventors—a multilevel network study. *Research Policy*, 46(4):768–783, 2017.

[CDMG17] Tanmoy Chakraborty, Ayushi Dalmia, Animesh Mukherjee, and Niloy Ganguly. Metrics for community analysis: A survey. *ACM Comput. Surv.*, 50(4):54:1–54:37, 2017.

[CDVMDW19] Lauranne Coppens, Jonathan De Venter, Sandra Mitrović, and Jochen De Weerdt. A comparative study of community detection techniques for large evolving graphs. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 368–384. Springer, 2019.

[Chu08] Frank Pok Man Chu. A simple linear time certifying lbfs-based algorithm for recognizing trivially perfect graphs and their complements. *Inf. Process. Lett.*, 107(1):7–12, 2008.

[For10] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[GHS$^+$20] Lars Gottesbüren, Michael Hamann, Philipp Schoch, Ben Strasser, Dorothea Wagner, and Sven Zühlsdorf. Engineering exact quasi-threshold editing. In Simone Faro and Domenico Cantone, editors, *18th International Symposium on Experimental Algorithms, SEA 2020, June 16-18, 2020, Catania, Italy*, volume 160 of *LIPIcs*, pages 10:1–10:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Gol78] Martin Charles Golumbic. Trivially perfect graphs. *Discret. Math.*, 24(1):105–107, 1978.

[Ham21] Michael Alexander Hamann. *Scalable Community Detection.* PhD thesis, Karlsruher Institut für Technologie (KIT), 2021.

[HKW16] Tanja Hartmann, Andrea Kappes, and Dorothea Wagner. Clustering evolving networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 280–329. Springer, 2016.

[Mar17] Grandjean Martin. Historical Network Analysis: Complex Structures and International Organizations. `http://www.martingrandjean.ch/complex-structures-and-international-organizations/`, 2017. [Online; accessed 22-June-2021].

[NG04] Mark EJ Newman and Michelle Girvan. Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113, 2004.

[NG13] James Nastos and Yong Gao. Familial groups in social networks. *Soc. Networks*, 35(3):439–450, 2013.

[RWB$^+$07] Sven Rahmann, Tobias Wittkop, Jan Baumbach, Marcel Martin, Anke Truss, and Sebastian Böcker. Exact and heuristic algorithms for weighted cluster editing. In *Proc. of Computational Systems Bioinformatics (CSB 2007)*, pages 391 – 401. ACM Press, 2007.

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Comput. Sci. Rev.*, 1(1):27–64, 2007.

[Spi19] Jonas Spinner. Weighted F-free Edge Editing. Bachelor thesis, Karlsruhe Institute of Technology, 2019.

[SSM16] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis. *Netw. Sci.*, 4(4):508–530, 2016.

[TKL97] Roman L Tatusov, Eugene V Koonin, and David J Lipman. A genomic perspective on protein families. *Science*, 278(5338):631–637, 1997.

[TMP12] Amanda L Traud, Peter J Mucha, and Mason A Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.

[WFS10] Guanming Wu, Xin Feng, and Lincoln Stein. A human functional protein interaction network and its application to cancer data analysis. *Genome biology*, 11(5):1–23, 2010.