Karlsruhe Institute of Technology

Inclusion-Minimal Quasi-Threshold Editing

Bachelor Thesis of

Luise Häuser

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisor:	Dr. Michael Hamann

Time Period: 1st May 2020 – 31st August 2020

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, August 29, 2020

Abstract

A graph is called a quasi-threshold graph if and only if it contains neither a path nor a cycle of length as an induced subgraph. The quasi-threshold editing problem is concerned with the question, how to construct a quasi-threshold graph from an arbitrary graph by inserting or deleting as few edges as possible. This problem is $\mathcal{NP}$-hard, here we present an algorithm which determines an inclusion-minimal solution. Additionally, we consider how the number of edits can be further reduced in several iterations. The running time for establishing an initial editing and for each improving iteration is linear with respect to the number of nodes plus the number of edges the graph admits. Data structures for a respective implementation get introduced as well. To make convergence in a local minimum more difficult, we propose an approach how certain decisions in the algorithm can be randomized. In experiments on various testing instances, the presented algorithm leads to better results than from previously proposed heuristics for quasi-threshold editing. Further, we confirm its scalability in practice.

Deutsche Zusammenfassung

Ein Graph ist ein Quasi-Threshold Graph genau dann, wenn er weder einen Pfad noch einen Zyklus der Länge 4 als induzierten Subgraph enthält. Das Quasi-Threshold Editing Problem beschäftigt sich mit der Frage, wie aus einem beliebigen Graphen ein Quasi-Threshold Graph konstruiert werden kann, indem man möglichst wenige Kanten löscht oder einfügt. Dieses Problem ist $\mathcal{NP}$-schwer. Hier stellen wir einen Algorithmus vor, der eine inklusions-minimale Lösung bestimmt. Zusätzlich betrachten wir, wie durch mehrere Iterationen die Anzahl an Edits noch weiter reduziert werden kann. Die Laufzeit für die Konstruktion des initialen Editing sowie für die Ausführung einer Iteration ist linear bezüglich Knotenanzahl plus Kantenanzahl des Eingabegraphen. Datenstrukturen für eine entsprechende Implementierung werden ebenfalls eingeführt. Um Konvergenz in einem lokalen Minimum zu erschweren, schlagen wir einen Ansatz vor, wie bestimmte Entscheidungen im Algorithmus randomisiert werden können. Bei Experimenten mit verschiedenen Testinstanzen führt der vorgestellte Algorithmus zu besseren Ergebnissen als bereits bekannte Heuristiken für Quasi-Threshold Editing. Zudem kann die Skalierbarkeit für die Praxis bestätigt werden.

Contents

1. Introduction

Within the context of this work, we are concerned *quasi-threshold graphs*, also known as *trivially perfect graphs*. They can be characterized by the property that they contain neither a path nor a cycle of length 4 as an induced subgraph. Additionally, several equivalent definitions are provided. The first one is of constructive nature: A graph is a quasi-threshold graph (QTG) if and only if it is either a single vertex or if it is produced by adding a universal vertex to a smaller QTG or by forming the disjoint union of two smaller QTGs. Further, each quasi-threshold graph can be regarded as the transitive closure of a rooted forest, the so called *skeleton*.

We consider the *quasi-threshold editing* problem, i.e. how an arbitrary graph can be transformed into a quasi-threshold graph with a minimum number of edge insertions and deletions. Finding such a minimum editing is $\mathcal{NP}$-hard[NG13]. Hence, we examine how a optimal solution can be approximated.

Community detection is a possible application for quasi-threshold editing [NG13]. As a community we describe a subgraph which is densely connected on the inside but has relatively few edges to vertices on the outside [For10]. If Q is a QTG resulting from an editing for a given graph G, the connected components of Q can be interpreted as communities in G [NG13]. Finding an editing, such that the respective QTG is close to the original graph, therefore helps to detect communities which represent the actual structure of G.

Quasi-threshold editing is $\mathcal{NP}$-hard but fixed-parameter tractable in the number of edits k [Cai96]. In [DP17] an exact approach is proposed which has a polynomial kernel of $\mathcal{O}(k^7)$ vertices. The branch-and-bound algorithm presented in [GHS$^+$20] admits an improved running time of $\mathcal{O}(k^6 \cdot (m+n))$.
The first heuristic algorithm got developed by Nastos and Gao [NG13], but its complexity is quadratic [BHSW15]. Further, the *Quasi-Threshold Mover* is introduced by Brandes et al. in [BHSW15]. This algorithm works on the graph's skeleton and approximates a quasi-threshold editing of minimal cardinality. It starts from an initial editing which is to be improved during several iterations. Such an iteration runs in $\mathcal{O}(m \log \Delta)$.

The class of *cographs* comprises all graphs which do not possess a path of length 4 as an induced subgraph. Hence, they form a super class for quasi-threshold graphs. In the same way as quasi-threshold editing, the editing problem for cographs is defined. In [Cre19] an approach for an inclusion-minimal solution of this problem is proposed. None of the heuristics introduced for quasi-threshold editing admits the same guarantee.

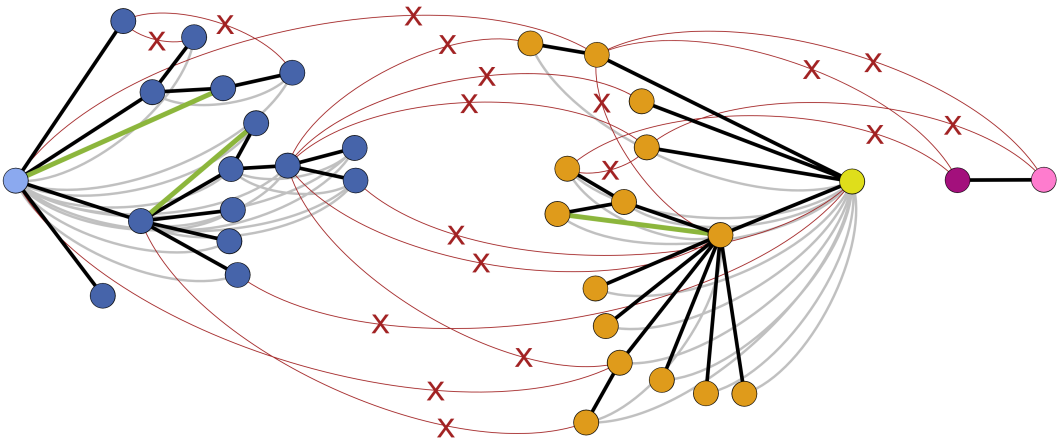


Figure 1.1.: Minimum editing of the graph *karate* with 21 edits, 3 insertions marked in green, 18 deletions marked in red. The skeleton of the resulting QTG is bolded, the transitive edges are indicated in grey. Node colorings correspond both to the communities of *karate* and to the connected components of the edited graph.

Thus, the main goal of this work is to develop such an algorithm. For this purpose, the Quasi-Threshold Mover from [BHSW15] gets improved, such that every step is locally optimal. Additionally, we develop data structures to achieve a running time in $\mathcal{O}(m)$ per iteration. As an approach to prevent converging in a local minimum, we furthermore randomize certain decisions occurring during the algorithm.
We perform experiments on various testing instances, including real world social networks and protein similarity graphs. The evaluation shows the practical capability of inclusion-minimal editings. Further, we observe that the improved algorithm leads to better results and requires less running time than the previously proposed heuristic from Brandes et al.

The thesis is structured as follows: In Chapter 2 we start by providing some formal definitions and notations used throughout the work. Then we go more into detail regarding the structure of QTGs by introducing some properties of subgraph skeletons in Chapter 3. In Chapter 4 we further explain the Quasi-Threshold Mover as it is proposed in [BHSW15]. We continue by discussing how the mover algorithm can be modified in order to obtain a locally minimal solution which is presented in Chapter 5. Based on that, in Chapter 6 we establish an approach for inclusion-minimal quasi-threshold-editing. In Chapter 7 we subsequently introduce the Bucket Queue and the Dynamic Forest, two advanced data structures, used to implement the algorithm with the running time afterwards proven in Chapter 8. Furthermore, we show how random decisions can help to improve the Quasi-Threshold Mover (see Chapter 9). Finally, in Chapter 10 we evaluate the results we obtained by running the algorithm on various instances.

2. Preliminaries

2.1. Basic Notions

Graphs. A *graph* G is a tuple (V, E), where V denotes a set of $n := |V|$ sequentially numbered vertices $V = \{v_1, v_2, \ldots, v_n\}$ and E a set of $m := |E|$ edges. In the context of this thesis unweighted and undirected graphs are considered, thus an edge is a set $\{v_i, v_j\}$ of two vertices $v_i, v_j \in V$. For the purpose of clarity, we sometimes give the related graph in subscript, i.e. we write V_G and E_G instead of V and E.
We say an edge $e = \{x, y\}$ is *incident* to a vertex v_i if $v_i \in e$. Two vertices v_i and v_j are called *adjacent* to each other if there is an edge $\{v_i, v_j\} \in E$.
By $N_G(v_i) = \{v_j \mid \{v_i, v_j\} \in E_G\}$ we denote the *neighborhood* of a vertex $v_i \in V_G$. The *degree* of v_i is defined as $\deg(v_i) = |N_G(v_i)|$. Δ denotes the maximum degree within a given graph.

Substructures. A graph $H = (V_H, E_H)$ is called a *subgraph* of a graph $G = (V_G, E_G)$ if $V_H \subseteq V_G$ and $E_H \subseteq E_G$. If it additionally holds for all vertices $u, v \in V_H$ that $\{u, v\} \in E_H \Leftrightarrow \{u, v\} \in E_G$, we call H an *induced subgraph* of G. We further write $H = G - v$ if $v \in V_G$, $V_H = V_G \setminus \{v\}$ and $E_H = E_G \setminus \{e \in E_G \mid v \in e\}$.

Connectivity. A *path* $P = \{v_1, v_2, \ldots, v_k\}$ is set of vertices such that every pair of consecutive vertices is connected with an edge. If the edge $\{v_1, v_k\}$ exists as well, one speaks of a *cycle*. By P_k (C_k) we denote a path (cycle) containing k vertices. A graph G is called *connected* if G contains a path with endpoints u and v for every vertex pair $u, v \in V_G$. A maximal connected subgraph of an arbitrary graph is called a *connected component*. By C_G we denote the set of connected components.

Trees and forests. A graph T is called a *tree* if any two vertices can be connected with a unique path. In the scope of this work, *rooted trees* are considered, i.e. in every tree one vertex $r_T \in V_T$ is designated the root. For a vertex $v \in V_T$ we define depth(v) as the length of the path with endpoints v and r_T. Furthermore, v's successor p on its path to the root is called the *parent* of v. We write $p = \text{parent}(v)$. Respectively v's *children* are defined as the set $\{u \in V_T \mid \text{parent}(u) = v\}$. We name an vertex an *ancestor* of v if it is v's parent or (recursively) an ancestor of v's parent. Analog, a vertex is called a *descendant* of v if it is either a child of v or recursively a descendant of a child of v.
By T_v we denote the *subtree rooted in* v, i.e. the subgraph of T containing v and all its descendants.
A *forest* is the disjoint union of a set of trees. Hence the definitions provided above are applied with respect to the respective connected component. In particular, a forest T is rooted if every component $C \in C_T$ admits a designated root r_C.

Traversal with depth-first-search. *Depth-first-search* (*DFS*) is a strategy for traversing graphs and especially forests. For every root a DFS is initiated. Then every branch is explored as deep as possible before backtracking. The order in which the vertices are processed is called *lexicographic order*. The complexity of a DFS is in $\mathcal{O}(n + m)$ and even in $\mathcal{O}(n)$ for forests, because they have at most $n - 1$ edges.

2.2. Quasi-Threshold Graphs

In this chapter we introduce the terms and definitions related to Quasi-Threshold graphs.

Definition 2.1. *The graph class of* ***Quasi-Threshold graphs*** *is recursively defined in the following way:*

(i) K_1 *is a Quasi-Threshold-Graph (QTG)*

(ii) From a QTG G *another QTG* G' *can be obtained by adding an* ***universal vertex***

(iii) For two QTGs G_1 *and* G_2 *the* ***disjoint union*** *is a QTG*

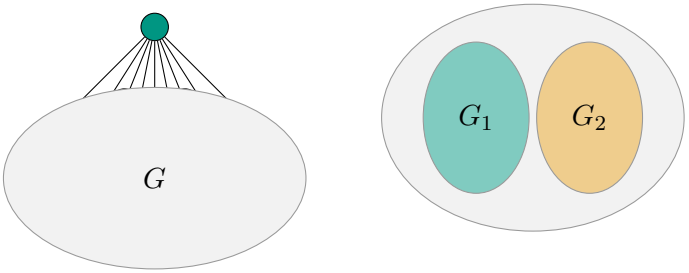


Figure 2.1.: Operations according to recursive definition (schematic)

Definition 2.2. *Let S a rooted forest. We write $u \sim_S v$, if vertices $u, v \in V_S$ are in an ancestor-descendant-relationship in S. For every QTG G there exists a rooted forest S such that S induces G, i.e. $V_S = V_G$ and for $u, v \in E_G$ it holds that $(u, v) \in E_G$ iff. $u \sim_S v$. S is called the **skeleton** of G and we write $G = \mathrm{ind}(S)$[YCC96].*

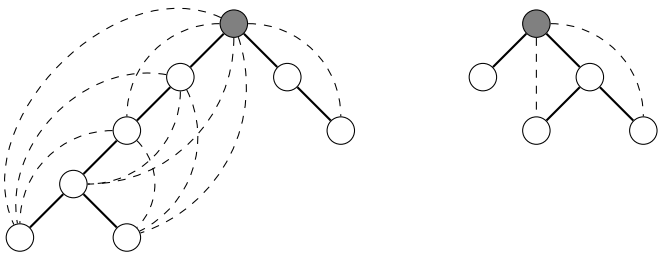


Figure 2.2.: A skeleton and the induced QTG

For a vertex $u \in V_G$ we define S_u as the subtree of S rooted in u. We also note that the QTG induced by S_u is an induced subgraph of G.
When we visualize a QTG, we sometimes only draw the edges of the skeleton and leave out the induced edges in order to make the drawing more readable.

In addition to the already referred definitions, there exist several alternative characterizations for the QTG graph class. In the following, we state two of them which are helpful within the context of this work.

Definition 2.3. *A graph G is a QTG if and only if*

(i) it does not contain a P_4 or a C_4 as an induced subgraph

(ii) for every edge $\{u, v\} \in E_G$ it holds that either $N_G(v) \subseteq N_G(u)$ or $N_G(v) \subseteq N_G(u)$

Lemma 2.4. *The Definitions 2.1, 2.2, 2.3(i) and 2.3(ii) are equivalent[YCC96].*

2.3. The Editing Problem

Definition 2.5. *For a graph G we define an **editing** $H \subseteq \{\{u, v\} \mid u, v \in V_G\}$ as a set of edges between existing vertices in G. $G' = G \Delta H$ describes the graph we obtain by applying H on G. For this purpose every edge $e \in H$ gets toggled, thus $V_{G'} = V_G$ and $E_{G'} = (E_G \setminus E_H) \cup (E_H \setminus E_G)$.*

Given an arbitrary graph G we want find an editing H of minimum size such that $Q = H \Delta G$ is a QTG. With regard to the skeleton S of Q, H comprises the insertion of edges between vertices which are in ancestor-descendant-relationship in S but not adjacent in G as well as the deletion of edges between vertices which are on different branches of S.

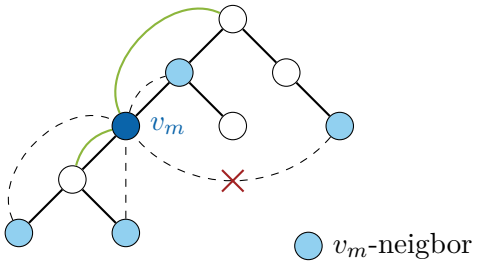


Figure 2.3.: Required edits incident to a fixed vertex v_m, insertions marked in green

Determining a QTG editing of minimum size is a $\mathcal{NP}$-hard problem[NG13], hence the algorithm developed within this work only approximates the minimum number of required edits.

3. Skeletons in Subgraphs

In this section we examine the skeletons of certain subgraphs of a QTG G. At first we focus on G's connected components.

Lemma 3.1. *Let G a QTG and S a skeleton such that $G = \mathrm{ind}(S)$. Then it holds that for every component C of G there is a component S_C of S such that $C = \mathrm{ind}(S_C)$.*

Proof. From Definition 2.2 it follows directly that every edge of the skeleton also exists in the induced graph, i.e. $E_S \subset E_G$. Thus, there cannot be vertices from more than one component of G in any component of S.
It remains to prove that if we pick two vertices u and w from an arbitrary component C of G, they are also in the same connected component of S. In G there is a path $P = (u, v_1, \ldots, v_k, w)$ connecting u and w. According to Definition 2.2 it must hold that $u \sim_S v_1$, $v_k \sim_S w$ and $v_i \sim_S v_{i+1}$ for $1 \leq i < k$. Two vertices can only be in an ancestor-descendant-relationship to each other if there exits a path between them. It follows that each consecutive pair of vertices on P is connected with a path in S. Overall, there is a path from u to w, i.e. they are in the same component of S. □

In the following, certain operations are defined which allow us to remove or isolate vertices from graphs or skeletons. Further, we examine the relationships among resulting structures, particularly we are interested in the skeleton of $G - v$ for $v \in V_G$.

Definition 3.2. *Let G a graph and $v \in V_G$. Then $H = (G \,\mathrm{iso}\, v) \Leftrightarrow V_H = V_G$ and $E_H = E_G \setminus \{\{v, w\} \mid w \in V_G\}$, i.e. H is produced by the isolation of v.*

Definition 3.3. *Let G a QTG, S a skeleton such that $G = \mathrm{ind}(S)$ and $v \in V_G$.*
$T = (S \circledcirc v) \Leftrightarrow V_T = V_S$ and $E_T = (E_S \setminus \{\{v, w\} \mid w \in V_G\}) \cup \{\{p, c\} \mid p$ is parent of v in S, c is child of v in $S\} \Leftrightarrow T$ produced by isolating v and moving v's children below v's old parent.
$T = (S \ominus v) \Leftrightarrow T = ((S \circledcirc v) - v) \Leftrightarrow T$ is produced by removing v and moving v's children below v's old parent.

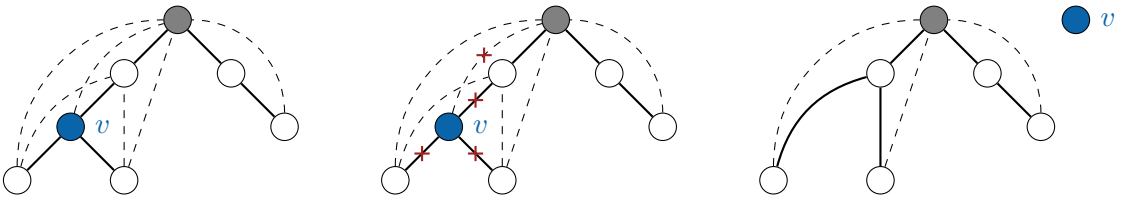


Figure 3.1.: Illustration of $S \circledcirc v$

Lemma 3.4. *Let G a QTG, S a skeleton such that $G = \text{ind}(S)$ and $v \in V_G$, then $(G \,\text{iso}\, v) = \text{ind}(S \circledcirc v)$.*

Proof. Let $H = (G \,\text{iso}\, v)$ and $T = (S \circledcirc v)$, $x, y \in V_G \setminus \{v\}$. From Definition 2.2 we know that $\{x, y\} \in E_G$ iff. $x \sim_S y$. Furthermore, according to Definition 3.2 $\{x, y\} \in E_H$ iff. $\{x, y\} \in E_G$ and according to Definition 3.3 $x \sim_T y$ iff. $x \sim_S y$. Hence, $\{x, y\} \in E_H$ iff. $x \sim_T y$ in T. Additionally ,v has no neighbors in H and no ancestor-descendant-relationship in T, thus overall $H = \text{ind}(T)$. □

Corollary 3.5. $(G - v) = \text{ind}(S \ominus v)$

Proof. Let $H = (G \,\text{iso}\, v)$ and $T = (S \circledcirc v)$, $x, y \in V_G \setminus \{v\}$. Note that $(G - v) = (H - v)$ and $(S \ominus v) = T - v$, thus we show now $H - v = \text{ind}(T - v)$. According to Lemma 3.4 $H = \text{ind}(T)$, i.e. $\{x, y\} \in E_H$ iff. $x \sim_T y$. As v is isolated both in H and T, it can be removed without any effect on ancestor-descendant-relationships in T or on the edge set of H. Thus, for $x, y \in V_G$ it holds that $\{x, y\} \in E_{H-v}$ iff $\{x, y\} \in E_H$ and $x \sim_{T-v} y$ iff $x \sim_T y$. Overall, $\{x, y\} \in E_{H-v}$ iff $x \sim_{T-v} y$, i.e. $H - v = \text{ind}(T - v)$. □

Now we are aware of the structure of the skeleton of $G - v$ for $v \in V_G$, what we can subsequently make use of.

3.1. Simple Paths

Definition 3.6. *As **simple path** denote a maximal subtree of a forest, for which it holds that all but the lowermost vertices have exactly one child.*

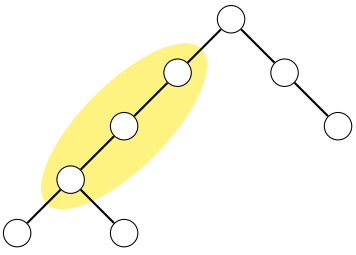


Figure 3.2.: A skeleton with a marked simple path

Definition 3.7. *Let S and S' two skeletons. We write $S \cong S'$ iff S and S' are equivalent up to the order of vertices on simple paths.*

The following lemma shows us that the induced QTG is independent from the order of the vertices on simple paths. In particular, this allows us to reorder paths. To prove the statement, we work on the recursive QTG structure and use the knowledge about subgraph skeletons gained above.

Lemma 3.8. *Let S and S' skeletons. It holds that $S \cong S' \Leftrightarrow \text{ind}(S) = \text{ind}(S')$*

Proof. "⇒": Let S and S' skeletons such that $S \cong S'$. In every skeleton it holds that the ancestors (descendants) of the uppermost (lowermost) vertex of a simple path are ancestors (descendants) of every vertex on the path. Additionally, on the path all vertices are in an ancestor-descendant-relationship to each other. Thus for each vertex its ancestor-descendant-relationships in the skeleton, and therefore its neighbors in the induced graph, are independent from the position on the simple path it belongs to. Overall it follows that the QTGs induced by S and S' are identical.

"⇐": Let G an arbitrary QTG. Show that all skeletons inducing G are in the same equivalence class of $\cong$.
If $G = K_1$, K_1 is also the only possible skeleton and thus the only representative of the equivalence class.
Otherwise we consider G's recursive structure according to Definition 2.1. We take it for granted that the claim holds for preexistent QTGs and show that it is maintained under all construction steps possibly producing G.

Case 1: G is the disjoint union of two QTGs G_1 and G_2.
For the sake of contradiction we assume there are skeletons S and S' such that $ind(S) = \text{ind}(S') = G$ and $S \ncong S'$. As G_1 and G_2 are different components of G we know from Definition 3.1 that S (S') is the disjoint union of S_1 (S'_1) and S_2 (S'_2), such that $G_1 = \text{ind}(S_1)$ ($G_1 = \text{ind}(S'_1)$) and $G_2 = \text{ind}(S_2)$ ($G_2 = \text{ind}(S'_2)$). Hence if $S' \ncong S$ it follows that $S'_1 \ncong S_1$ or $S'_2 \ncong S_2$. This contradicts the prerequisite that the claim holds for G_1 and G_2, therefore $S' \cong S$.

Case 2: G is produced by adding a universal vertex r to another QTG G'.
Let S an arbitrary skeleton inducing G. As for every vertex $v \in V_{G'}$ $(r, v) \in E_G$, it must hold that $v \sim_S r$ for every $v \in V_G \setminus \{r\}$

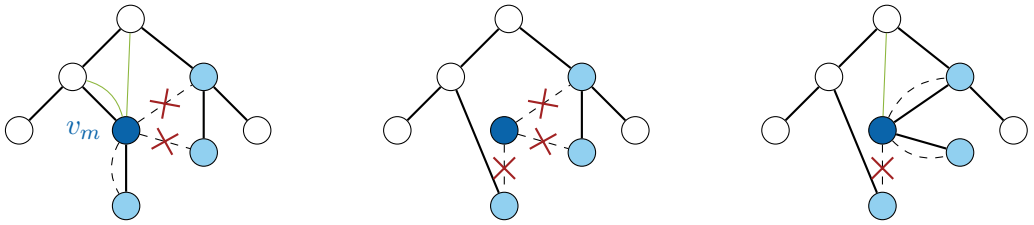
Case 2.1.: G' is disconnected
Assume r has an ancestor r' in S. Then the neighborhood of r' must be a (non-strict) superset of r's neighborhood, i.e. r' must be universal as well. This would establish connectivity in G' which implies that r' cannot exist. It follows that r needs to become an ancestor of all vertices, i.e. the root of any skeleton inducing G.

Case 2.2.: G' is connected
Also in this case the neighborhood of every ancestor of r in S must be a (non-strict) superset of r's neighborhood, thus r must lie on the same simple path P as G''s root r'.

Next we consider another skeleton S' such that $G = \text{ind}(S')$. We can apply Corollary 3.5 to $G' = G - r$ and see that $\text{ind}(S' \ominus r) = G' = \text{ind}(S \ominus r)$. As we assume that the claim holds for G', it follows that $S' \ominus r \cong S \ominus r$. As furthermore in both cases described above r has a position in S which is unique upto the position on its simple path, $S' \cong S$ holds as well. □

4. The Quasi-Threshold Mover

In [BHSW15], Brandes et al. proposed the Quasi-Threshold Mover(QTM) as a heuristic for the quasi-threshold editing problem. The algorithm is initialized with an editing for the given graph. Then several iterations are executed in order to improve this editing. Within every iteration step, the nodes are processed one after the other. For the current vertex v_m a *local move* is performed. For this purpose we work on the skeleton which induces the QTG resulting from the current editing. At first, v_m gets isolated according to the $\circledcirc$ operation. Further, the algorithm tries to determine a position for v_m leading to a smaller number of edits incident to it.
The algorithm terminates, when during an iteration the number of edits could not be reduced or after a defined maximum number of iterations.



Initial editing | Isolate v_m | Move v_m to new position

Figure 4.1.: Steps of a local move, insertions marked in green, v_m-neighbors in light blue

To simplify the description of the algorithm, we introduce a virtual root r which is adjacent to all vertices. Hence, it suffices to consider the case of a single connected component.

4.1. The Algorithm

At first, we introduce some concepts occurring in the local mover algorithm:

Definition 4.1. *Let G a graph, $v_m, w \in V(G)$, H a QTG editing of G and S a skeleton of $Q = G \Delta H$.*

- *We define the **child closeness** $\text{child}_{\text{close}}(w)$ of w as the number of v_m-neighbors minus the number of non-v_m-neighbors in the subtree rooted in w*
- *If $\text{child}_{\text{close}}(w) > 0$, we say w is a **close child***
- *By $\delta_{v_m}(S, w)$ denote the number of edits incident to v_m when placing it below w and adopting w's close children. A vertex $u \in V_G$ for which $\delta_{v_m}(S, u)$ is minimal among all vertices in G is called a **best parent** for v_m in S and we denote $\delta_{v_m}(S, u)$ by $\delta^*_{v_m}(S)$*

We note that the best parent is not necessarily unique, as there are possibly several positions where placing v_m results in the same number of edits. Whenever there are equally good parents, we pick an arbitrary one. How this decision can be properly randomized is explained in Chapter 9.
Moreover, we keep in mind that the value of δ_{v_m} and thus whether a node is a best parent depends on the skeleton S.

The algorithm described in [BHSW15] determines the target position for v_m by placing it below a best parent and adopting this parent's close children. To identify best parents and close children, we work on the existing skeleton S, from which v_m got isolated. We proceed in bottom up fashion, i.e. starting from skeleton's leaves. For every node u that is processed, we determine a best parent in subgraph induced by S_u. Best parents regarding the subtrees rooted in u's children are already determined, as the algorithm works bottom up. Thus, we can check, whether one of them is also a best parent regarding S_u or whether u itself is a better parent. To identify best parents, the child closeness is required. Hence, the respective values are calculated in the course of the same traversal. Proceeding bottom up, we finally reach r. We obtain the desired best parent and further we know which of its children are close.

Now that we have explained the algorithm's basic proceeding, in the following we go more into detail. For a better understanding you can refer to the pseudo code provided in Algorithm 4.1. At first we introduce a score which makes it possible to identify a best parent within a subtree.
For a node $u \in V_G$ and a node w in S_u we define the set $X_u(w) \subseteq V_{S_u}$. $X_u(w)$ contains w, its ancestors in S_u, its close children and their descendants. By $\sigma_u(w)$ we denote the difference of the number of v_m-neighbors and non-v_m-neighbors in $X_u(w)$. The algorithm determines for every vertex $u \in V_G$ a vertex $\text{p}_{\text{best}}(u) \in V_{S_u}$ which maximizes $\sigma_u(w)$ among all vertices w in this subtree. $\text{score}_{\max}(u)$ is then defined as $\sigma_u(\text{p}_{\text{best}}(u))$.

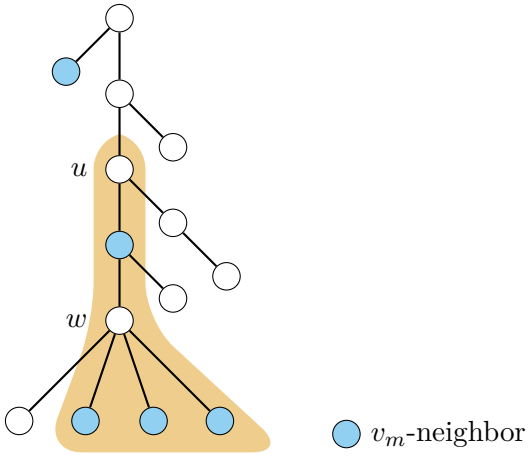


Figure 4.2.: Example with $\text{score}_{\max}(u) = 2$
$X_u(w)$ is highlighted in orange that $w = \text{p}_{\text{best}}(u)$

When we proceed in bottom up style, $\text{score}_{\max}$ can be calculated quite easily. If the vertex u we process, is a leaf, $\text{score}_{\max}(u)$ is 1 if u is a v_m-neighbor and -1 otherwise. If u is an inner vertex, we at first determine c, the child of u with the maximum $\text{score}_{\max}$ among all children. A best parent regarding S_u is either u itself or $\text{p}_{\text{best}}(c)$ (see [BHSW15], Theorem 3). Hence, we only need to calculate $\sigma_u(\text{p}_{\text{best}}(c))$ and $\sigma_u(u)$. If u is a v_m-neighbor (non-v_m-neighbor), in $X_u(\text{p}_{\text{best}}(c))$ there is one v_m-neighbor (non-v_m-neighbor) more than in $X_c(\text{p}_{\text{best}}(c))$. Thus, we need to increase (decrease) $\text{score}_{\max}(c)$ by 1 to obtain $\sigma_u(\text{p}_{\text{best}}(c))$. The resulting value is compared to $\sigma_u(u)$ to check whether u is a better parent. If this is the case, it holds that $\text{score}_{\max}(u) = \sigma_u(u)$ and $\text{p}_{\text{best}}(u) = u$, otherwise, $\text{score}_{\max}(u) = \sigma_u(\text{p}_{\text{best}}(c))$ and $\text{p}_{\text{best}}(u) = \text{p}_{\text{best}}(c)$.
It is left to explain how $\sigma_u(u)$ can be calculated. $X_u(u)$ only contains u, its close children and their descendants. Further, the $\text{child}_{\text{close}}$ values of u's children are already calculated, because we proceed bottom up. Thus, can simply sum up the child closeness of u's close children and add or subtract 1 depending on whether u is a v_m neighbor. Like this we obtain $\sigma_u(u)$.
When processing a node u during the bottom-up traversal, we also need to determine $\text{child}_{\text{close}}(u)$, because this value is necessary for the calculation of $\text{score}_{\max}$ of subsequent nodes. At first we can set $\text{child}_{\text{close}}(u)$ to $\sigma_u(u)$, as this value already covers the subtrees rooted in u's close children and u itself. Further, we start a DFS below u. Subtrees of close children get skipped, in the remaining subgraph we count v_m-neighbors and non-v_m-neighbors to determine $\text{child}_{\text{close}}(u)$.

Algorithm 4.1: localMove

```
1  S ← S ⊙ v_m
2  foreach node u in bottom up fashion do
3      c ← child with maximum score_max among u-children;
4      σ_u(u) ← sum of child_close of close u-children;
5      if u is a v_m-neighbor then
6          σ_u(u) ← σ_u(u) + 1;
7          σ_u(p_best(c)) ← σ_c(p_best(c)) + 1;
8      else
9          σ_u(u) ← σ_u(u) − 1;
10         σ_u(p_best(c)) ← σ_c(p_best(c)) − 1;
11     if σ_u(u) ≥ σ_u(p_best(c)) then
12         score_max(u) ← σ_u(u);
13         p_best(u) ← u;
14     else
15         score_max(u) ← σ_u(p_best(c));
16         p_best(u) ← p_best(c);
17     child_close(u) ← σ_u(u);
18     determine child_close(u) by DFS in S_u;
19 move v_m below p_best(r) and adopt close children;
```

4.2. Details and Optimizations

The algorithm how it got introduced so far, admits a quadratic complexity. Hence, Brandes et al. present it with several optimizations. We explain them in the following, in Algorithm 4.2 and Algorithm 4.3 the corresponding pseudo code is provided.
The vertices to consider during a local move are kept in a maximum priority queue using the vertices depth in the current skeleton as a key. The implementation from [BHSW15] uses the Level Queue explained in Section 4.3.1. In Section 7.1 we propose the Bucket Queue, another priority queue adapted for this specific use case.
Further, not all nodes necessarily need to be processed during the bottom up traversal. Let u a vertex with $\text{score}_{\max}(u) \leq 0$ and $\text{child}_{\text{close}}(u) \leq 0$. As the isolated position of v_m below r is related to a score of 0, u does not come into consideration to be a best parent. As u also is not a close child it will not affect the score of any other vertex. Consequently we leave out vertices which are no close children and do not admit a score greater than 0. In particular, this means that values of $\text{score}_{\max}$ and $\text{child}_{\text{close}}$ which are below 0, do not need to be exactly calculated. A vertex u can only be a close child if it is a v_m-neighbor or if it has a close child. Likewise it can only admit $\text{score}_{\max}(u) > 0$ if it is a v_m-neighbor or if it has a child w with $\text{score}_{\max}(u) > 0$. Therefore it suffices to initially fill the queue with the v_m-neighbors. Whenever we encounter a vertex which is a close child or admits a score greater than 0, we insert its parent into the queue (see Algorithm 4.2, line 27 f.).

Moreover, it is too time consuming, to fully execute every DFS for the calculation of $\text{child}_{\text{close}}$. Instead, we only partially traverse the subtree below u, as it is implemented in lines 13-24 of Algorithm 4.2. For this approach, we use the fact that child closeness values below 0 do not need to be exactly calculated. $\text{child}_{\text{close}}(u)$ is initialized with $\sigma_u(u)$ which already covers the subtrees below close children of this node. The subtrees left to traverse can only lead to a decrease of this initial value. Hence, we only start a DFS below u if $\sigma_u(u) \geq 0$. Additionally, for every vertex u we store a pointer $\text{DFS}_{\text{next}}(u)$ that works as a shortcut to skip already traversed subgraphs. Initially $\text{DFS}_{\text{next}}(u)$ points at u itself.

Whenever $\text{child}_{\text{close}}(u)$ falls below 0, we abort the DFS and store the node in which the DFS stopped as $\text{DFS}_{\text{next}}(u)$. When processing subsequent nodes, we can use it as a shortcut, because the subtree we skip when jumping from u to $\text{DFS}_{\text{next}}(u)$ decreases the child closeness by 1.
Let x the node currently processed in the DFS. If x did not get touched during this local move or if $\text{child}_{\text{close}}(x) < 0$, we know that x is not a close child. Thus, $\text{child}_{\text{close}}(u)$ gets decreased by 1. If now $\text{child}_{\text{close}}(u)$ is below 0, we store $\text{DFS}_{\text{next}}(x)$ as a shortcut for u and exit the DFS. Otherwise, we take the existing shortcut and continue with the next node in lexicographic order after $\text{DFS}_{\text{next}}(x)$. Subtrees of close children can be skipped as they already included in the initial value $\sigma_u(u)$.

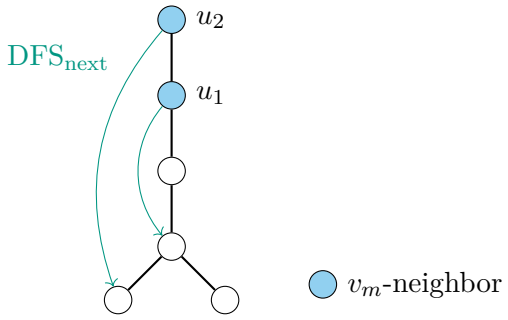


Figure 4.3.: Illustration of DFS_{next}.
Processing u_1 its child closeness gets -1 and DFS_{next} is set.
When proceeding with u_2, this shortcut is taken. But immediately, also $\text{child}_{\text{close}}(u_2)$ gets -1 and DFS_{next} ist set for u_2 respectively.

As another implementation detail we consider how values can be propagated upwards. It takes too much time to iterate over all children to determine c, the child maximizing $\text{score}_{\max}$ among all children, or to calculate $\sigma_u(u)$. Hence, when processing a node u, we already pass information to its parent node p (see Algorithm 4.2, lines 29-33). If $\text{score}_{\max}(u) > \text{score}_{\max}(p)$, the best parent in the subgraph of S_p which got traversed so far, is the best parent in S_u. We set $\text{score}_{\max}(p)$ to $\text{score}_{\max}(u)$ and $\text{p}_{\text{best}}(p)$ to $\text{p}_{\text{best}}(u)$. Further, if u is a close child, we add $\text{child}_{\text{close}}(u)$ to $\text{child}_{\text{close}}(p)$. Thus, when we process p, we know that $\text{child}_{\text{close}}(p)$ is initiated with $\sigma_p(p)$ and $\text{score}_{\max}(p)$ with $\text{score}_{\max}(c)$, where again c is the child maximizing $\text{score}_{\max}$. Like this, it is not necessary to iterate over a vertex' children, because we can use the values resulting from upwards propagation.

In order to make upwards propagation work properly, for every node u, $\text{child}_{\text{close}}(u)$ and $\text{score}_{\max}(u)$ are initialized with 0. After every local move, these values need to get restored. For this purpose, we store which nodes got touched while moving the current node v_m and reset their scores afterwards.

Finally, if there is a position for v_m, leading to fewer edits than isolating it, $p = \text{p}_{\text{best}}(r)$ is the corresponding best parent. We iterate over all touched nodes and filter close children of p. If the resulting position causes fewer edits than v_m's previous place, v_m gets moved below p and close children get adopted. Otherwise v_m is pushed back to its original place and the skeleton is the same as before the move.

Algorithm 4.2: `processNode`(u)

```
1  mark u as touched;
2  score_max(u) ← max over score_max of u-children;
3  child_close(u) ← ∑ over child_close of close u-children;
4  if child_close(u) > score_max(u) then
5      score_max(u) ← child_close(u);
6      p_best(u) ← u;
7  if u is v_m-neighbor then
8      score_max(u) ← score_max(u) + 1;
9      child_close(u) ← child_close(u) + 1
10 else
11     score_max(u) ← score_max(u) − 1;
12     child_close(u) ← child_close(u) − 1;
13 if child_close(u) ≥ 0 and u has children then
14     x ← first child of u;
15     while x ≠ u do
16         if x not touched or child_close(x) < 0 then
17             child_close(u) ← child_close(u) − 1;
18             x ← DFS_next(x);
19             if child_close(u) < 0 then
20                 DFS_next(u) ← x;
21                 break;
22             x ← next node in lexicographic order after x below u;
23         else
24             x ← next node in lexicographic order after T_x below u;
25 if u ≠ r then
26     p ← parent(u);
27     if score_max(u) > 0 or child_close(u) > 0 then
28         insert p in queue;
29     if score_max(u) > score_max(p) then
30         score_max(p) ← score_max(u);
31         p_best(p) ← p_best(u);
32     if child_close(u) ≥ 0 then
33         child_close(p) ← child_close(p) + child_close(u);
```

Algorithm 4.3: `localMove`(v_m)

```
1  foreach v_m-neighbor u do
2      insert u in queue;
3  S ← S ⊚ v_m
4  while queue not empty do
5      u ← next element in queue;
6      processNode(u);    //(Alg. 4.2)
7  foreach node w in touched nodes do
8      if parent(w) = p_best(r) and child_close(w) > 0 then
9          add w to c_adopt;
10     child_close(w) ← 0;
11     score_max(w) ← 0;
12     DFS_next(w) ← w;
13 if position below p_best(r) is better than old position then
14     move v_m below p_best(r) and adopt c_adopt;
15 else
16     move v_m back to old position;
```

4.3. Data Structures

For the implementation of the Quasi-Threshold Mover, specialized data structures are used. In the following section, we introduce the Level Queue for storing the nodes to process and the Dynamic Forest which represents the skeleton related with the current editing.

4.3.1. Level Queue

During a local move we process the nodes ordered by their depth, i.e. we start from the deepest occurring level and then proceed upwards. For this purpose, the vertices are kept in a maximum priority queue using their depth as a key. Initially, this queue is filled with the neighbors of v_m. The only kind of node which gets inserted afterwards, is the parent of the currently processed vertex, and thus it is situated exactly one level higher. The implementation from [BHSW15] makes use of this restriction and proposes a data structure which only sorts the neighbors during initialization. Subsequently it possible, to obtain the next node or to insert a parent in constant time. This so called *Level Queue* consists out of different lists. The first list contains all v_m-neighbors ordered by their depth. Nodes which are encountered later, are stored in two additional lists according to their level. One of them is used to keep the vertices of the currently processed level, the other one contains nodes belonging to the next level. As we only insert the parent of the current node, these lists suffice.
Working off the nodes, at first all elements from the current level list are processed, then we continue with the vertices in the neighbor list which also belong to the current level. Afterwards, there are no more nodes in the this level which need to get processed. Hence, we replace the current level list with the list for the next level and continue with these elements.
If a parent node is to be inserted, we can simply add it to the next level list. We know that all relevant nodes are processed, when both the neighbor list and the list of the current level are empty.
We note that initializing the Level Queue with the neighbors of the current node v_m has a running time in $\mathcal{O}(d \log(d))$ for $d = \deg(v_m)$. A faster approach in implemented in the Bucket Queue introduced in Section 7.1.

4.3.2. Dynamic Forest

In the implementation provided by [BHSW15], the current skeleton is represented by a so called *Dynamic Forest*. For every vertex, this data structure holds pointers to its child nodes and to its parent node. Moreover, a node's position in the parent is kept, which makes it possible to find the next node in lexicographic order during a DFS.
In addition to the Dynamic Forest, an array is used to store the depth of each node in the current skeleton.

4.4. Proof of Correctness

Corollary 4.2. *Let G a graph, $v_m \in V(G)$, H a QTG editing of G, S a fixed skeleton of $Q = G\Delta H$. Algorithm 4.3 determines a best parent for v_m in S.*

Proof. In [Ham20] it is show that Algorithm 4.3 determines $\text{score}_{\max}(r)$ correctly. Further it is stated that $\text{p}_{\text{best}}(r)$ is chosen such that moving v_m below it and adopting close children leads to a minimum number of edits incident to v_m among all possible parents and all possible subsets of children for adoption in S. Hence, $\text{p}_{\text{best}}(r)$ is a best parent for v_m. □

4.5. Proof of Running Time

In Section 9.5.5 of [Ham20] it is shown that Algorithm 4.3 runs in $\mathcal{O}(m \log(\Delta))$ per iteration and in amortized $\mathcal{O}(d \log(d))$ for moving a node v_m with degree d.
To understand the respective analysis, we recall the steps of a local move. At first, the current node v_m gets isolated, then a target position is determined by processing all relevant nodes. Finally, v_m gets moved to this position. Hence it needs to be proven that it is possible, to go through all relevant vertices in the queue and to modify the skeleton within the desired running time.
In [Ham20], it is justified that only $\mathcal{O}(d)$ nodes per move need to be considered and get inserted into the queue. Then it is shown that Algorithm 4.2 processes a node from the queue in amortized constant time. However, the queue needs to be initialized and filled with the v_m-neighbors, what requires $\mathcal{O}(d \log(d))$ time (see Section 4.3.1. This results in the $\log(d)$ factor of the running time for one node and in the factor $\log(\Delta)$ regarding the complexity of one iteration.

Further in the proof it is argued how isolating a node from the skeleton and re-inserting it at the target position is possible in amortized $\mathcal{O}(d)$ time. Both operations involve that the pointers in the Dynamic Forest get adapted accordingly. This requires constant time per node. Additionally, the depth needs to be updated for each descendant of v_m at the position it got isolated from or at the position it gets moved to.
Placing v_m at a new position, every ancestor or descendant which is not a v_m-neighbor causes an insert. Thus v_m can have at most $2 \cdot \deg(v_m)$ ancestors and descendants, because otherwise this new position would cause more edits than isolating v_m. In particular, it follows that there are $\mathcal{O}(d)$ v_m-descendants at the target position. Therewith, updating depth values and hence the whole move is possible in $\mathcal{O}(d)$ time.
The running time for isolating v_m depends on the number of descendants at the current position. But in this case, we cannot apply the same bound as for the target position of the move, because the original position is possibly worse. Instead, we make use of the token method. When we move v_m to a position, we give a token to every ancestor and every descendant and keep one token for v_m itself. Later, these tokes are used to pay for updating the depth values.

The number of initially required tokens depends on the initial editing. We note that, when we isolate all nodes, this requires m edits. Hence, it is assumed that an initial editing does not comprise more edits. It follows that the number of initially required tokens is in $\mathcal{O}(m)$. When we move a node to another position, we generate tokens and pass them to the ancestors and descendants at this new position. Consider the case that v_m is to be isolated from a position, where it has more than $\mathcal{O}(d)$ ancestors and descendants. Then some of these vertices must have been placed there after v_m got moved for the last time. Hence, v_m has received tokens from them to pay for updating the depth values of its descendants at the position it now gets isolated from. According the argument above at most $2 \cdot \deg(v_m)$ tokens will be generated for one move, hence $\mathcal{O}(m)$ tokens per iteration. It follows that also isolating a node is possible in amortized $\mathcal{O}(d)$ time.

At the end of every local move, certain initial values need to get restored for all touched nodes. As the number of nodes which get processed, is linear with regard to the degree of v_m, this step works in $\mathcal{O}(d)$ as well.

Having analyzed all parts of a local move, we see that it has an amortized complexity in $\mathcal{O}(d \log(d))$ per node and that overall, one iteration runs in $\mathcal{O}(m \log(\Delta))$. Later, when we analyze a modified version of the provided algorithm, it is important to remember the time contingent for modifying the skeleton. The operations must be linear with respect to the number of ancestors and descendants at the respective position for the running time guarantees to apply.

Finally, we remark that the analysis assumes the connected case. As we mentioned before, this is achieved by extending the input graph G by a virtual root r which is connected to all vertices. By G' we denote the resulting graph and by m' the number of edges it admits. Compared to the original graph, G' has n more edges, i.e. it holds that $m' = m + n$. So in fact, we have proved a running time in $\mathcal{O}(m') = \mathcal{O}(m + n)$. For most of the graphs this describes the same complexity as $\mathcal{O}(m)$, only for sparse graphs with $m < n$, the $+n$ term must not be neglected.

4.6. Initial Editing

The Quasi-Threshold Mover always requires an initial editing it can optimize. One possibility is a trivial initialization, in which every node is isolated, i.e. directly placed below r. But if the algorithm starts from a more cleverly chosen editing, this can lead to faster convergence and to a better final result. In [BHSW15], a QTG recognition algorithm is modified to construct such an initial editing.
The proposed recognition algorithm tries to find a skeleton S by determining parent(u) for every vertex $u \in V_G$. For this purpose, the nodes are processed ordered decreasingly by degree. We observe that nodes which are placed at a higher level in the skeleton must have a higher degree. Hence, the skeleton gets somehow constructed in top down fashion.
Initially, r is set as the parent of every node. When a vertex $u \in V_G$ is processed, we go through those of its neighbors which have not been processed before. For such a neighbor v, we check, whether parent(v) = parent(u). If this is the case, we set parent(v) = u. In the degree descending order v follows on u, thus it holds that $\deg(u) \geq \deg(v)$. From Definition 2.3 (ii) we can further infer that in a QTG, the neighborhood of v must be a subset of the neighborhood of u. So far in the algorithm, parent(v) hence only got updated, if and only if parent(u) changed likewise. Thus, in a QTG parent(v) = parent(u) must always hold and the recognition algorithm fails as soon as this is not the case. For constructing an editing, we instead try to fix the skeleton with as few edits as possible.

If parent$(v) \neq$ parent(u), we can show that there is an induced subgraph which is forbidden in QTGs according to Definition 2.3 (i), i.e. a P_4 or a C_4. Without loss of generality, we assume that parent(v) got processed before parent(u). Thus, when processing parent(u), the parent of v did not get updated. Hence, the edge $\{\text{parent}(u), v\}$ cannot exist. As we go through the nodes ordered decreasingly by degree, it must hold that deg(parent$(u)) \geq$ deg(u). u has at least two neighbors (parent(u) and v). But v is not a neighbor of parent(u), and thus, parent(u) must have at least one neighbor x which is not a neighbor of parent(u). Depending on whether the edge $\{v, x\}$ exists, the vertices x, parent(u), u and v induce either a P_4 or a C_4, i.e. a subgraph which is forbidden in QTGs.
When constructing an editing, we aim to break this forbidden subgraph. For this purpose, it is possible to either ignore the edge $\{u, v\}$, to set parent(u) to parent(v) or to set parent(v) to parent(u). The algorithm tries to choose an option minimizing the resulting number of edits. More details about the exact proceeding can be found in [BHSW15].

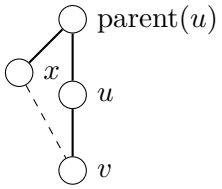


Figure 4.4.: Forbidden subgraph in case parent$(v) \neq$ parent(u)

5. Locally minimal Quasi-Threshold Moving

5.1. Extending the Quasi-Threshold Mover

Whether a node is best parent, depends on the preexisting skeleton. Hence, also the number of edits resulting from Algorithm 4.3 is only minimal among all positions in a fixed skeleton. In the following, we extend the Quasi-Threshold Mover such that simple paths get reordered before the current node v_m is inserted. Subsequently, we show that this to set of edits incident to v_m which admits minimum size.

At first, we focus on the reordering of simple paths. This is possible without affecting the induced QTG as stated in Lemma 3.8. In the following we aim to construct an ordering with a certain property:

Definition 5.1. *Let G a graph, $v_m \in V(G)$, H a QTG editing of G, S a skeleton of $Q = G\Delta H$ and P a simple path in S. We say P admits a **v_m-order** if v_m-neighbors are positioned above non-v_m-neighbors. In case P contains v_m, it is placed between neighbors and non-neighbors. Further, we say S admits a v_m-order if all simple paths in S do so.*

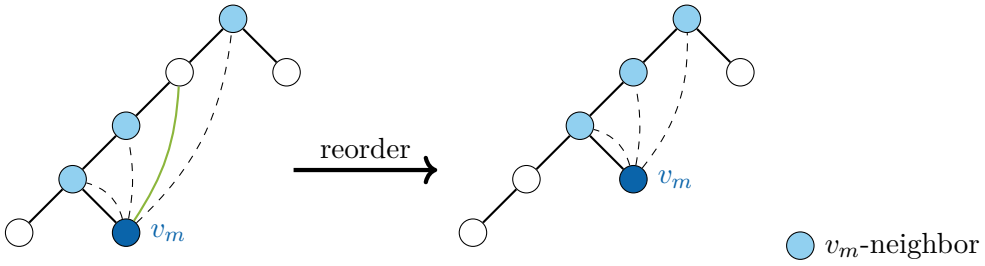


Figure 5.1.: Example for the effect of reordering a simple path:
Without reordering simple paths, at least one edit incident to v_m is required.
When a v_m-ordering is established, no edits are necessary.

The example provided in Figure 5.1 illustrates how the presence of a v_m order can reduce the number of required edits.

Further, we can define a new equivalence relation of skeletons as a refinement of $\cong$ based on the v_m-order of simple paths:

Definition 5.2. *Let S and S' two skeletons. We write $S \equiv_{v_m} S'$ iff $S \cong S'$ and both S and S' admit a v_m-order with regard to the same vertex v_m. S and S' only differ in the order among v_m-neighbors (non-v_m-neighbors) on simple paths.*

In Algorithm 5.1 we go through v_m's neighbors and move them up within their simple path, what results in a skeleton admitting a v_m-order. We claim that inserting v_m in this skeleton analog to Algorithm 4.3 results in an editing minimizing the number of edits incident to v_m. Moreover, we show that the order within v_m-neighbors and non-v_m-neighbors on simple paths has no effect on the minimum number of edits. Hence, it does not make a difference whether we move up a v_m-neighbor to the top of the simple path or just below the last v_m-neighbor that has already been considered.

Algorithm 5.1: `localMove`(v_m)

```
1  foreach v_m-neighbor u do
2      move up u on simple path in S
3      insert u in queue;
4  move up v_m on simple path in S
5  S ← S ⊚ v_m
6  while queue not empty do
7      u ← next element in queue;
8      processNode(u);    //(Alg. 4.2)
9  foreach node w in touched nodes do
10     if parent(w) = p_best(r) and child_close(w) > 0 then
11         add w to c_adopt;
12     child_close(w) ← 0;
13     score_max(w) ← 0;
14     DFS_next(w) ← w;
15 if position below p_best(r) is better than old position then
16     move v_m below p_best(r) and adopt c_adopt;
17 else
18     move v_m back to old position;
```

5.2. Proof of Correctness

In order to prove the correctness of Algorithm 5.1, we begin showing smaller statements that lead us to the final theorem. At first, we observe a connection between the positions considered by the Quasi-Threshold Mover and the $\circledcirc$ operation for isolating a vertex from the skeleton introduced in Definition 3.3.

Lemma 5.3. *Let G a graph and H an editing of G such that $Q = G\Delta H$ is a QTG. For a fixed skeleton S of Q, the position corresponding to $\delta^*_{v_m}(S)$ induces the minimum number of edits incident to v_m over all possible inversions of $S \circledcirc v_m$.*

Proof. In order to invert the isolation with respect to S, it is sufficient to consider every node as a potential parent and every subset of children for adoption. Only close children have more v_m-neighbors than non-v_m-neighbors in their subtrees. Hence for a fixed node as a parent it is optimal to adopt exactly the set of its close children. Overall, a position minimizing $\delta_{v_m}(S, w)$ among all vertices $w \in G$ is an optimum among all positions from which v_m can get isolated. □

Next, we consider simple paths and the distinct positions where best parents can be found:

Lemma 5.4. *Consider a graph G, $v_m \in V(G)$ and a QTG editing H of G. Let S a skeleton with a v_m-order inducing $Q = G\Delta H$, u a best parent regarding S and P the simple path of S on which u is located. With $\mathrm{p_{low}}$ denote the lowermost vertex on P, with $\mathrm{sum_{close}}$ the sum of the child closeness of $\mathrm{p_{low}}$'s close children and with $\mathrm{c_{non}}$ the number of non-v_m-neighbors on P. Then it holds that:*

(i) If $\mathrm{sum_{close}} \geq \mathrm{c_{non}}$, $\mathrm{p_{low}}$ is a best parent

(ii) Otherwise u is the lowermost v_m-neighbor on P

Proof. Case 1: $\mathrm{sum_{close}} \geq \mathrm{c_{non}}$
When we adopt $\mathrm{p_{low}}$'s close children, we save at least as many edits as we need to insert the edges between v_m and non-v_m-neighbors on P. At an optimal position in S v_m hence has ancestor-descendant-relationships to all ancestors of P, to all nodes on P, to all close children of $\mathrm{p_{low}}$ and to their descendants. This can be achieved by moving v_m below $\mathrm{p_{low}}$ and adopting close children, hence $\mathrm{p_{low}}$ is a best parent. If all children of $\mathrm{p_{low}}$ are close, moving v_m below any vertex on P and adopting its only child leads to the same ancestor-descendant-relationships and thus to the same edits. This is why $u = \mathrm{p_{low}}$ does not need to hold necessarily.

Case 2: $\mathrm{sum_{close}} < \mathrm{c_{non}}$
Adopting any subset of children of $\mathrm{p_{low}}$ does not save enough edits to compensate the required insertion of edges between v_m and non-v_m-neighbors on P. Moving v_m below any vertex on P but $\mathrm{p_{low}}$ and adopting this vertex' only child again leads to the same edits as moving v_m below $\mathrm{p_{low}}$ and adopting all children. Thus in this case, v_m cannot have any children at its optimal position, i.e. u cannot have close children. The number of edits incident to v_m is therefore minimal by placing the vertex so that it has an ancestor-descendant-relationship to all v_m-neighbors and to none of the non-v_m-neighbors on P. Note that there must be a v_m-neighbor on P. Otherwise it is not optimal to place v_m so that it has an ancestor-descendant-relationship to any node on P, i.e. the best parent cannot be located on P. Hence, u must be the lowermost v_m-neighbor on P.

□

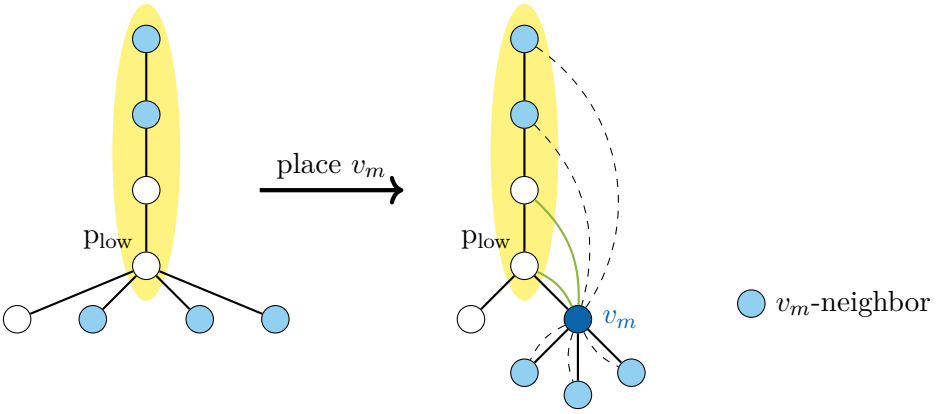


Figure 5.2.: Illustration of *Case (i)*, inserted edges drawn in green

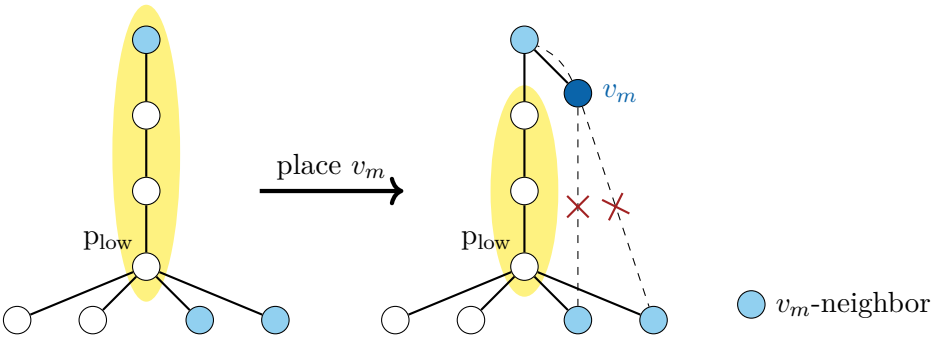


Figure 5.3.: Illustration of *Case (ii)*

The following Lemma is concerned with the effect of different orders of v_m-neighbors and non-v_m-neighbors on simple paths. Especially it is shown that minimum number of edits incident to v_m is independent of these orders.

Lemma 5.5. *Consider a graph G, $v_m \in V(G)$ and a QTG editing H of G. Let S and S' skeletons both inducing $Q = G \Delta H$ such that $S' \equiv_{v_m} S$. Let further P as simple path of S and P' its permutation in S'. Pick $i \in \{1, 2, \ldots, |V_P|\}$, let u the i-th vertex on P from the top and u' on P' respectively. Then it holds that $\delta_{v_m}(S, u) = \delta_{v_m}(S', u')$.*

Proof. At first observe that reordering simple paths can change the children of the lowermost vertex of a path but does not affect the sets of vertices which are covered by the subtrees rooted in these children. Therefore the child closeness of the lowermost vertex is the same and also the same vertices are part of a subtree rooted in a close child. Hence if $i = V_P$, adopting close children causes the same number of edits.
Now we consider the case that u and u' are inner vertices of their simple path and therefore each of them has exactly one child. Given that the lowermost vertices of P and P' have the same child closeness, we can walk upwards the path and for each vertex the child closeness is increased if it is a v_m-neighbor or decreased otherwise. As v_m-neighbors are above all non-v_m-neighbors, vertices on the same position of different v_m-orders of the same simple path always admit the same child closeness. In particular, u''s child is close if and only if u's child is close. In case of adoption, edges need to be inserted to all non-neighboring descendants. The number of non-v_m-neighbors on P' below u' is the same as below u on P and also the descendants below the simple path are the same. Overall the same number of insertions is required. If the children of u und u' are not close, the analog argumentation can be applied for the deletion of edges to neighboring vertices, so that in every the same number of edits concerning descendants is required.

Apart from that, moving v_m below u makes it necessary to insert edges to non-neighboring ancestors and to delete edges to neighbors on other branches. Above u in P and above u' in P' there is the same number of non-v_m-neighbors. Additionally, we obtain the same ancestors outside the simple path in both skeletons, what results in the same number of required insertions. Also the set of edges whose deletion is necessary does not get influenced by the reordering of simple paths.
Thus, the number of edits incident to v_m is the same, i.e. $\delta_{v_m}(S, u) = \delta_{v_m}(S', u')$. □

Corollary 5.6. *Let G a graph, $v_m \in V(G)$, S and S' as above. Then it holds that $\delta^*_{v_m}(S) = \delta^*_{v_m}(S')$.*

Proof. Let u a best parent in S. We can deduce from Lemma 5.5 that there is a vertex $u' \in V_G$ such that $\delta_{v_m}(S', u') = \delta_{v_m}(S, u)$. Hence $\delta^*_{v_m}(S') \leq \delta^*_{v_m}(S)$. Now assume there is a vertex $u_{\text{better}} \in V_G$ such that $\delta_{v_m}(S', u_{\text{better}}) < \delta_{v_m}(S', u')$. Then again applying Lemma 5.5 provides a vertex u^* such that $\delta_{v_m}(S, u^*) = \delta_{v_m}(S', u_{\text{better}})$. But then it also holds that $\delta_{v_m}(S, u^*) < \delta_{v_m}(S, u)$, what contradicts the fact that u is best parent in S. Hence, no better parents can arise from the reordering of simple paths and it follows that $\delta^*_{v_m}(S) = \delta^*_{v_m}(S')$. □

With the help of previously shown statements, we can now prove the major theorem, providing the correctness of Algorithm 5.1.

Theorem 5.7. *Algorithm 5.1 determines an editing such the number of edits incident to v_m is minimal.*

Proof. Let G a graph, I an arbitrary QTG editing of G and H the editing constructed from I by isolating v_m, i.e. $Q = G\Delta H = ((G\Delta I)\,\text{iso}\,v_m)$.
Let S_{sort} a skeleton of Q admitting a v_m-order and let H_{alg} the editing Algorithm 4.3 constructs based on S_{sort}. From Corollary 4.2 we know that this results in $\delta^* v_m(S_{\text{sort}})$ edits incident to v_m.
Further we consider an editing $H_{\min}$ in which the number $\delta_{\min}$ of edits concerning v_m is minimal, but the edits affecting edges not incident to v_m are the same as in H. Our aim is to show now that $\delta^*_{v_m}(S_{\text{sort}}) = \delta_{\min}$.
In the following we pick a fixed skeleton $S_{\min}$ of $Q_{\min} = G\Delta H_{\min}$ in which all simple paths admit a v_m-order. Then we construct Q_{iso} as induced QTG of $S_{\text{iso}} = S \circledcirc v_m$. It can be observed that $Q_{\min}$ is obtained from Q by adding a set of edges incident to v_m. By isolating v_m these edges get removed again, hence it holds that $Q_{\text{iso}} = Q$.
Next we want to show that $S_{\text{iso}} \equiv_{v_m} S_{\text{sort}}$. As $ind(S_{\text{sort}}) = Q = Q_{\text{iso}} = \text{ind}(S_{\text{iso}})$ we know from Lemma 3.8 that $S_{\text{iso}} \cong S_{\text{sort}}$.
Most of the simple paths of S_{iso} inherit the v_m-order from $S_{\min}$ by construction. We only need to consider the simple paths which are affected by the isolation of v_m. If v_m is located on a non-trivial simple path in $S_{\min}$ the order on that path will be maintained in S_{iso} as v_m is placed between neighbors and non-neighbors in the v_m-order. If v_m has no children and exactly one sibling in $S_{\min}$, the simple paths of v_m's parent u and its sibling will be united to a single simple path P in S_{iso}. u must be the lowermost vertex on its simple path and from Lemma 5.4 we know that it is a v_m-neighbor as well. Thus we can infer that also P has a v_m-order.
Overall in both S_{sort} and S_{iso} all simple paths have an v_m-order, thus $S_{\text{sort}} \equiv_{v_m} S_{\text{iso}}$. Now we can apply Lemma 5.6 to deduce that $\delta^*_{v_m}(S_{\text{sort}}) = \delta^*_{v_m}(S_{\text{iso}})$. Moreover with Lemma 5.3 it follows that $\delta_{\min} = \delta^*_{v_m}(S_{\text{iso}})$. Hence, $\delta^*_{v_m}(S_{\text{sort}}) = \delta_{\min}$, i.e. the algorithm moves v_m to a position such that it is incident to a minimum number of edits. □

6. Inclusion-minimal Quasi-Threshold Editing

In the following section we introduce an algorithm which constructs an inclusion-minimal editing. Subsequently we prove its correctness.

6.1. The Algorithm

Algorithm 5.1 can not only be applied to improve a provided editing but also to initially construct an inclusion-minimal editing. For this purpose, we modify the algorithm such that the vertices get inserted one by one. In every step, the current vertex is then placed at the optimal position regarding the vertices inserted before.

Algorithm 6.1: Constructing an inclusion-minimal editing

```
1 Q ← empty graph
2 for v_m ∈ V_G in given order do
3 |   Q ← Q ∪ {v_m} ∪ {{v_m, w} | w ∈ V_Q}
4 |   localMove(v_m)    //(Alg. 5.1)
```

6.2. Proof of Correctness

The following Lemma justifies how a series of insertions which are minimal as reasoned above can overall lead to a inclusion-minimal editing. A corresponding statement was previously shown for other graph classes, for example in [KF81].

Lemma 6.1. *Let G be an arbitrary graph and let H be a minimal QTG editing(resp. completion or deletion) of G. Consider a new graph $G' = G + x$, obtained by adding to G a new vertex x adjacent to an arbitrary set $N(x)$ of vertices of G. There is a minimal QTG editing (resp. completion or deletion) H' of G' such that $H' - x = H$.*

Proof. Consider an editing $H^* = H + I$, where I is a minimal set of edits such that $G'\Delta H^*$ is a QTG.
We first show that it is possible to find such a set I of edits augmenting H without changing existing edits. For this purpose, let $Q = G\Delta H$, which is a QTG by construction. By adding x as an isolated (universal) vertex to Q we again obtain a QTG. Thus adding the edits for removing (inserting) all edges $x \sim G$ from G' to H yields an editing transforming G' into a QTG. This shows that a set of edits complementing H to a proper editing for G' respective deletion (completion) exists. Pick I inclusion minimal among these sets.
Now we want to prove that H^* is inclusion minimal. For this purpose we assume the contrary, i.e. that H^* contains superfluous edits. These edits cannot only concern edges incident to x, as I is chosen minimal. Thus, if $H_{\min}$ is the inclusion minimal (strict) subset of H^*, $H \setminus H_{min}$ is not empty $(*)$. We know that $Q' = G'\Delta H_{\min}$ is a QTG, i.e. it does not contain a P_4 or a C_4 as an induced subgraph (see Definition 2.3(i)). Consequentially, also $Q' - x$ does not admit such a forbidden subgraph and hence is a QTG as well. Thus G got transformed into a QTG only using the edits in $H_{\min}$ which concern edges having both endpoints in G. From $(*)$ we know that these edits are a strict subset of H. This contradicts the prerequisite that H is a minimal QTG editing of G. □

Finally we can now deduce the theorem about the correctness of Algorithm 6.1.

Theorem 6.2. *Algorithm 6.1 determines an inclusion-minimal editing.*

Proof. According to Theorem 5.7 in every step a position with a minimal number of edits is determined and from Lemma 6.1 it follows that this overall constructs an inclusion-minimal editing. □

Constructing an inclusion minimal QTG editing, nodes get inserted in a certain order. Instead of using a random node sequence, one can sort the vertices in ascending order according to their degree. This approach is oriented towards the recursive Definition 2.1 of the quasi-threshold graph class and we try to build the QTG in bottom up style. Effects of the insertion order are examined in Section 10.2.

Further, we note that inclusion-minimality guarantees that an editing of size 0 is detected if it exists. Hence, the algorithm can also be used for QTG recognition.

7. Data Structures

In Section 4.3, we have already explained the Level Queue and the Dynamic Forest, two data structures used in the implementation provided by Brandes et al. In the following, we present the Bucket Queue which fulfills the same function as the Level Queue but admits an improved running time. Additionally, we show how the Dynamic Forest is extended for the management of simple paths.

7.1. Bucket Queue

In Algorithm 4.3 as well as in the improved version we need a priority queue to keep the relevant nodes when executing a local move for a node v_m. As only specific operations are required, it is possible to construct a priority queue that uses buckets and works in amortized linear time.

7.1.1. The Data Structure

The Bucket Queue consists of two arrays of size n. The first array stores the nodes in ascending order regarding their depth in the skeleton. The second array holds pointers to the bucket borders, i.e. border$[i]$ points on the first node of depth $\geq i$ in the node array. During one local move every vertex gets inserted at most once into the queue, hence length n is adequate for the node array. As moreover depth$(u) < n$ for every node u in the skeleton, an array of size n also suffices to store the borders.
Furthermore the Bucket Queue stores a pointer $\text{node}_{\text{next}}$ to the element in nodes which will be removed next, and another pointer $\text{bucket}_{\text{cur}}$, indicating the bucket the last removed node has belonged to.

7.1.2. Operations

The interface of the Bucket Queue offers the following functions:

- `fill` to initialize the Bucket Queue with the list of v_m-neighbors
- `next` to process the next node from the queue
- `insertParent` to insert a node's parent into the queue
- `empty` to check whether the queue is empty

7.1.2.1. `fill`

With the help of the `fill` operation we initialize the Bucket Queue with all relevant v_m neighbors. We make use of counting sort to order these vertices by their depth and to determine the respective bucket borders. For this purpose, the border array is at first used to count the number of occurrences of each depth value among the v_m-neighbors of depth $\leq 2 \cdot \deg(v_m)$. Then the prefix sum is calculated on that array. Subsequently we iterate over the v_m-neighbors in reversed order to preserve stability. Inserting a node u into the Bucket Queue, we decrease border[depth(u)] by 1 and set nodes[border[depth(u)]] = u. Again nodes of depth $> 2 \cdot \deg(v_m)$ are skipped.
Having inserted all relevant neighbors, also the pointer border array are proper. We set $\text{node}_{\text{next}}$ to point on the last element in nodes and initiate $\text{bucket}_{\text{cur}}$ so that it points behind the used part of the border array.
Let l the number of elements which get inserted into the queue and let k the number of buckets. The running time of the `fill` operation is dominated by counting sort running in $\mathcal{O}(l + k)$. As we only insert nodes of depth $\leq 2 \cdot \deg(v_m)$ into the queue, it holds that k is asymptotically equivalent to $\deg(v_m)$, i.e. $\mathcal{O}(k) = \mathcal{O}(\deg(v_m))$. Moreover $l \leq \deg(v_m)$, hence the running time of the `fill` operation is linear with respect to $\deg(v_m)$.

Algorithm 7.1: `fill`

```
1  depth_max ← min(n − 1, 2 · deg(v_m));
2  border ← [0, 0, ..., 0]   (depth_max + 1 entries);
3  foreach v_m-neighbor u do
4      if depth(u) > depth_max then
5          continue;
6      border[depth(u)] ← border[depth(u)] + 1;
7  calculate prefix sum on border
8  bucket_cur ← depth_max;
9  node_next ← none;
10 foreach v_m-neighbor u in reversed order do
11     if depth(u) > depth_max then
12         continue;
13     border[depth(u)] ← border[depth(u)] − 1;
14     nodes[border[depth(u)]] ← u;
15     node_next ← node_next +1;
```

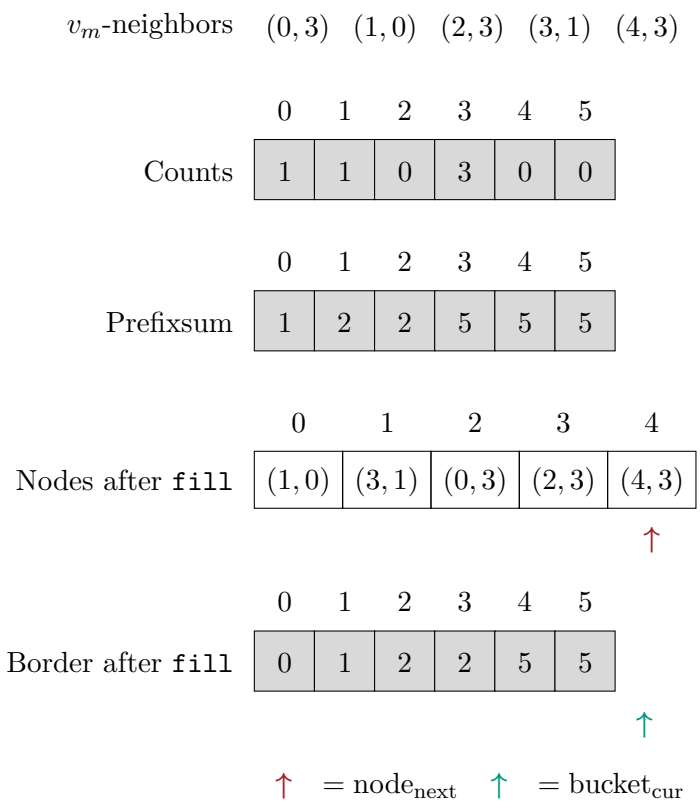


Figure 7.1.: Example for `fill`, nodes given as pairs (id, depth)

7.1.2.2. `next`

To obtain the next element from the queue, we check at first, whether the queue is empty. In this case we return none, otherwise we return nodes[node$_{\text{next}}$]. Furthermore bucket$_{\text{cur}}$ gets updated if necessary and finally node$_{\text{next}}$ is decremented. In order to update bucket$_{\text{cur}}$ we skip all empty buckets between the previously removed node and the node removed in this call. Basically this operation has a running time in $\mathcal{O}(k)$. But we note that during one local move, bucket$_{\text{cur}}$ decreases monotonously, i.e. no bucket gets skipped more than once. Hence, `next` has amortized constant running time.

Algorithm 7.2: `next`

```
1 if empty then
2     return none;
3 result ← nodes[node_next];
4 while node_next < border[bucket_cur] do
5     bucket_cur ← bucket_cur − 1;
6 node_next ← node_next − 1;
7 return result;
```

7.1.2.3. `insertParent`

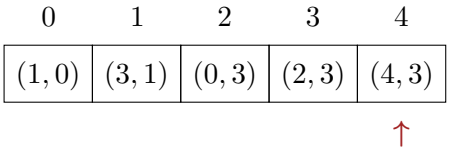
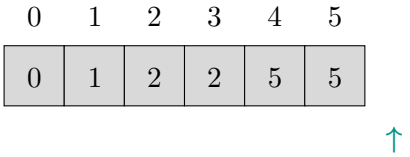
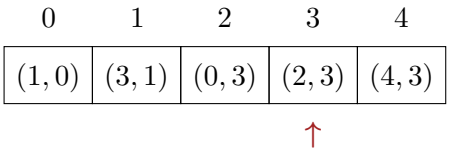
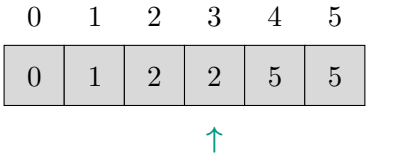
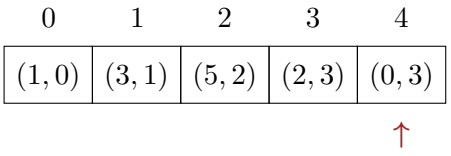
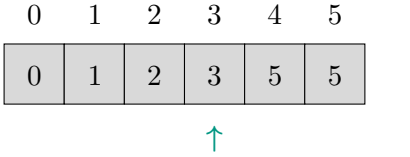
Here, the generic insert operation for priority queues tailored to the specific use case. We know that the only kind of element which gets inserted after the initialization, is the parent p of the previously removed node u. Hence, we know that it needs to be placed in the subsequent bucket. However, we need to make room for p in the nodes array. Shifting all the vertices of the current bucket is too time-consuming. But we can be sure that there is a free place at the end of the current bucket, because this is exactly the position u just got removed from. Like this we obtain an empty place which is used for p. At last, border array gets updated accordingly (see Figure 7.1). With the help of the stored pointers this can be realized in constant time. We can make use of the $\text{bucket}_{\text{cur}}$ pointer to access the first element f of the bucket u belonged to. Then we put f at the end of the bucket and increment $\text{node}_{\text{next}}$, so that it points on f. Now, there is a free place at the old position of f, where p can get inserted. Finally we need to update the border of $\text{bucket}_{\text{cur}}$ as it got shifted one position upwards.

Algorithm 7.3: `insertParent`

1 $f \leftarrow \text{nodes}[\text{border}[\text{bucket}_{\text{cur}}]]$;
2 $\text{node}_{\text{next}} \leftarrow \text{node}_{\text{next}} + 1$;
3 $\text{nodes}[\text{node}_{\text{next}}] \leftarrow f$;
4 $\text{nodes}[\text{border}[\text{bucket}_{\text{cur}}]] \leftarrow p$;
5 $\text{border}[\text{bucket}_{\text{cur}}] \leftarrow \text{border}[\text{bucket}_{\text{cur}}] + 1$;

7.1.2.4. `empty`

The queue is empty if and only if $\text{node}_{\text{next}}$ is none. This can be checked in constant time.

Nodes	Border
	
`next()`	
	
	`return (4,3)`
`insertParent()` $p = (5, 2)$	
	

↑ = $\text{node}_{\text{next}}$ ↑ = $\text{bucket}_{\text{cur}}$

Table 7.1.: Exemplary usage of the Bucket Queue

7.1.3. Proof of Correctness

In order to prove the correctness of the Bucket Queue it is the non-trivial part to argue, why nodes of depth $> 2 \cdot \deg(v_m)$ do not need to be inserted into the queue.

Lemma 7.1. *Moving a node v_m, neighbors of* depth $> 2 \cdot \deg(v_m)$ *do not need to be considered.*

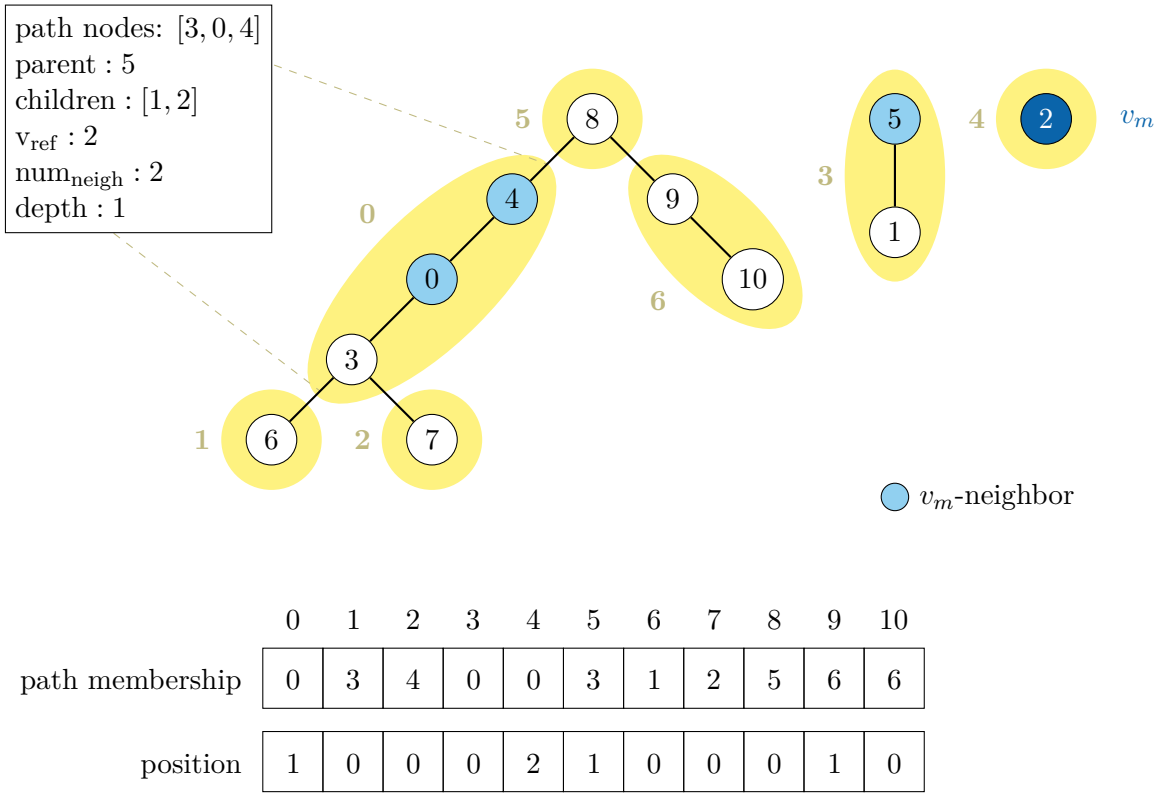
Proof. Denote $2 \cdot \deg(v_m)$ with $\text{depth}_{\max}$.
To show that skipping neighbors of depth $> \text{depth}_{\max}$ does not affect the correctness of the algorithm, we note at first that from a v_m-neighbor w with $\text{depth}(w) > \text{depth}_{\max}$ there is a path P from w to the component's root which contains more than $deg(v_m)$ non-v_m-neighbors. Thus, whenever keeping the edge (v_m, w), this requires at least the insertion of edges to all non-neighboring ancestors, i.e. $> \deg(v_m)$ edits, what is worse than isolating v_m using $\deg(v_m)$ edits.
It follows directly that w is not supposed to be chosen as best parent.
Moreover any vertex in which a subtree reaching deeper than $\text{depth}_{\max}$ is rooted, can never become a close child, because adopting this vertex requires more edits than isolating v_m. Thus we can set the child closeness of a vertex to -1 if we encounter any vertex of depth $> \text{depth}_{\max}$ in its subtree. □

7.2. Dynamic Forest with Simple Paths

QTM is initialized with an editing which gets adapted during local moving. Each editing is related with the skeleton inducing the corresponding QTG. Hence, a data structure is required which represents the current skeleton and can be updated according to changes in the editing. We adapt the Dynamic Forest explained Section 4.3.2, such that it additionally manages simple paths. The resulting data structure is introduced in the following section.

7.2.1. The Data Structure

The basic idea behind our data structure is to construct a forest, as we would obtain it when contracting all simple paths. Hence, the tree nodes are *simple path nodes* (SP nodes), each representing a simple path. Note that in the trivial case, this path consists out of a single vertex. Let P a simple path in the given skeleton, by u we denote the uppermost, by v the lowermost vertex in P. In the Dynamic Forest, the children of P's SP-node are the SP-nodes representing the simple paths of v's children. Respectively, the SP-node of u's parent in the skeleton is the parent of the SP-node of P in the Dynamic Forest.
A SP node stores the graph nodes it contains. Note that they are placed in a vector beginning with the lowermost one, this plays an important role to ensure certain running time guarantees. SP nodes also hold pointers to their parent and child simple path nodes and store the depth of the uppermost node in the path. Furthermore, to sort the path as desired, a pointer $\mathrm{v_{ref}}$ to v_m and $\mathrm{num_{neigh}}$, the current number of v_m-neighbors in the path are kept.
For every graph node the Dynamic Forest stores the SP node it belongs to and its position within that node. Therefrom information like the node's depth, parent or children can be inferred in constant time.



	0	1	2	3	4	5	6	7	8	9	10
path membership	0	3	4	0	0	3	1	2	5	6	6
position	1	0	0	0	2	1	0	0	0	1	0

Figure 7.2.: Exemplary Dynamic Forest after isolation of v_m

In order to represent simple paths properly in the Dynamic Forest, we `unify` or `split` SP nodes in order to keep up correctness and length maximality.
When we unify two simple paths, we move the nodes of the upper path to the SP node of the lower one. As the graph nodes in the path are stored from bottom to top, this requires linear time with respect to the length of the upper path. Further, the parent pointer needs to get updated, this is an operation of constant effort.
When we split a path, we move the graph nodes of the upper part into a new SP node. Again we make use of the order in which the graph nodes in the path are stored. Doing so, we obtain linear running time regarding the number of nodes above the split point. Again updating pointers can be done in constant time.
Some modifications of the skeleton make it moreover necessary to update the depth of some SP nodes. If the depth of a node changes, this affects the depth of all SP nodes in the respective subtree. Hence, within in this subtree a DFS can be executed to keep the depth values up-to-date. This has a linear time complexity with respect to the number of nodes in the subtree.

Algorithm 7.4: `unify`$(\mathrm{P_{up}}, \mathrm{P_{low}})$

```
1 if P_up = P_low then
2     return
3 foreach node u in P_up do
4     add u to P_low
5 move P_low below parent(P_up)
6 delete P_up
```

Algorithm 7.5: `split`$(\mathrm{P}, \mathrm{pos})$

```
1 if length(P) < 2 or pos = 0 then
2     return
3 P_new ← new Path
4 for u in P above pos do
5     add u to P_new
6     delete u from P
7 move P_new below parent(P)
8 move P below P_new
```

7.2.2. Construction

The Dynamic Forest is constructed in linear time provided $\mathrm{p}_{\mathrm{init}}$, an array specifying the initial parent for every vertex. At first every vertex is placed in its own trivial simple path node and parent and child pointers are set with respect to the input, which can be achieved in $\mathcal{O}(n)$. Then the simple path nodes get united such that each of them represents a simple path of maximal length. Finally, depth values are determined. The two last steps are both realized with DFS instances starting from every root, which also takes linear time.

Algorithm 7.6: Construction of Dynamic Forest from $\mathrm{p}_{\mathrm{init}}$

```
1  for u ← 0 to n do
2      path(u) ← new Path
3  for u ← 0 to n do
4      P ← path(u)
5      p ← p_init(u)
6      if p = none then
7          place P as root
8      else
9          P_par ← path(p)
10         move P below P_par
11 foreach SP node P in lexicographical order do
12     P_par ← parent(P)
13     if P_par ≠ none and children(P_par) = {P} then
14         unify(P_par, P)
15 set depth for SP nodes in lexicographical order
```

7.2.3. Modification

To modify the Dynamic Forest, the interface offers the following functions:

- `isolate`(v_m) to isolate v_m from the forest
- `moveUpNeighbor`(u, v_m) to move up a v_m-neighbor u within its simple path
- `moveToPosition`($v_m, p, \mathrm{c}_{\mathrm{adopt}}$) to move v_m below a node p and adopt a subset of children $\mathrm{c}_{\mathrm{adopt}}$

All these operations involve updating the simple paths by respective use of `unify` and `split`.

7.2.3.1. `moveUpNeighbor`

This operation aims to move a v_m-neighbor at a position in its simple path such that only other v_m-neighbors are placed above it. Like this, only the respective SP node itself is affected. For every SP node we store in $\mathrm{num}_{\mathrm{neigh}}$ the number of v_m-neighbors we have already moved up, hence we can simply swap u with the first node f not considered as v_m-neighbor so far and increase the $\mathrm{num}_{\mathrm{neigh}}$ afterwards. If the node v_m passed as an argument differs from the reference node $\mathrm{v}_{\mathrm{ref}}$ stored in the SP node, we need to update this reference and reset $\mathrm{num}_{\mathrm{neigh}}$ to zero, as the SP node is touched for the first time during the local move of v_m. Each step and hence the complete operation admits a constant running time.

Algorithm 7.7: `moveUpNeighbor`(u, v_m)

```
1 P ← path(u)
2 if v_ref(P) ≠ v_m then
3 |   v_ref(P) ← v_m
4 |   num_neigh(P) ← 0
5 f ← node at position num_neigh on P
6 swap f and u
7 num_neigh(P) ← num_neigh(P) + 1
```

7.2.3.2. `isolate`

When we isolate a vertex v_m from the forest, two cases need to be regarded. Either, the vertex is part of a simple path together with other vertices or it is only part of a trivial simple path. In the latter case, the complete simple path node gets isolated. If it had exactly one sibling, this sibling SP node needs to get unified with its parent. Further, depth needs to get updated below each of v_m's former children.
If v_m needs to get removed from a longer simple path, this requires shifting the remaining nodes. Remember that the member nodes of a simple path are stored starting from its lower end. Hence, to isolate v_m, we need to touch all vertices placed above it in its simple path. Then v_m is put into a new trivial and isolated simple path node. Because its former simple path is shorter than before, depth gets updated in the subtree below.
As discussed in Section 4.5, we need to ensure that the complexity of the `isolate` operation is linear with respect to the number of ancestors and descendants at the position, the node gets removed from.

At first, we show that the isolation itself, without updating the depth values, requires linear time regarding the number of ancestors. If the whole SP node gets isolated, the worst case running time is determined by the `unify` operation and hence is linear regarding the size of v_m's parent path. We note that this path cannot contain more nodes than v_m has ancestors. If v_m gets isolated from a longer simple path, shifting the nodes is the most time consuming step. The related costs are linear with respect to the number of nodes above v_m in its original simple path. Also this number is bounded by the ancestor count. Hence, in both cases the complexity is linear regarding the number of ancestors of v_m at the original place.
The isolation operation also comprises updating depth values in the subtree below v_m's previous position. With the help of a DFS this can be realized in linear time regarding the descendant count. Overall, the desired bound for the complexity of `isolate` isolate holds, i.e. it is linear with respect to the number of ancestors and descendants.

Algorithm 7.8: `isolate`(v_m)

```
1  P ← path(v_m)
2  if length(P) = 1 then
3      P_par ← parent(P)
4      foreach P_child in children(P) do
5          move P_child below P_par
6          update depth below P_child
7      if children(P_par) = {P_c0} then
8          unify(P_par, P_c0)
9      place P as root
10 else
11     foreach node u in P above v_m do
12         move u one position to the front
13     P_new ← new Path
14     add v_m to P_new
15     place P_new as root
16     update depth below P
```

7.2.3.3. `moveToPosition`

The third modification for the Dynamic Forest, moving v_m below a node p and adopting a subset c_{adopt} of p's children, can require different adaptations of the SP nodes. If all children are adopted, we can add v_m anywhere to p's path. We decide to add it on the top of the path, as this is possible in constant time by appending it to list of graph nodes stored in p's SP node. If at least one child is not adopted, we need to split the path of p. Further, if v_m adopts exactly one child, we can add it to the path of that child (again on the the top, due to the argument from above). Otherwise, v_m is only part of a trivial simple path, i.e. has its own SP node which gets inserted into the tree structure by updating respective child and parent pointers. In either case, finally the depth values in the subtree rooted in v_m's target SP-node need to get updated.

To maintain the running time guarantees introduced in Section 4.5 the complexity of `moveToPosition` must be linear with respect to the number of ancestors and descendants of v_m at the target position. The worst case running time of the operation occurs, when the parent path needs to get split and a subset of children is adopted. The costs for the `split` operation linear regarding the number of ancestors v_m has at its new position. Adopting children and updating depth values requires linear time with respect to the number of resulting v_m-descendants. Overall, the desired bound holds.

Algorithm 7.9: `moveToPosition`$(v_m, p, \mathrm{c_{adopt}})$

```
1  P ← path(v_m)
2  P_par ← path(p)
3  if c_adopt = children(p) then
4  |  add v_m to P_par
5  |  delete P
6  else
7  |  split(P_par, pos(p))
8  |  if c_adopt = [c_0] then
9  |  |  add v_m to path(c_0)
10 |  |  delete P
11 |  else
12 |  |  move P below P_par
13 |  |  foreach node c in c_adopt do
14 |  |  |  move path(c) below P
15 update depth below path(v_m)
```

8. Proof of Running Time

For an implementation of Algorithm 5.1 we make use of the Bucket Queue and the Dynamic Forest. The resulting pseudo code can be found in Algorithm 8.1. We state that this algorithm runs in $\mathcal{O}(m)$ per iteration and that moving a node v_m has an amortized complexity in $\mathcal{O}(\deg(v_m))$. To prove our claim, we proceed analogical to the running time analysis for Algorithm 4.3 in Section 4.5. We consider the modifications we made on the algorithm and investigate their effect on the complexity.

During a local move, a new position for the current node v_m is determined. For justifying the claimed running time, we need to show that working off the queue with all relevant nodes is possible in $\mathcal{O}(\deg(v_m))$.
In Section 9.5.5 of [Ham20] it is stated that the number of vertices which get inserted into the queue is linear with respect to $\deg(v_m)$ and further, that `processNode` (Algorithm 4.2) runs in amortized constant time. We did not change how a node gets processed and none of our modifications affects the decision, whether a node is put into the queue or not. Hence, we can retain the provided guarantees.
Algorithm 4.3 runs in $\mathcal{O}(m \log(\Delta))$ per iteration and in amortized $\mathcal{O}(d \log(d))$ for moving a node v_m with $\deg(v_m) = d$. Here, the log-factors result from using the Level Queue, but we can get rid of them with the help of the Bucket Queue. It takes $\mathcal{O}(d)$ to initiate this data structure with the neighbors of v_m, all other required operations are possible in amortized constant time (see Section 7.1). Like this, working off the whole queue is possible in $\mathcal{O}(\deg(v_m))$.

A local move also comprises isolating v_m from the skeleton and moving it to a new position. In Section 4.5 it is argued that this is possible in amortized $\mathcal{O}(\deg(v_m))$, if the skeleton can be modified in linear time regarding the number of ancestors and descendants of v_m at the respective position. In the Dynamic Forest, the operations `isolate` and `moveToPosition` admit the required complexity, as we justify Section 7.2.3. Hence, also this parts of the local move are possible in amortized $\mathcal{O}(\deg(v_m))$ per node and in $\mathcal{O}(m)$ for one iteration. In our version of the algorithm, additionally, simple paths get reordered. According to Section 7.2.3, moving up one v_m-neighbor within a simple path is a constant operation, thus a v_m-order can be constructed in $\mathcal{O}(\deg(v_m))$ by moving up all v_m-neighbors.

Having considered all steps of a local move we obtain a running time in amortized $\mathcal{O}(\deg(v_m))$ per node and and in $\mathcal{O}(m)$ per iteration as we have claimed it above. Obviously, this bounds also hold for constructing an inclusion-minimal initial editing by insertion according to Algorithm 6.1. If a graph is a QTG, i.e. admits an editing for size 0, it is detected by this algorithm. Hence, it can also be used for linear-time QTG recognition.

Algorithm 8.1: `localMove`(v_m)

```
1  p_cur ← parent(v_m);
2  c_cur ← children(v_m);
3  foreach v_m-neighbor u do
4      moveUpNeighbor(u, v_m);
5  moveUpNeighbor(v_m, v_m);
6  isolate(v_m);
7  queue.fill(v_m-neighbors);
8  while !queue.empty() do
9      u ← queue.next();
10     processNode(u);    //(Alg. 4.2)
11 foreach node w in touched nodes do
12     if parent(w) = p_best(r) and child_close(w) > 0 then
13         add w to c_adopt;
14     child_close(w) ← 0;
15     score_max(w) ← 0;
16     DFS_next(w) ← w;
17 if position below p_best(r) is better than old position then
18     moveToPosition(v_m, p_best(r), c_adopt);
19 else
20     moveToPosition(v_m, p_cur, c_cur);
```

9. Random Decisions

It is a possible weakness of the local mover algorithm that it can get stuck in a local minimum. In such a situation it is beneficial to provide for more node movement and to alternate between different editings even if they admit the same size. However, with regard to a different skeleton, new opportunities for reducing the number of edits can arise. We put this idea into practice by randomizing certain decisions if the options lead to an editing of the same size. This concerns the choice whether a child of closeness 0 gets adopted. Further, if there exist two or more equally good best parents in a local move, we want to choose randomly, below which one v_m gets placed. To get a chance to escape local minima, we do not terminate the algorithm in the first iteration without an improvement but only when there is a plateau which exceeds a certain size.

9.1. Modification of the Quasi-Threshold Mover

In the following we discuss how Algorithm 4.2 and Algorithm 5.1 are modified in order to randomize decisions in the situation described above. The related implementations are depicted in Algorithm 9.2 and Algorithm 9.1.
Adopting children with closeness 0 does not affect the number of required edits. But it can lead to more node movement and thus help to escape local minima. All nodes with $\text{child}_{\text{close}} = 0$ get inserted into the queue (see [Ham20], Proposition 9.1.). Hence, when we iterate over the touched nodes to filter the close children of $\text{p}_{\text{best}}(r)$, we also find each closeness-0-child and we can adopt it with probability $\frac{1}{2}$.

When we perform a local move for a node v_m in a graph G and encounter e equally good best parents, we want to pick one of them with probability $\frac{1}{e}$. For this purpose, we do not collect all best parents, but we randomize the decision which one to choose for upwards propagation in the tree. To obtain the desired uniform distribution, the probabilities related with this decision must be weighted according to the number of best parents in the subtrees below. Hence, when processing a node u, we determine $\text{c}_{\text{best}}(u)$, i.e. the number of best parents regarding S_u. For every vertex, c_{best} is initialized with 0 and restored after every iteration for all touched nodes (together with $\text{child}_{\text{close}}$ and $\text{score}_{\text{max}}$). The counts get propagated upwards together with the chosen best parent $\text{p}_{\text{best}}(u)$ (see Algorithm 9.2, lines 18-25).

Let $p = \text{parent}(u)$. If $\text{score}_{\max}(u) \geq \text{score}_{\max}(p)$, $\text{p}_{\text{best}}(u)$ is also a best parent in the subgraph of S_p which got considered so far (i.e. excluding subtrees rooted in children of p that still need to be processed). If $\text{score}_{\max}(u) > \text{score}_{\max}(p)$, $\text{p}_{\text{best}}(u)$ is better than any best parent propagated to p before. Subsequently, we update $\text{p}_{\text{best}}(p)$ to $\text{p}_{\text{best}}(u)$ and also set $\text{c}_{\text{best}}(p) = \text{c}_{\text{best}}(u)$.
Otherwise, if $\text{score}_{\max}(u) = \text{score}_{\max}(p) > 0$ there is at least one equally good parent in subtrees rooted in the children of p which already have been considered. Hence, we add $\text{c}_{\text{best}}(u)$ to $\text{c}_{\text{best}}(p)$ and set $\text{p}_{\text{best}}(p)$ to $\text{p}_{\text{best}}(u)$ with probability $\frac{\text{c}_{\text{best}}(u)}{\text{c}_{\text{best}}(p)}$.

In principle, it is correct to proceed likewise, if $\text{score}_{\max}(u) = \text{score}_{\max}(p) = 0$. At this point we remember that the default values of c_{best} etc. only get restored for vertices which have been inserted into the queue. Hence, we can only modify it for those nodes which get processed sooner or later during the current local move. Otherwise, the counts are faulty for following iterations and lead to incorrect results.
Back to the case that $\text{score}_{\max}(u) = \text{score}_{\max}(p) = 0$, p is not inserted into the queue while processing u. Thus, we only carry out the described upwards propagation, if p is a v_m-neighbor, i.e. if it is part of the queue anyway. Why we can omit it for non-neighbors is justified in the proof of correctness below.

Processing p, we compare p with the best parents encountered below (see Algorithm 9.2, lines 4-12). If p is a better parent, we set p_{best} to p and reset c_{best}. Hence values resulting from upwards propagation are discarded. If p is as good as the best parent found below, we increase $\text{c}_{\text{best}}(p)$ and we update p_{best} with probability $\frac{1}{\text{c}_{\text{best}}(p)}$. Otherwise we keep the best parent which got propagated upwards. In the case that p is worse than the best parents found below, the values of c_{best} and p_{best} are retained.

Randomizing decisions also means that v_m always gets moved below $\text{p}_{\text{best}}(r)$, even if the target position is just as good as v_m's original place. However, it still is possible that $\text{p}_{\text{best}}(r)$ is v_m's former parent because belongs to the set of candidates in this case.

Algorithm 9.1: `localMove`(v_m)

```
1  foreach v_m-neighbor u do
2      move up u on simple path in S
3      insert u in queue;
4  move up v_m on simple path in S
5  S ← S ⊙ v_m
6  while queue not empty do
7      u ← next element in queue;
8      processNode(u);    //(Alg. 9.2)
9  foreach node w in touched nodes do
10     if parent(w) = p_best(r) then
11         if child_close(w) = 0 then
12             add w to c_adopt with probability 1/2;
13         if child_close(w) > 0 then
14             add w to c_adopt;
15     child_close(w) ← 0;
16     score_max(w) ← 0;
17     DFS_next(w) ← w;
18     c_best(w) ← 0;
19 move v_m below p_best(r) and adopt c_adopt;
```

Algorithm 9.2: `processNode`(u)

```
1  mark u as touched;
2  score_max(u) ← max over score_max of u-children;
3  child_close(u) ← ∑ over child_close of close u-children;
4  if child_close(u) > score_max(u) then
5  |  score_max(u) ← child_close(u);
6  |  c_best(u) ← 1;
7  |  coin ← 1;
8  else if child_close(u) = score_max(u) then
9  |  c_best(u) ← c_best(u) + 1;
10 |  coin ← true with probability 1/c_best(u);
11 if coin then
12 |  p_best(u) ← u;
13 determine score_max(u) and child_close(u);    //see Alg. 4.2, lines 7-24
14 if u ≠ r then
15 |  p ← parent(u);
16 |  if score_max(u) > 0 or child_close(u) > 0 then
17 |  |  insert p in queue;
18 |  if score_max(u) > score_max(p) then
19 |  |  c_best(p) ← c_best(u);
20 |  if score_max(u) = score_max(p) and (score_max(u) > 0 or p is v_m-neighbor) then
21 |  |  c_best(p) ← c_best(p) + c_best(u);
22 |  |  coin ← true with probability c_best(u)/c_best(p);
23 |  if coin or score_max(u) > score_max(p) then
24 |  |  score_max(p) ← score_max(u);
25 |  |  p_best(p) ← p_best(u);
26 |  if child_close(u) ≥ 0 then
27 |  |  child_close(p) ← child_close(p) + child_close(u);
```

9.2. Proof of Correctness

Lemma 9.1. *Let S a skeleton graph with a root r on which Algorithm 9.1 performs a local move for a node $v_m \in V_S$. Then it holds that:*

(i) Moving v_m below $\mathrm{p}_{\mathrm{best}}(r)$, children with closeness 0 get adopted with probability $\frac{1}{2}$

(ii) If in S there are e best parents for v_m, $\mathrm{p}_{\mathrm{best}}(r)$ is chosen among them uniformly at random, i.e. each of them is considered with probability $\frac{1}{e}$.

Proof. Part (i)
Let $p = \mathrm{p}_{\mathrm{best}}(r)$, i.e. the parent of v_m at the target position of the local move.
To find the children, we want to adopt, we iterate over the touched nodes. We adopt children of p with $\mathrm{child}_{\mathrm{close}} > 0$, for children with $\mathrm{child}_{\mathrm{close}} = 0$ we throw a coin, i.e. adopt them with probability $\frac{1}{2}$ (Algorithm 9.1, line 12). In Proposition 9.1. in [Ham20] it is shown that every vertex u with $\mathrm{child}_{\mathrm{close}}(u) \geq 0$ is processed by the algorithm. Thus, going through touched nodes, we find all closeness-0-children of p and the claim follows. *Part (ii)*
To prove this statement, we show inductively that for every processed node $u \in V_S$ the claim holds regarding S_u.
We use as induction hypothesis that for every node w which got processed before u, $\mathrm{c}_{\mathrm{best}}(w)$ is determined properly and that $\mathrm{p}_{\mathrm{best}}(w)$ is chosen uniformly at random among all best parents in S_w, i.e. with probability $\frac{1}{\mathrm{c}_{\mathrm{best}}(w)}$. Based on that, we aim to show that also $\mathrm{c}_{\mathrm{best}}(u)$ is correct and that $\mathrm{p}_{\mathrm{best}}(u)$ gets picked among all best parents in S_u according to a uniform probability distribution.
In the following, the sum over the child closeness of close u-children is denoted by x and the maximum $\mathrm{score}_{\mathrm{max}}$ among all processed u-children by y (We set $y = -1$ if u is a leaf).
At first, we consider the case that $x > y$. In this situation, u is a better parent than all best parents encountered in the processed subtrees below, i.e. it is the only best parent regarding S_u (see Section 4.1). The algorithm sets $\mathrm{c}_{\mathrm{best}}(u) = 1$ and $\mathrm{p}_{\mathrm{best}}(u) = u$, i.e. picks u with probability 1 (Algorithm 9.2, line 4 ff.). Hence, in this case the claim holds regarding S_u.
If $y \geq x$, there is at least one best parent below u which is at least as good as u itself. To prove correctness in this case, we at first ensure proper upwards propagation, before we finally show, why the claim holds for S_u after processing u.
To verify that the best parent counts are properly propagated upwards, we consider c', the value of $\mathrm{c}_{\mathrm{best}}(u)$ immediately before u gets processed and show that it corresponds to the best parent count in $V_{S_u} \setminus u$.
According to Theorem 9.3. in [Ham20], all children w of u with $\mathrm{score}_{\mathrm{max}}(w) \geq 0$ get processed before u. As $y \geq x \geq 0$, this holds in particular for all children with score y.
If $y = 0$, u must be a v_m-neighbor, because a non-v_m-neighbor with $x = y = 0$ does not get inserted into the queue. Thus, for all u-children which have a score of 0, their best parent counts get added up to $\mathrm{c}_{\mathrm{best}}(u)$ (which was initially 0). Applying the induction hypothesis, it follows that c' corresponds to the number of best parents in the subtrees rooted in u's children.
If $y > 0$, $\mathrm{c}_{\mathrm{best}}(u)$ is set to $\mathrm{c}_{\mathrm{best}}(w)$ for the first child w with $\mathrm{score}_{\mathrm{max}}(w) = y$ we encounter. The best parent counts of all further children with score y are then summed up. Again we can apply the induction hypothesis to deduce that the counts got properly propagated upwards and that c' is set as desired.

We further need to consider p', the best parent to which $\mathrm{p_{best}}(u)$ points immediately before u gets processed. We claim that p' is chosen under a uniform probability distribution regarding all best parents in $V_{S_u} \setminus u$, i.e. with probability $\frac{1}{c'}$.
Let k the number of children w of u such that $\mathrm{score_{max}}(w) = y$. By w_i $(i = 1, \ldots, k)$ we denote the i-th u-child with score y which gets processed. Further by $c_1, \ldots, c_k$ we denote the value of $\mathrm{c_{best}}(u)$ after processing w_i. It holds that $c_k = c'$. We observe that $\mathrm{p_{best}}(u)$ gets updated to $\mathrm{p_{best}}(w_i)$ with probability $\frac{\mathrm{c_{best}}(w_i)}{c_i}$ (Algorithm 9.2, line 22). We fix a best parent $p_0 \in S_{w_i}$ for some $i \in (1, \ldots, k)$ and show that it gets picked with probability $\frac{1}{c'}$. For this purpose we consider the probability P for the event E that $p' = p_0$ holds. For E to occur, p_0 must be picked as $\mathrm{p_{best}}$ for S_{w_j}. Further, $p_{best}(u)$ must be updated when processing w_i and it must not change anymore for all children processed afterwards. Hence, E corresponds to the co-occurrence of the following events and P is the product of the respective probabilities:

E_1: $\mathrm{p_{best}}(w_i) = p_0$	$P_1 = \frac{1}{\mathrm{c_{best}}(w_i)}$ (induction hypothesis)
E_2: $p_{best}(u) \leftarrow p_{best}(w_i)$	$P_2 = \frac{\mathrm{c_{best}}(w_i)}{c_i}$
E_3: $\mathrm{p_{best}}(u) \not\leftarrow w_j, j = (i+1, \ldots, k)$	$P_3 = \prod_{j=i+1}^{k} 1 - \frac{\mathrm{c_{best}}(w_j)}{c_j}$

We obtain the following probability P:

$$
\begin{aligned}
P =& \frac{1}{\mathrm{c_{best}}(w_i)} \cdot \frac{\mathrm{c_{best}}(w_i)}{c_i} \cdot \prod_{j=i+1}^{k} 1 - \frac{\mathrm{c_{best}}(w_j)}{c_j} \\
=& \frac{1}{c_i} \cdot \prod_{j=i+1}^{k} \frac{c_j - \mathrm{c_{best}}(w_j)}{c_j} \\
=& \frac{1}{c_i} \cdot \prod_{j=i+1}^{k} \frac{c_{j-1}}{c_j} \\
=& \frac{1}{c_i} \cdot \frac{c_i}{c_k} \\
=& \frac{1}{c_k} = \frac{1}{c'}
\end{aligned}
\tag{9.1}
$$

Finally, we examine how $\mathrm{p_{best}}(u)$ and $\mathrm{c_{best}}(u)$ change, while u is processed, and deduce that the claim afterwards holds regarding S_u. If $y > x$, u is not a best parent. Hence, the best parent count in V_{S_u} is the same as in $V_{S_u} \setminus u$. Simply resuming the upwards propagated counts leads to the desired values for $\mathrm{c_{best}}(u)$ and $\mathrm{p_{best}}(u)$. In the case that $y = x$, u is a best parent as well. Thus, $\mathrm{c_{best}}(u)$ is set to $c' + 1$. Additionally, we update $\mathrm{p_{best}}(u)$ with probability $\frac{1}{\mathrm{c_{best}}(u)}$ (Algorithm 9.2, line 8 ff.). Any best parent except u is consequently chosen with probability $P \cdot (1 - \frac{1}{\mathrm{c_{best}}(u)}) = \frac{1}{c'} \cdot (1 - \frac{1}{\mathrm{c_{best}}(u)}) = \frac{1}{c'} \cdot \frac{\mathrm{c_{best}}(u)-1}{\mathrm{c_{best}}(u)} = \frac{1}{c'} \cdot \frac{c'}{\mathrm{c_{best}}(u)} = \frac{1}{\mathrm{c_{best}}(u)}$. Therefore, the desired uniform probability distribution is followed and the induction hypothesis holds.

With the help of this induction, we have shown that Lemma 9.1 holds regarding S_u for every processed vertex u in G and hence in particular for $S_r = S$.

□

9.3. Termination on Plateau

So far, the mover algorithm terminates when there is an iteration, during which no improvement of the editing is achieved or after a fixed maximum number of iterations. If randomness is involved, it is possible that during an iteration the editing is improved, although no progress was made in some preceding iterations. Thus, we define a *maximum plateau size*, i.e. the maximum number of iterations we continue running QTM if no improvement of the editing occurs. If for example the maximum plateau size is 5, QTM terminates, when there are 5 consecutive iterations, during which the number of edits could not be reduced. With the help of this technique we aim to escape local minima, the effect is evaluated in section 10.5.

9.4. Proof of Running Time

To randomize decisions, we add some operations to Algorithm 5.1 and to `processNode`. But however, they are all constant and hence, Algorithm 9.1 admits the same complexity as Algorithm 5.1, i.e. it runs in $\mathcal{O}(m)$ per iteration in amortized $\mathcal{O}(\deg(v_m))$ per node v_m. Nevertheless a complete QTM execution might run significantly longer if randomness is switched on, because more iterations are executed.

10. Experimental Evaluation

For the implementation of the described algorithms, we extended the Quasi-Threshold Mover from [BHSW15] which is realized in C++ within the framework of NetworKit [SSM16]. For evaluation we used the library's Python interface. The experiments were carried out on an Intel Core i7-2600K CPU with 32GB RAM. All executions ran 10 times using different seeds, the mean of the results is considered. Only the values for running times are not averaged but result from a single run.

The evaluation is structured as follows: At first, we provide an overview of the testing instances. In Section 10.2 we analyze the effect of different initializations and confirm the capability of the inclusion-minimal editing constructed with Algorithm 6.1. Subsequently, we examine the impact of reordering simple paths, before we investigate the algorithm's convergence behavior. The improvements achieved by randomness are considered in Section 10.5 and finally we have a look at the running times in practice (see Section 10.6).

name	n	m	vertices	edges	
karate	34	78	members of a karate club	friendship between members	[BMSW13]
terrorist	62	152	terrorists	associations between them	[Kre02]
dolphins	62	159	dolphins in a community	frequent association between dolphins	[BMSW13]
grassweb	75	113	species living in grassland	prey-predator-relationships	[DHC95]
lesmis	77	254	characters in the novel "Les Miserables"	coappearance in a chapter	[BMSW13]
polbooks	105	441	books about US politics	frequent co-purchasing in amazon	[BMSW13]
adjnoun	112	425	common adjectives and nouns	joint appearance in the novel "David Copperfield"	[BMSW13]
football	115	613	US colleges	opponents in an American football matches	[BMSW13]
jazz	198	2742	jazz musicians	having played together in a band	[BMSW13]

Table 10.1.: Properties and background of benchmark graphs

10.1. Instances

To obtain meaningful results, we examined a spectrum of exemplary instances. The first kind of graphs we have a look at, are rather small and quite well-known *benchmark instances*. An overview is provided in Table 10.1. As all of these graphs have < 200 nodes, thus they are not helpful to evaluate the scalability of the presented algorithms. Hence, we also consider larger instances, like the *web graphs eu-2005*, *in-2004*, *cnr-2000* and *uk-2002* obtained from web crawls[BMSW13].
In [NG13] it is stated that the structures of social networks bear resemblance to quasi-threshold graphs. We examine real world *social networks* like the graphs *lj*, *orkut* and *youtube* in which nodes stand for users and edges for friendships between them[LK14]. In the *amazon* graph, nodes represent products and they are linked with an edge, when the respective products are frequently purchased together. The *dblp* graph illustrates authors and their joint work on publications in the Digital Bibliography And Library Project. Even more instances are drawn from a project, in which the social structure of *facebook networks* is analyzed[TMP12]. *Penn94* is the largest of them, *Caltech36* the smallest.
Further, we use input from a *biological* background, more precisely protein similarity data. From the graphs provided in [RWB$^+$07] and [BBBT08], we evaluate the 101 instances which require 400 or more edits. Additionally, we examine how many of the instances which are already QTGs, get recognized by the algorithm. Last, we work on *synthetic graphs*, generated to achieve certain properties as explained in Appendix E of [BHSW15].

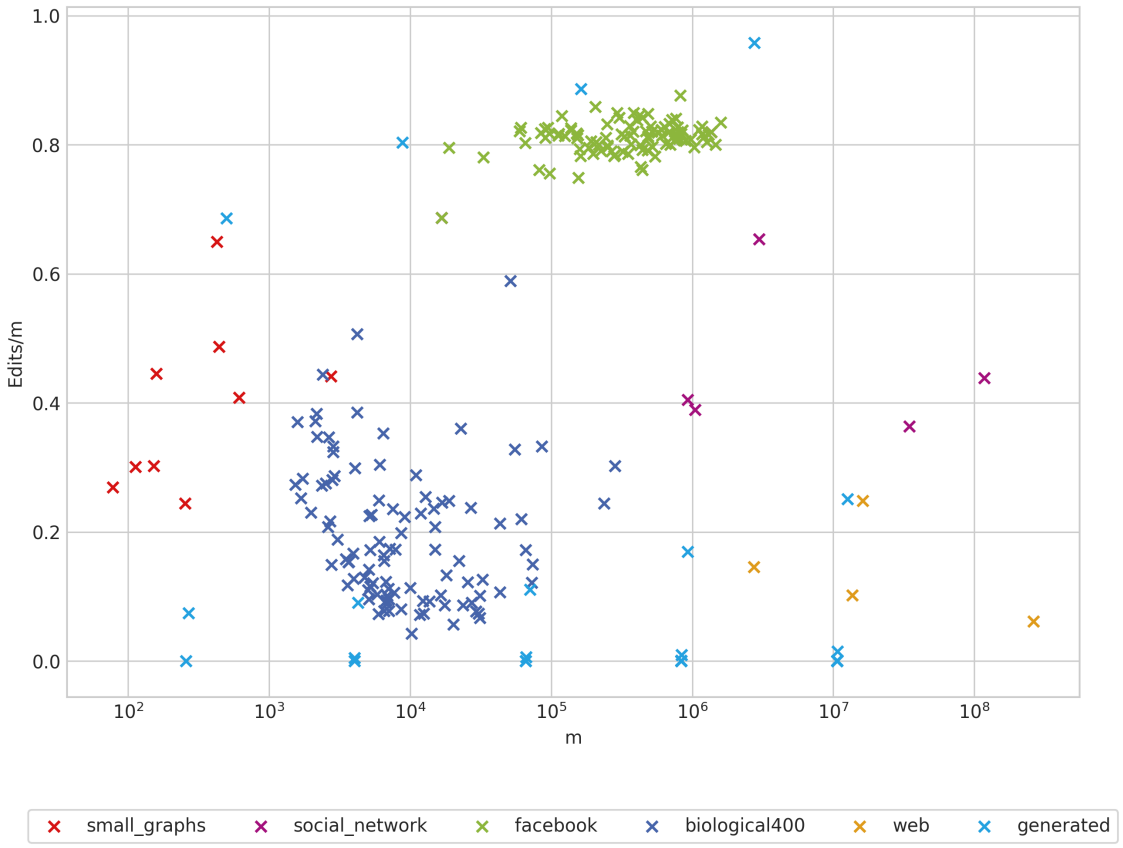


Figure 10.1.: Edits in ratio to the number of edges, the related editing is determined by running an implementation of Alg. 9.1 till convergence (initialization = asc. degree insert, maximum plateau = 5)

In order to provide an idea how the different graphs are characterized, we anticipate some of the results. In Figure 10.1, the determined number of edits is set into a ratio to the number of edges the respective graph admits. If a relatively large number of edits is required, finding a minimal editing is more difficult. Concerning the diagram, we can hence infer that the social networks and especially the facebook instances are relatively complex to solve in comparison to the web graphs and to the biological graphs. Keeping this in mind, we can understand some of the results presented within this chapter.

	i	n	m	trivial	editing	random insert	asc degree insert
karate	0	34	78	78.0	27.0	27.1	23.0
	∞			22.1	21.0	21.1	21.0
dolphins	0	62	159	159.0	95.0	84.9	84.0
	∞			75.1	75.6	76.4	74.8
terrorist	0	62	152	152.0	71.0	61.4	53.0
	∞			47.5	49.0	48.4	46.0
grassweb	0	75	113	113.0	41.0	38.2	34.0
	∞			36.2	35.0	35.8	34.0
lesmis	0	77	254	254.0	100.0	72.7	70.0
	∞			62.5	60.2	61.9	62.0
polbooks	0	105	441	441.0	285.0	253.9	247.0
	∞			231.6	223.4	229.4	222.8
adjnoun	0	112	425	425.0	354.0	302.1	313.0
	∞			290.3	282.0	290.9	289.8
football	0	115	613	613.0	404.0	312.6	287.0
	∞			253.5	255.6	256.0	251.0
jazz	0	198	2742	2742.0	2000.0	1437.1	1439.0
	∞			1246.9	1269.7	1244.6	1253.2

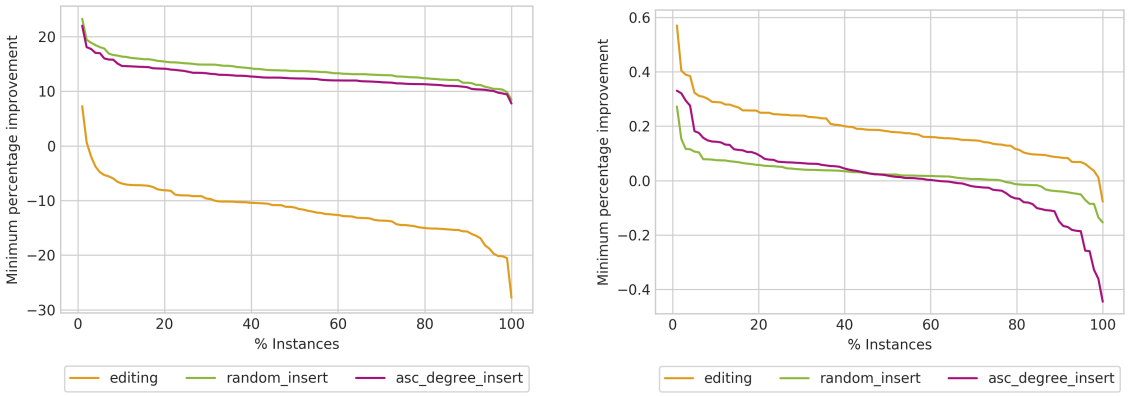
Table 10.2.: Effect of the different initializations on editing obtained for benchmark instances, number of edits is given for each initialization without further local moving ($i = 0$) and after running Alg. 5.1 till convergence ($i = \infty$)

10.2. Comparison of Initializations

In this section we evaluate the effect of different initializations for QTM. The mover algorithm can run based on a trivial initialization, where all nodes are isolated. This is compared to an initial editing constructed as proposed in [BHSW15] (see Section 4.6). Then we have a look at the improvements achieved by inserting the vertices according to Algorithm 6.1. We moreover compare different node sequences for insertion. Either the vertices are processed randomly or in ascending order according to their degree. For this section we use an implementation of Algorithm 5.1, hence simple paths get reordered but decisions are not randomized.

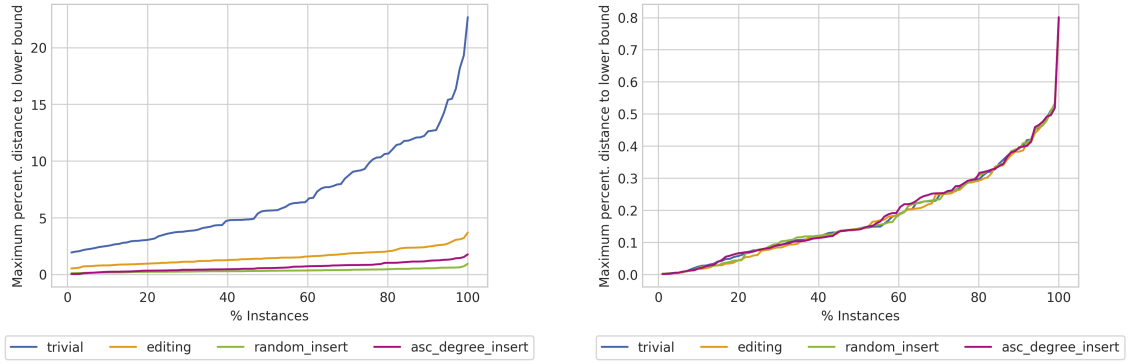
Experiments show that in most of the cases, inserting the vertices leads to a small number of initial edits, i.e. to a relatively good initial editing. Executing further iterations, the differences among the initializations strategies become more and more insignificant.
In Table 10.2, the resulting numbers of edits are shown for the small benchmark instances (for further graphs it can be found in Table A.1 in the appendix). Studying this numbers, the described effects can already be observed, but we also present further results in order to depict the impact of the initialization more clearly.

Concerning the 100 provided facebook networks, we consider the percentage improvement achieved by the advanced initialization strategies in comparison to the trivial one. We examine, for what percentage of instances an improvement of a certain minimum percentage occurs. The resulting plots can be found in Figure 10.2. Here it is obvious that inserting the vertices works better than the initial editing proposed in [BHSW15]. Further, we can observe that the initialization does not play such an important role if the algorithm runs for several iterations.



(a) Initial editing before running QTM (b) After running Alg. 5.1 till convergence

Figure 10.2.: Analysis of advanced initialization strategies for facebook networks, percentage improvement in comparison to trivial initialization



(a) Initial editing before running QTM (b) After running Alg. 5.1 till convergence

Figure 10.3.: Analysis of advanced initialization strategies for protein similarity graphs, percentage distance to optimum

Also the analysis of the results for biological graphs supports the capability of the introduced initialization strategy. With an algorithm presented in [GHS+20], lower bounds for the minimum number of edits in the protein similarity graphs can be determined. Hence, we can compare our results to these bounds. Figure 10.3 depicts, for what percentage of graphs the distance of the calculated number of edits to the lower bound is below a certain percentage. Initialization by insertion produces editings which do not deviate more than 2 % from the bound and therewith are superior to the results of the other strategies. However, we cannot observe any effect of the initialization after running the algorithm till convergence.
For this evaluation, only the 101 biological graphs requiring at least 400 edits are taken into account. Moreover, the algorithm from [GHS+20] determined 1665 instances in the bio data base which do not require any edits. The three advanced initialization strategies recognize all of them, i.e. an initial editing of size 0 is calculated.

From the technique which is used to generate the synthetic graphs, there results an upper bound for the minimum number of edits. We worked on 24 instances, and for 9 of them both inserting strategies produce an initial editing which was as least as good as the provided upper bound. Using the preexisting method to determine an initial editing, this could only be achieved for 7 instances.

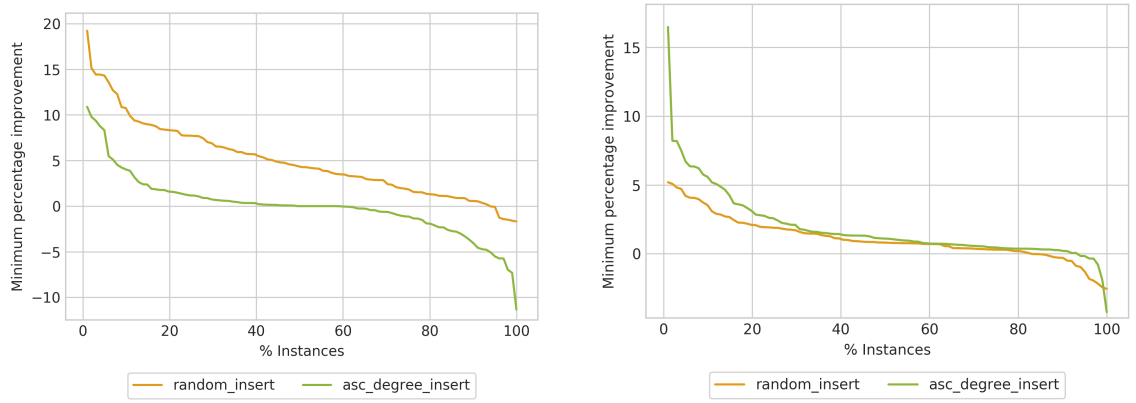
	trivial	editing	random insert	asc degree insert
karate	3.1	-1.1	-0.5	-0.5
dolphins	3.4	-0.4	-0.9	-0.4
terrorist	3.3	-0.5	0.0	-0.5
grassweb	2.7	-0.4	-1.0	-1.7
lesmis	3.4	-0.6	-0.8	-1.4
polbooks	4.6	-0.8	-0.4	-1.4
adjnoun	3.7	-0.2	-0.8	-0.2
football	3.3	-0.2	-0.1	-1.3
jazz	6.0	1.1	-1.3	0.2

Table 10.3.: Considering small benchmark graphs, the required iterations after a trivial initialization are given. Further, it is provided, how using an advanced initialization technique affects this number.

Regarding most of the graphs considered so far, a random node sequence for insertion seems to perform slightly better than sorting the vertices by their degree. Comparing the two orderings concerning all investigated instances, the random order produces a better final result in 71 graphs, whereas ascending degree insert helped in 78 examples to find a smaller editing. In 91 instances, the difference was below 1‰ with respect to the size of the resulting editing. Hence, none of the two orderings for insertion seems the have a big lead over the other one, at least if simple paths get reordered.

Apart from the resulting number of edits we analyze the number of used iterations, i.e. how many iterations are executed, before no improvement can be achieved. Like this, we get an idea of the algorithm's convergence speed. If we initialize QTM by insertion, the first insertion run is not counted as an iteration.
Using an advanced initialization technique causes that roughly speaking one iteration less is required. This result is in some way conclusive, because after trivial initialization the extra iteration is required to construct a first feasible editing. However, the running time of an advanced initialization is only approximately half of the time which is needed for an additional iteration (see Table A.3 in the appendix).
The effect of the initialization strategies on the convergence of the algorithm on small benchmark graphs is shown in Table 10.3. Regarding larger testing instances, a comparable influence can be observed .

Overall, the editing constructed by insertion is initially superior to the result of the previously proposed approach from Brandes et al. After running the Quasi-Threshold Mover, the initialization is less decisive for the size of the editing, however it contributes to faster convergence of the algorithm. Comparing different insertion orders, no statistically significant difference can be observed.



(a) Initial editing

(b) After convergence

Figure 10.4.: Effect of reordering simple paths for protein similarity graphs

10.3. Effect of Reordering Simple Paths

Reordering simple paths offers remarkable theoretical guarantees as it ensures inclusion-minimality of the respective editing. To assess what this property means in terms of number of edits, we compare the inclusion-minimal result of Algorithm 6.1 to the outcome of a version where local moving is performed according to Algorithm 4.3. This adjustment causes that simple paths do not get reordered and hence the constructed editing is not inclusion-minimal. After running QTM, inclusion-minimality cannot be guaranteed, even if simple paths get reordered. Nevertheless, we investigate the effect of the technique on the editings after convergence.

In the following, we consider the protein similarity graphs and we examine, for what percentage of instances, reordering simple paths leads to an improvement of a certain minimum percentage. We both take into account the initial editings resulting from the different versions of Algorithm 6.1 as well as the outcomes after convergence. In Figure 10.4a a largely positive impact can be observed for the initial editing produced by random insert. If nodes get inserted sorted by their degree, the effect is on average neutral. Probably sorting and reordering have similar consequences. Hence, an editing resulting from ascending degree insert is pretty close to an inclusion minimal one, even if simple paths do not get reordered. Regarding random insert, the technique helps instead to eliminate the weakness of a more unfavorable node sequence. This observation also justifies, why in the previous section, we could not detect a significant difference when comparing the results of both orderings. Regarding the editings after convergence (see Fig. 10.4b), reordering simple paths brings an improvement for some instances, in others it leads to slightly more edits.

Running the algorithms on further graphs, we obtained similar results. In Table A.2 in the appendix, the impact for the benchmark graphs is illustrated. For these instances, the effect is not that clearly visible. This could be, because they are rather small. Hence, simple paths are rather short and reordering them does not have such a big influence. Also for the facebook networks, the effect is quite weak, probably as a consequence of their increased complexity.

To sum it up, reordering simple paths turns out to be a helpful measurement for calculating editings closer to an optimal solution. It should be emphasized that the technique makes it possible, to construct relatively small initial editings, independent of the insertion order. The empirical results further suggest practical relevance of inclusion-minimality.

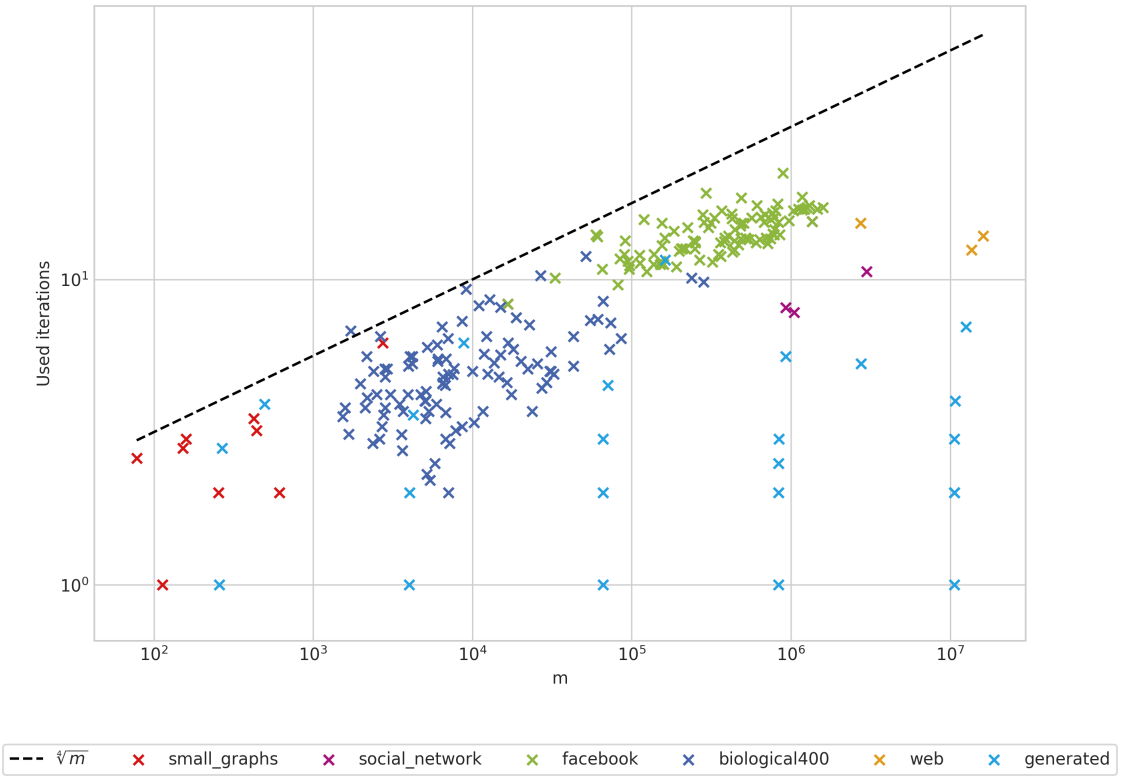


Figure 10.5.: Used iterations depending on the number of edges, the related editing is determined by running an implementation of Alg. 5.1 till convergence (initialization = ascending degree insert)

10.4. Convergence

In this section, we examine the convergence speed of QTM and we give a clue how to limit the maximum number of iterations. At first, we consider how many iterations are used till no further improvements can be found. In the corresponding Figure 10.5 we observe that this number depends on the number of edges the graph admits. A connection to the complexity of the instances cannot be ascertained. The web graphs, for example can be solved with relatively few edits and hence they are less complex than the social networks. However they require more iterations till convergence. Curve fitting revealed $\sqrt[4]{m}$ as a coarse upper bound for the used iterations in most of the graphs. Hence, the running time of a complete QTM execution can roughly be bounded from above by $\sqrt[4]{m} \cdot m$.

Brandes et al. already mentioned that if we stop the mover after a few iterations, we obtain an editing which is not significantly worse than the result after convergence [BHSW15]. We make the same observation investigating how the editing develops during the iterations. Figure 10.6 depicts how the percentage distance to the final solution decreases while running the algorithm. Even on the logarithmic scale, the respective curves fall rather steep, i.e. the main work is done during the first iterations. We have observed before that the complexity of the graphs is not reflected in the number of used iterations, but here we see the direct impact on the convergence behavior. Regarding those instances which are simpler to solve, a more extensive development takes place. This does not result from a poor initial editing but from the fact that local moving brings significant improvement on the respective graphs. For more complex instances, convergence takes place in smaller steps.

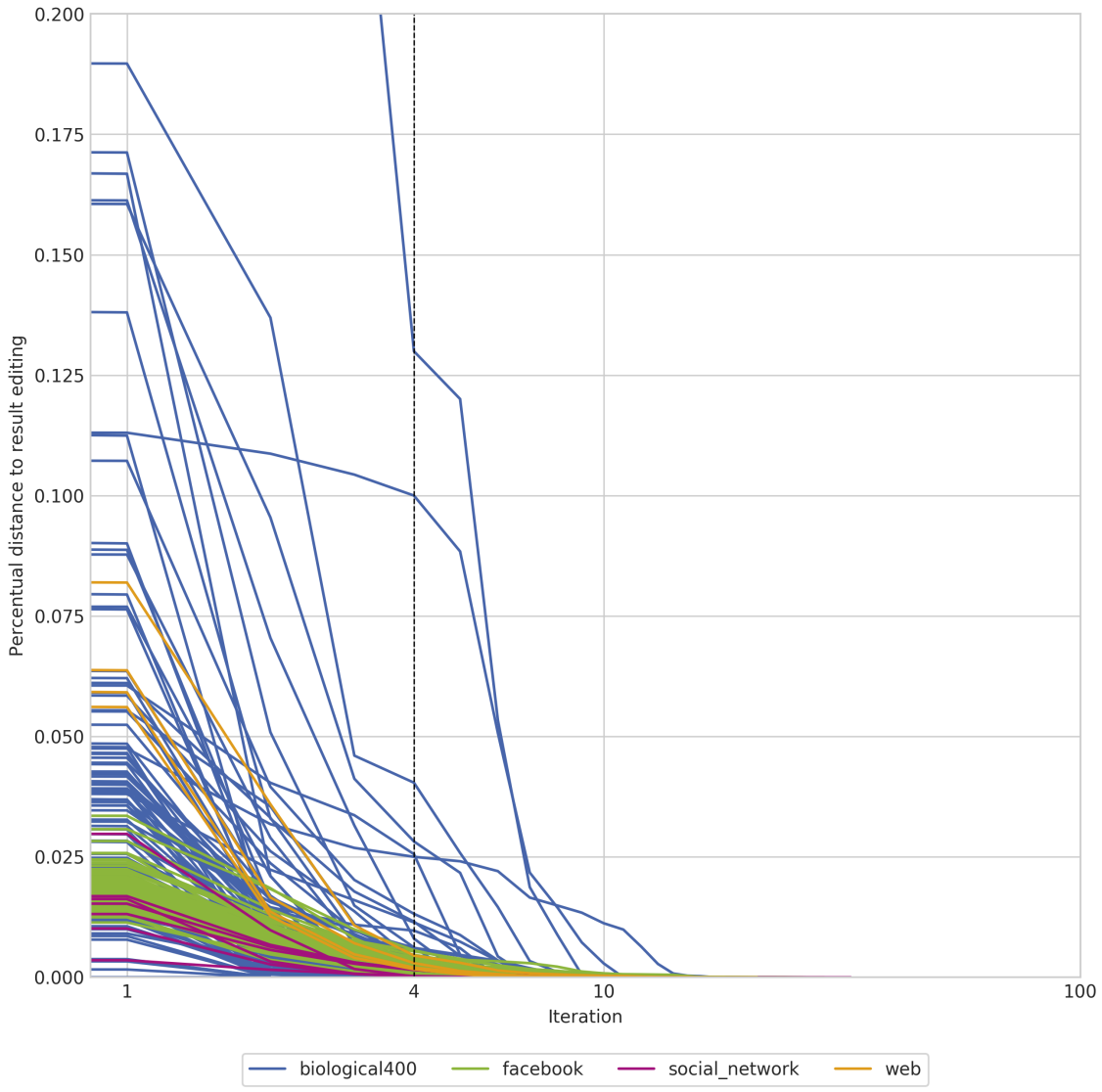
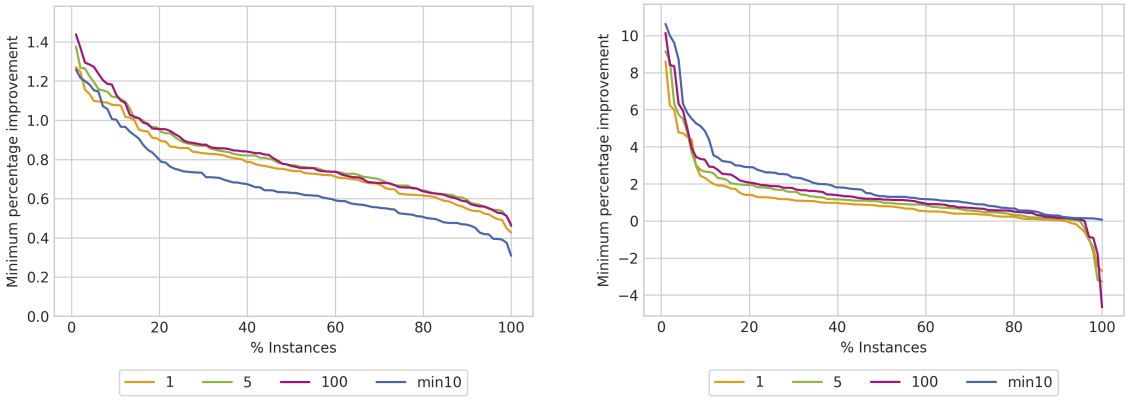


Figure 10.6.: The line graphs shows how the final editing is approached by running an implementation of Alg. 5.1 till convergence (initialization = ascending degree insert), one outlier from the biological graphs starts at $(0, 0.31)$

Overall, we agree with the statement from Brandes et al. that after 4 iterations suitable editings can be obtained. To give a prognosis for the further convergence behavior, one can compare the number of initially required edits with the number after 4 iterations. A significant percentage improvement (comparable to the biological graphs) indicates fast progress and hence it is likely that within the next few iterations the size of the editing gets further reduced to a considerable extend.



(a) Facebook networks

(b) Protein similarity graphs

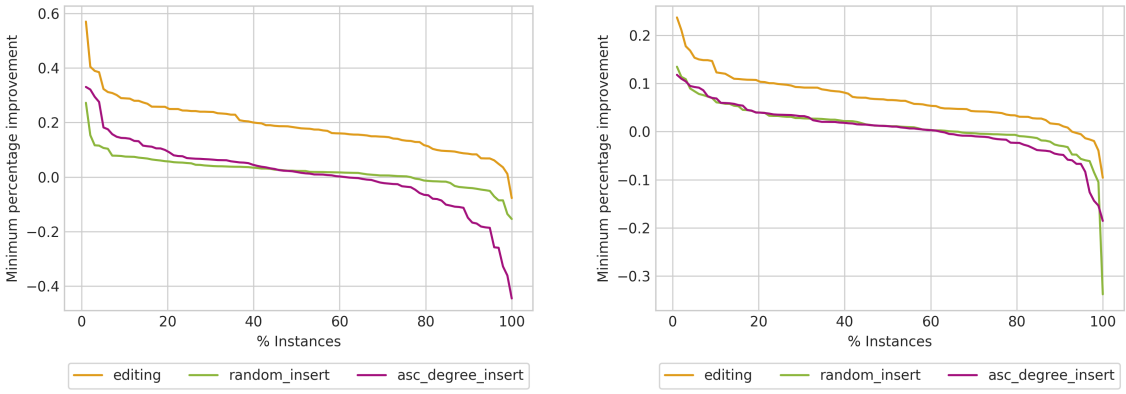
Figure 10.7.: Effect of randomization with different maximum plateau sizes

10.5. Effect of Random Decisions

Within this section we examine how random decisions affect the behavior of the Quasi-Threshold Mover. For this purpose we compare results of our implementation of Algorithm 9.1 to those of Algorithm 5.1. Additionally we examine the impact of different maximum plateau sizes, i.e. for how many iterations we keep running the algorithm even if no improvement occurs.

For the first experiment we initialize the mover with ascending degree insert and we terminate it the latest after 100 iterations. Then we investigate by what percentage Algorithm 9.1 (with maximum plateau 1, 5 and 100 resp.) improves the size of the calculated editings in comparison to Algorithm 5.1. The results are mainly positive, both for social networks (Figure 10.7a) and for protein similarity graphs (Figure 10.7b). However, the impact regarding the latter instances is much more considerable. Such a graph is rather simple to solve, thus it is probably easier to achieve improvements by random node movement. But as the convergence behavior reacts that sensitively, also negative outliers arise.

Regarding the different maximum plateau sizes, we observe that allowing larger plateaus leads only to slightly better results. In a further experiment, we evaluate a technique which is an alternative to running QTM with large plateaus. Instead, we terminate the algorithm after 10 iterations or after a plateau of size 5, but from the executions with 10 different seeds, we consider the best result instead of the average. The corresponding improvements in comparison to the editings calculated without randomness are depicted by the line *min10* in Figure 10.7. Regarding the protein similarity graphs, the technique produces superior results, for the facebook networks it does not seem suitable. Hence, in the biological instances different seeds must lead to significantly different editings. This observation reflects again that for these graphs the local mover reacts quite sensitive to randomization.

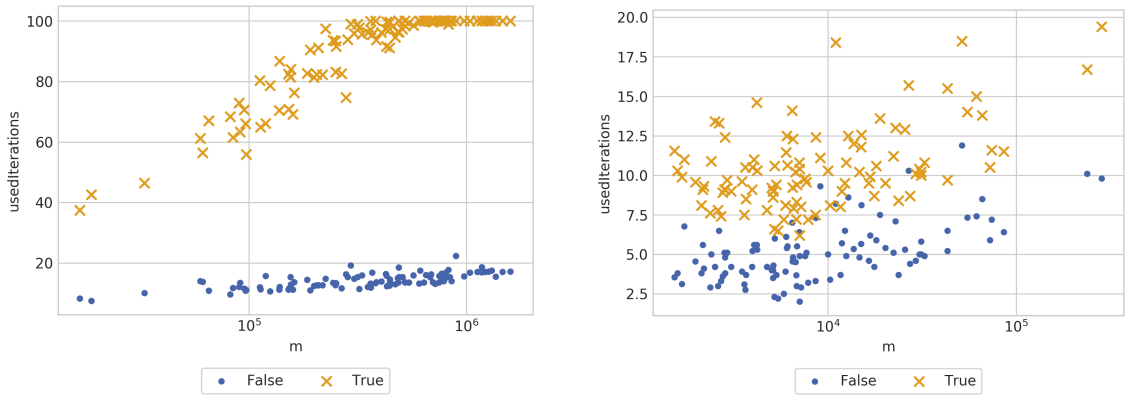


(a) After running Alg. 5.1 till convergence

(b) After running Alg. 9.1 till convergence

Figure 10.8.: Analysis of advanced initialization strategies for facebook networks, percentage improvement in comparison to trivial initialization

To confirm that the presented technique helps escaping local minima, we show that it makes the result more independent of the initialization. For this purpose, we again have a look at the diagram in which we consider the improvement of editings achieved by advanced initialization strategies. In Section 10.2, we observed that the effects only differ weakly after convergence (see Figure 10.2b and Figure 10.8a resp.). The gap between the strategies gets even smaller, if additionally, randomization is used (see Figure 10.8b). Editings of different size arise, when the algorithm converges in distinct minima depending on the initialization. As random decisions cause that all initializations produce editings of almost the same size, the technique must prevent convergence to a poor local minimum after an unfavorable initialization is used. However, the result based on the initial editing as proposed in [BHSW15] has a small but clear advance in comparison to the other ones. This speaks for the fact that the initialization is still not completely irrelevant.



(a) Facebook networks

(b) Protein similarity graphs

Figure 10.9.: Used iterations depending on whether randomization (with maximum plateau size 5) is used, the algorithm is initialized with ascending degree insert and terminated after 100 iterations at the latest

Further, we analyze the effect of randomization on the number of used iterations. We run the mover with maximum plateau size 5, i.e. at least 5 iterations are executed. Concerning the protein similarity graphs (see Figure 10.9b), the data points get approximately shifted up by 5, when randomness is switched on. Hence, probably most of the executions terminate after the first larger plateau they encounter. In contrast to that, we can have a look at the results for the facebook networks in Figure 10.9a. Here, randomization leads to a significantly higher number of used iterations, presumably because the algorithm runs on a plateau for several times but finds improvements, before the maximum plateau size is reached. We also note that on more than half of the facebook networks, the algorithm does not converge within the first 100 iterations. Hence, the observed improvements might not exploit the full potential of the technique. Overall, randomization reinforces the tendency that for a graph which is close to a QTG, good editings can be obtained after a few iterations, while the instances which are more complex to solve, converge in small steps.

In order to determine reasonable values for the maximum plateau size, we have a look at the size of actual occurring plateaus. For this purpose, we run QTM for 100 iterations and identify the maximum number of iterations with no improvements, before in the following iteration a better editing is found. To interpret the resulting Figure 10.10, we note that there is a connection between the closeness of an editing to the optimum and the biggest occurring plateau. If an editing is quite far away from minimal, there are numerous places, where edits can be saved. Hence, improvements can be found after a relatively small plateau. In the reversed case, only few changes are possible to make the editing better. This can lead to a large number of iterations without any progress, before the random movements lead to a skeleton in which local moving can further optimize the editing. For the depicted plateau sizes, the algorithm was restricted to 100 iterations. This seems to be enough to find editings for the protein similarity graphs which are quite close to an optimal solution. This justifies the large occurring plateaus. Concerning the instances which are more complex to solve, like the facebook networks, the observed plateaus are much more smaller, presumably because after 100 iterations, the calculated editing is still quite far from minimal. Obviously, several graphs must admit a maximum plateau < 5 as they did not converge in the analysis above.

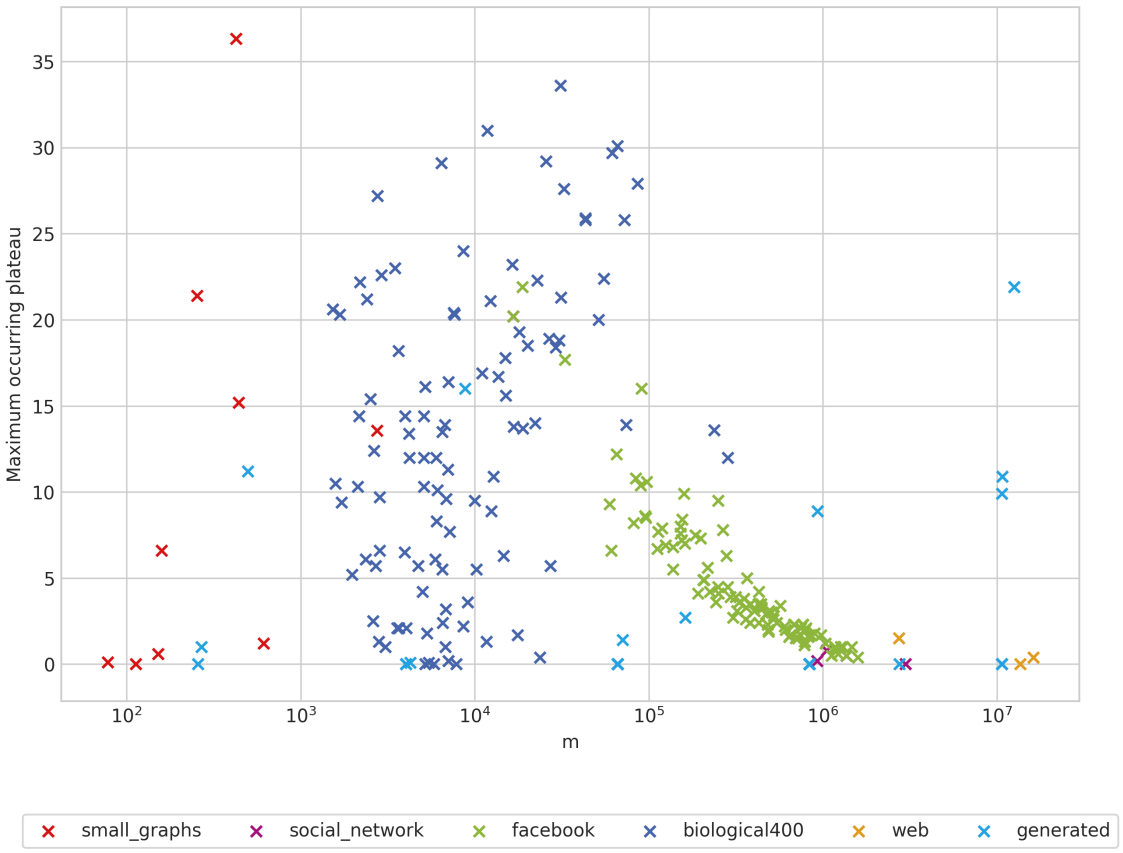


Figure 10.10.: Maximum occuring plateau depending on number of edges, the related editing is determined by running an implementation of Alg. 9.1 till for max. 100 iterations (initialization = ascending degree insert)

For tuning the parameters of QTM with randomization, the interplay of the maximum plateau size and the maximum number of iterations needs to be taken into account. To begin with, we recommend to terminate the algorithm after 10 iterations. Further, a small number $\in [1, 5]$ should be chosen as maximum plateau size, because this can already result in large numbers of required iterations (see Figure 10.9a). When the algorithm terminates before the iteration limit is reached, increasing the maximum plateau size can improve the results. Otherwise, if algorithm does not converge, there are still plateaus below the given limit and running the algorithm for some further iterations might help to get closer to the optimum. In Figure 10.7 we have already observed the weak impact of the maximum plateau size. Hence, one should not lose sight of the fact that only small improvements can be expected from the proposed technique for parameter tuning.

In conclusion, randomization turns out to be a feasible approach for further improvement of the results of the Quasi-Threshold Mover. Nevertheless one should keep in mind that increasing the maximum plateau size has only a minor effect on the calculated editing but leads to a higher number of used iterations and makes the algorithm run significantly longer.

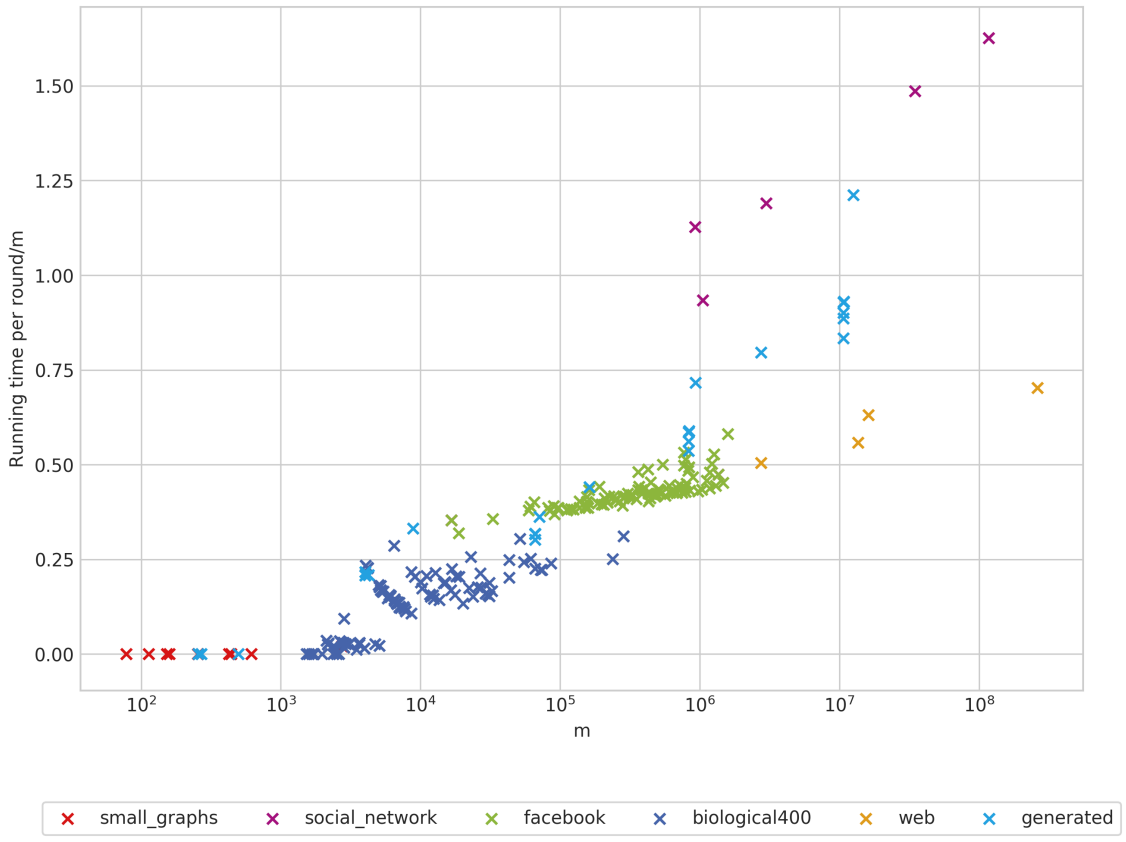


Figure 10.11.: Average running time for an iteration of Alg. 9.1 (in microseconds) in ratio to the number of edges

10.6. Running Time

In this section we analyze the running time of our implementation in order to check for compliance of practical results with the theoretical bounds. At first, we consider the average running time for one round of local moving in ratio to the number of edges the graph admits (see Figure 10.11). As one iteration has a complexity in $\mathcal{O}(m)$, we would expect that the data points are situated on a constant line. However, we observe that the relative running time grows with the size of the graph. This behavior can occur due to various reasons. One possibility is that only for large graphs the number of processed nodes gets close to the $\mathcal{O}(\deg(v_m))$ bound. Therefore, one would interpret the measured times in a way that the algorithm does not take too long on the large instances but runs even faster on the smaller graphs. Nevertheless, the relative running times must converge above a certain size. This cannot be deduced from the measurement data. In a more detailed analysis we could track, how many running time tokens are actually consumed. If the number is significantly lower than assumed in theory, this verifies the provided argument for the non-linear running time behavior.
In practice, also cache effects come into play. The larger the input instance, the more often cache faults occur. Hence, the increase in the relative running time can also result from the decreasing cache hit rate.
However, the growth factor of the running time is rather small, hence the algorithm still scales well for larger instances.

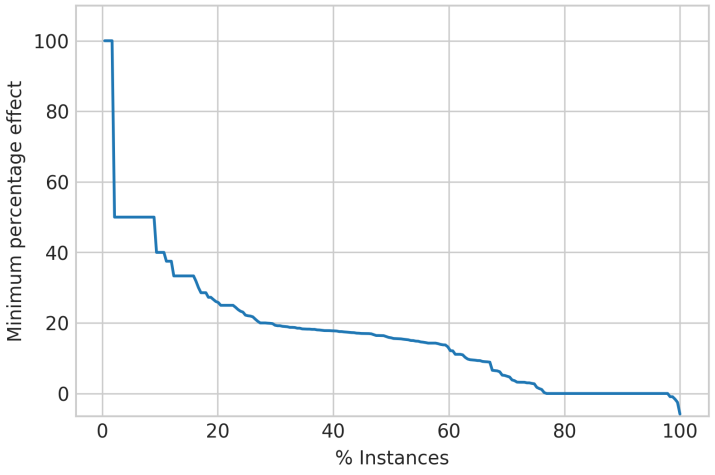


Figure 10.12.: Improvements (regarding running time of first iteration) by means of the Bucket Queue in comp.to the Level Queue, our implementation of Alg. 9.1 ran on all investigated instances

	(a)	(b)
Penn94	4.2	3.0
dblp	5.8	4.5
amazon	5.5	4.7
youtube	31.3	16.3
lj	241.9	229.4
orkut	866.4	841.6
cnr-2000	12.8	6.1
eu-2005	90.7	45.0
in-2004	72.4	32.9
uk-2002	1638.0	793.4

Table 10.4.: Running time for initialization and 4 iterations (in s), (a) - from [BHSW15], (b) - Impl. of Alg. 9.1

To evaluate how the Bucket Queue (see Section 7.1) accelerates the algorithm in contrast to the Level Queue (see Section 4.3.1), we compare running times of implementations using the respective data structure. Figure 10.12 illustrates the effect of the Bucket Queue on the running time of the first iteration after initialization. Regarding a considerable proportion of instances, the new data structure clearly improves the running time, in most of the cases, the impact is at least not negative. Only for a few instances, the Bucket Queue has a slightly decelerating effect, but this can also be due the the inaccuracies related with running time measurement. For more detailed data, we refer to Table A.4 in the appendix. Finally, we run our implementation for 4 iterations and compare the required time with the running times reported in [BHSW15]. Also here we can observe a clear improvement despite the extended dynamic forest in our implementation.

Although linear running time cannot be confirmed, we can observe the acceleration due to the Bucket Queue and furthermore, the measured times argue for the algorithm's scalability in practice.

11. Conclusion

Within the scope of this thesis, we considered the quasi-threshold editing problem and we proposed an algorithm for an inclusion-minimal solution. At first we had a look at skeletons of quasi-threshold graphs and introduced the notion of simple paths. Then we explained the Quasi-Threshold Mover (Algorithm 4.3) which optimizes an initial editing by local moving. In Chapter 5, we extended this algorithm, such that the vertices on simple paths get reordered before every local move. In particular, we sort them, such that the neighbors of the current node are placed above its non-neighbors. Further we justified that like this, the current node is moved to a position, at which the number of edits incident to it is minimized.

Based on that, we modified the mover, such that it constructs an initial editing by inserting the vertices one after the other. Subsequently, we proved that the editing resulting from the respective Algorithm 6.1 is inclusion minimal. Experimental evaluation revealed that this editing is superior to the result of the preexisting algorithm for calculating initial editings (see Section 10.2).

Furthermore, we introduced the Bucket Queue (Section 7.1) and the Dynamic Forest (Section 7.2), two data structures tailored to use them in the mover algorithm. We proved their correctness and how they make it possible, to implement the Quasi-Threshold Mover, such that it admits a running time of amortized $\mathcal{O}(m)$ per iteration. The experiments presented in Section 10.6 show that the Bucket Queue speeds up the algorithm in comparison to the Level Queue which was used before. With regard to the running time, we were moreover able to verify the algorithm's scalability in practice.

In Chapter 9 we additionally modified the Quasi-Threshold Mover such that decisions are randomized, when the respective choices lead to the same number of edits. We showed that all relevant options are considered and that we pick one among them following a uniform probability distribution. Evaluating practical results, we found out that randomization can be helpful to improve the calculated editing.

Despite of the achieved progress, there are still some questions related to quasi-threshold editing which remain open. We suppose that after several iterations, the editing is already optimal regarding certain subgraphs. During the subsequent iterations, it is hence not necessary to perform a local move for the respective vertices. For this purpose it is desirable, to develop an approach for detecting such subgraphs, on which the algorithm has already converged. Then the respective nodes can be deactivated and in each iteration, a node only gets moved if it is still active.

So far, the Quasi-Threshold Mover can only cope with unweighted graphs. As soon as weights come into play, additional challenges arise. For the protein similarity graphs, which are actually weighted, the corresponding adjustments to the algorithm could help to find more suitable editings.

Bibliography

[BBBT08] Sebastian Böcker, Sebastian Briesemeister, Quang Bao Anh Bui, and Anke Truss. A fixed-parameter approach for Weighted Cluster Editing. In *Asia-Pacific Bioinformatics Conference (APBC 2008)*, volume 5, pages 211–220, 2008.

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. Fast Quasi-Threshold Editing. Technical report, Computer & Information Science, University of Konstanz / Faculty of Informatics, Karlsruhe Institute of Technology, 2015.

[BMSW13] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, 2013.

[Cai96] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties* 1. *Information Processing Letters*, 58:171–176, May 1996.

[Cre19] Christophe Crespelle. Linear-time minimal cograph editing. 2019.

[DHC95] Hassan Ali Dawah, Bradford A. Hawkins, and Michael F. Claridge. Structure of the Parasitoid Communities of Grass-Feeding Chalcid Wasps. *Journal of Animal Ecology*, 64.6:708–720, 1995.

[DP17] Pål Drange and Michał Pilipczuk. A polynomial kernel for trivially perfect editing. *Algorithmica*, December 2017.

[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)*, volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:14. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[Ham20] Michael Hamann. *Scalable Community Detection*. PhD thesis, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), 2020.

[KF81] Toshinobu Kashiwabara and Toshio Fujisawa. On Minimal Augmentation of a Graph to Obtain an Interval Graph. *Journal of Computer and System Sciences*, 22:60–97, 1981.

[Kre02] Valdis E. Krebs. Mapping networks of terrorist cells. *Connections*, 24:43–52, April 2002.

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection, July 2014. http://snap.stanford.edu/data/index.html.

[NG13] James Nastos and Yong Gao. Familial groups in social networks. *Social Networks*, 35:439–450, July 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 *Computational Systems Bioinformatics (CSB 2007)*, volume 6, pages 391–401, 2007.

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

[TMP12] Amanda L. Traud, Peter J. Mucha, and Mason A. Porter. Social Structure of Facebook Networks. *Physica A: Statistical Mechanics and its Applications*, 391:4165–4180, August 2012.

[YCC96] Jing Yan, Jer-Jeong Chen, and Gerard Chang. Quasi-threshold graphs. *Discrete Applied Mathematics*, 69:247–255, August 1996.

Appendix

A. Additional Evaluation Results

	i	n	m	trivial	editing	r.insert	a.insert
Caltech36	0	769	16656	16656.0	15447.0	12783.2	12993.0
	∞			11620.3	11587.8	11625.9	11583.0
Penn94	0	41554	1362229	1362229.0	1514116.0	1181094.5	1198145.0
	∞			1127432.7	1125116.2	1127082.6	1126735.1
dblp	0	317080	1049866	1049866.0	477525.0	456032.1	449990.0
	∞			420539.0	413011.6	418523.4	423345.7
amazon	0	334863	925872	925872.0	493474.0	440199.9	441861.0
	∞			398075.2	389459.3	395993.9	398752.7
youtube	0	1134890	2987624	2987624.0	2146571.0	2027773.7	2032960.0
	∞			1980067.9	1959686.6	1975308.0	1994619.2
lj	0	3997962	34681189	26794000.0[1]	32451000.0[1]		27315367.0[2]
	∞			25749000.0[1]	25577000.0[1]		25659243.0[2]
orkut	0	3072441	117185083	106367000.0[1]	133086000.0[1]		107715816.0[2]
	∞			10350700.0[1]	103278000.0[1]		103344479.0[2]
cnr-2000	0	325557	2738969	2738969.0	1030457.0	451993.1	619676.0
	∞			407758.8	405996.2	405824.3	411239.5
eu-2005	0	862664	16138468	16138468.0	7645330.0	4205309.6	6970580.0
	∞			3899219.1	3880592.7	3861626.3	4041165.1
in-2004	0	1382908	13591473	13591473.0	2706124.0	1521164.7	2228698.0
	∞			1380587.6	1392871.0	1375389.2	1434061.9
uk-2002	0	18520486	261787258	42193000.0[1]	68969000.0[1]		63590595.0[2]
	∞			31042000.0[1]	31178000.0[1]		32725796.0[2]

Table A.1.: Results for benchmark instances, social networks and web graphs
Number of edits is given for each initialization without further local moving ($i = 0$) and after running Alg. 5.1 till convergence ($i = \infty$)
r. insert = random insert, a. insert = ascending degree insert

[1] Results from [BHSW15]
[2] Results from a single run only

	i	random insert		asc degree insert	
		Alg. 4.3	Alg. 5.1	Alg. 4.3	Alg. 5.1
karate	0	28.6	-1.5	21.0	2.0
	∞	21.2	-0.1	21.0	0.0
dolphins	0	89.6	-4.7	85.0	-1.0
	∞	76.5	-0.1	77.9	-3.1
terrorist	0	63.0	-1.6	53.0	0.0
	∞	48.8	-0.4	46.2	-0.2
grassweb	0	44.4	-6.2	34.0	0.0
	∞	37.3	-1.5	34.0	0.0
lesmis	0	77.2	-4.5	70.0	0.0
	∞	62.3	-0.4	62.0	0.0
polbooks	0	262.2	-8.3	256.0	-9.0
	∞	231.4	-2.0	235.5	-12.7
adjnoun	0	306.0	-3.9	309.0	4.0
	∞	293.8	-2.9	288.0	1.8
football	0	296.4	16.2	293.0	-6.0
	∞	255.3	0.7	251.7	-0.7
jazz	0	1435.9	1.2	1379.0	60.0
	∞	1231.2	13.4	1237.9	15.3

Table A.2.: For each of the benchmark graphs, the table provides the number of edits we obtain when running Algorithm 4.3. Further, it is indicated how this number changes when additionally simple paths get reordered, i.e. Algorithm 5.1 is used instead.

graph	iteration	initialization
Caltech36	6	3
Penn94	646	340
dblp	981	478
amazon	1044	516
youtube	3554	1935
orkut	190588	91408
lj	51542	24728
cnr-2000	1383	552
eu-2005	10181	3593
in-2004	7586	2558
uk-2002	183911	55574

Table A.3.: Comparison of the time required for initialization by inserting the vertices ordered by degree with the average time for an iteration of QTM, values are given in milliseconds.

	bucket queue	level queue
Caltech36	6	7
Penn94	635	786
dblp	1017	1047
amazon	1051	1086
youtube	3668	3782
orkut	186040	192154
lj	50695	52368
cnr-2000	1420	1566
eu-2005	10772	11364
in-2004	7509	8253
uk-2002	186213	199137

Table A.4.: Time for the first iteration of Alg. 9.1 under use of the respective queue, values are given in milliseconds