Karlsruhe Institute of Technology

On Threshold Editing

Bachelor Thesis of

Matthias Schimek

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisor:	Michael Hamann, M.Sc.

Time Period: 15th June 2016 – 14th October 2016

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 10th October 2016

Abstract

In this thesis we consider the problems threshold deletion and threshold editing on quasi-threshold graphs. Both problems ask to transform an input graph into a threshold graph using a minimal number of edge operations. In threshold deletion these are restricted to the deletion of edges only, whereas in threshold editing both the deletion and the insertion of edges are permitted. Threshold graphs are graphs that do not contain a P_4, C_4 or $2K_2$ as a vertex-induced subgraph. Both problems are $\mathcal{NP}$-complete on general graphs.

We focus on quasi-threshold graphs as input graphs, i.e. graphs that do not contain a P_4 or C_4 as a vertex-induced subgraph. For the problem of threshold deletion we will give an algorithm that needs $\mathcal{O}(|V| + |E|)$ time, where $Q = (V, E)$ is the input graph. In the case of threshold editing we present an algorithm that needs exponential time in the size of the vertex set of the input graph. However, our evaluation shows that the running time of this algorithm is feasible for a set of randomly generated input graphs consisting of up to 300 vertices. Furthermore, we give a heuristic for this problem with a quadratic running time.

Deutsche Zusammenfassung

In dieser Arbeit werden die Probleme Threshold Deletion und Threshold Editing auf Quasi-Threshold-Graphen untersucht. Das Problem Threshold Deletion besteht darin, eine minimale Anzahl von Kanten des Eingabegraphen zu löschen, um diesen in einen Threshold-Graphen umzuwandeln. Für Threshold Editing dürfen nicht nur Kanten gelöscht, sondern auch Kanten hinzugefügt werden, um den Eingabegraphen in einen Threshold-Graphen zu überführen. Threshold-Graphen können durch verbotene Subgraphen charakterisiert werden. Es sind genau die Graphen, welche keinen P_4, C_4 oder $2K_2$ als knoteninduzierten Subgraphen enthalten. Auf allgemeinen Graphen sind beide Probleme $\mathcal{NP}$-vollständig.

Als Eingabegraphen werden in dieser Arbeit nur Quasi-Threshold-Graphen betrachtet. Diese sind Graphen, die keinen P_4 oder C_4 als knoteninduzierten Subgraphen enthalten. Für Threshold Deletion wird ein Algorithmus vorgestellt, der das Problem in $\mathcal{O}(|V| + |E|)$ löst, wobei $Q = (V, E)$ der Eingabegraph ist. Ebenso wird ein Algorithmus für das Problem Threshold Editing gegeben, dieser hat eine exponentielle Laufzeit in der Knotenanzahl des Eingabegraphen. Jedoch erweist sich der Algorithmus auf den zur Evaluation verwendeten zufällig generierten Graphen mit bis zu 300 Knoten als praktikabel. Weiterhin wird eine Heuristik für dieses Problem vorgestellt, deren Laufzeit quadratisch ist.

Contents

1. Introduction

The main topics of this thesis are the graph modification problems *threshold editing* and *threshold deletion*. In general, a graph modification problem asks to transform an input graph G into a graph G' with certain properties using a number of operations such as the insertion or deletion of vertices or edges. We denote by threshold editing the modification problem that only allows the insertion/deletion of edges in order to obtain a threshold graph from an input graph G. A threshold graph is a graph that does not contain P_4, C_4 or $2K_2$ as vertex-induced subgraphs (see Section 2.2 for further details and Figure 1.1 for an illustration). In the following, we define the problem more precisely:

Threshold Editing

Input: A graph $G = (V, E)$ and $k \in \mathbb{N}$.

Problem: Is there a set $F \subseteq V \times V$ of size at most k such that $G' = (V, E\Delta F)$ is a threshold graph?

If the only modification allowed is the deletion of edges, the corresponding problem is called threshold deletion.

Threshold Deletion

Input: A graph $G = (V, E)$ and $k \in \mathbb{N}$.

Problem: Is there a set $F \subseteq E$ of size at most k such that $G' = (V, E \backslash F)$ is a threshold graph?

The $\mathcal{NP}$-completeness of threshold deletion is shown in [Mar94]. In [DDLS15], *Drange et al.* show that the same holds for threshold editing, even on split graphs. Those are graphs whose vertex sets can be partioned into a clique and an independent set [BLS99]. These graphs can also be characterized as the graph class that does not contain C_4, C_5 or $2K_2$ as vertex-induced subgraphs. Furthermore, they give algorithms solving threshold editing and threshold deletion for an input graph $G = (V, E)$ in $2^{\mathcal{O}(\sqrt{k}\log k)}+$ poly$(|V|)$ time, where k is the parameter from the problem definitions. This shows in particular that threshold editing and threshold deletion are *fixed-parameter tractable*, i.e. their complexity can be described

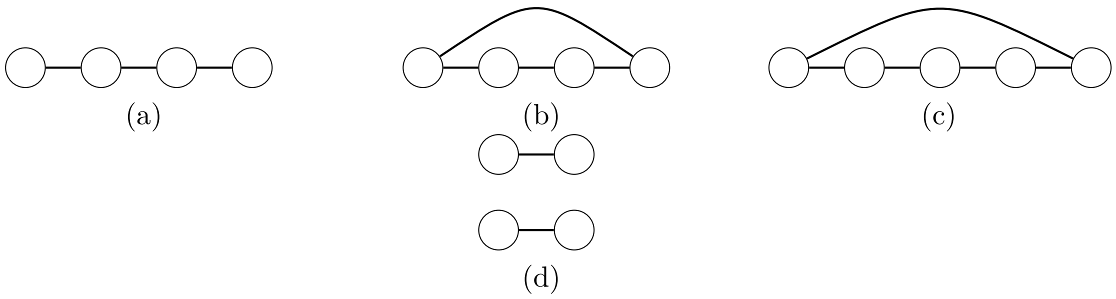


Figure 1.1: P_4 (a), C_4 (b), C_5 (c) and $2K_2$ (d).

by $f(k) \cdot |P|^c$, where P is a problem instance, $f : \mathbb{N} \to \mathbb{N}$, $c \in \mathbb{N}$ and k is a parameter of P. This result can be genearalized. In [Cai96], *Cai* shows that all graph modification problems are fixed-parameter tractable if the target graph class can be characterized by a finite set of forbidden subgraphs.

In this thesis we will focus on threshold editing and threshold deletion with *quasi-threshold graphs* as our problem input, i.e. graphs that do not contain P_4 or C_4 as vertex-induced subgraphs. As threshold editing is $\mathcal{NP}$-complete on split graphs, the question arises whether the same holds true for quasi-threshold graphs, whose set of forbidden subgraphs is related. Quasi-threshold graphs can be recognized in linear time [JHJJC96].

One of the applications of quasi-threshold graphs is in graph clustering. In [NG13] *Nastos et al.* present the following idea in order to find clusters in social networks: a graph G representing a social network is transformed into a nearest quasi-threshold graph Q, i.e. the editing distance from G to Q is minimal. The connected components of Q are viewed as clusters of G. In the field of bioinformatics, the approach of transforming an input graph to a set of disjoint cliques and interpreting these cliques as clusters is called *cluster editing* and widely discussed (see for instance [BB13]). One could adapt this concept for threshold graphs. Hence, theoretical considerations in this area might be of interest.

Our contribution. In the case of threshold deletion, we give an algorithm that computes for any quasi-threshold graph $Q = (V, E)$ a nearest threshold graph $G' = (V, E \backslash F)$, i.e. the set F has minimal size among all threshold graphs with vertex set V. The algorithm has $\mathcal{O}(|V| + |E|)$ complexity, whereas the problem is $\mathcal{NP}$-complete on general graphs. Similarly for threshold editing we present an algorithm that computes a nearest threshold graph $G' = (V, E \Delta F)$, i.e. F has minimal size among all threshold graphs with vertex set V. This algorithm needs $\mathcal{O}(2^{n(\log n+1)+\log n} + |V| + |E|)$ time, where n is the number of vertices that are not leaves in a skeleton of the input graph (see Section 2.1 for details about the skeleton representation of quasi-threshold graphs). Our evaluation shows that its running time is feasible for a set of randomly generated graphs consisting of up to 300 vertices. This suggests that our algorithm can solve threshold editing for quasi-threshold graphs for which the use of the algorithm given in [DDLS15] would not be practical as the quasi-threshold graphs of this size that we examined in our evaluation required about 800 edits. The question whether threshold editing is $\mathcal{NP}$-complete on quasi-threshold graphs remains open. Furthermore we give a heuristic for threshold editing. It is based on the *Quasi-Threshold Mover* given in [BHSW15], a heuristic for quasi-threshold editing. Additionally, we compare the exact algorithms for threshold deletion, threshold editing and the heuristic for threshold editing with respect to running time and to differences in the number of calculated edits.

Preliminaries. Let G be a graph. We define V_G as its vertex set and E_G as its edge set. For a vertex $v \in V_G$, by $N_G(v)$ we denote the neighbourhood of v, i.e. $\mathrm{N}(v) = \{u \mid \{u, v\} \in E_G\}$.

Further we define $\bar{\mathrm{N}}_G(v) := \mathrm{N}_G(v) \cup \{v\}$. For a vertex $v \in V_G$, the graph $G - v$ is the subgraph of G induced by $V_G \backslash \{v\}$. The graph $G + v$, where $v \notin V_G$, has the same edge set as G and the vertex set $V_G \cup \{v\}$. For a directed tree T and $v \in V_T$, $\mathrm{Depth}_T(v)$ denotes the depth of v in T. With Leaves_T as the set of all leaves in T we define $\mathrm{Inner}_T := V_T \backslash \mathrm{Leaves}_T$ as the set of all 'inner' vertices, i.e. the vertices that are not leaves. For a vertex v, $\mathrm{Ancestors}_T(v)$ shall be the set of all ancestors of v in T, i.e. all vertices on the path from the root of T to v but not the vertex v itself. Let the set $\mathrm{Children}_T(v)$ consist of all children of v in T and let the graph $\mathrm{Subtree}_T(v)$ be the subtree of v in T including v.

Let G be a graph and F a set with $F \subseteq V_Q \times V_Q$. Let T be a threshold graph with $V_T = V_G$ and $E_T = E_G \Delta F$. The *edit distance from G to T* is the size of F. For a set $F \subseteq E_G$ and a threshold graph T with $V_T = V_G$ and $E_T = E_G \backslash F$, the size of F is called *deletion distance from G to T*. When there is no 'target' threshold graph mentioned, the terms *edit* or *deletion distance of Q* mean the edit or deletion distance to a nearest threshold graph of G.

2. Editing to Threshold Graphs

2.1 Quasi-Threshold Graphs

First we define the class of *quasi-threshold graphs*.

Definition 2.1. *A quasi-threshold graph (*QTG*) is a graph that does not contain a P_4 or C_4 as a vertex-induced subgraph.*

The following is an equivalent inductive characterization [JHJJC96]:

1. A single vertex is a QTG.
2. Adding a universal vertex to a QTG results in a QTG. A universal vertex is a vertex that is adjacent to all other vertices of a graph.
3. The disjoint union of two QTGs is a QTG.

In [JHJJC96], it is shown as well that every quasi-threshold graph is induced by a directed forest. A directed forest $F = (V', E')$ is a set of directed trees. A quasi-threshold graph Q is induced in the following way: The vertex set of F is taken as that of Q and $\{u, v\} \in E_Q$ if and only if there is a path from v to u or a path from u to v in F. A directed forest which induces a quasi-threshold graph is called a *skeleton representation* of this graph (see Figure 2.1). It also holds that every directed forest induces a quasi-threshold graph.
In [JHJJC96] and [BHSW15] linear time recognition algorithms are given that yield a skeleton representation if the input graph is a quasi-threshold graph.

In the following we prove some structural properties of skeleton representations.

Lemma 2.2. *Let* $\mathrm{S}(Q)$ *be a skeleton of a connected QTG Q, then the following holds:*

1. *If* $\mathrm{Depth}_{\mathrm{S}(Q)}(v) < \mathrm{Depth}_{\mathrm{S}(Q)}(w)$ *and* $\{v, w\} \in E_Q$ *then* $\bar{\mathrm{N}}_Q(w) \subseteq \bar{\mathrm{N}}_Q(v)$.
2. *If* $\mathrm{Depth}_{\mathrm{S}(Q)}(v) = \mathrm{Depth}_{\mathrm{S}(Q)}(w)$ *then* $\{v, w\} \notin E_Q$.

Proof. 1. If $\{v, w\} \in E_Q$, then there must be a path from v to w in $\mathrm{S}(Q)$ or vice versa. As $\mathrm{Depth}_{\mathrm{S}(Q)}(v) < \mathrm{Depth}_{\mathrm{S}(Q)}(w)$, the path goes from v to w. But then there is a path from v to every vertex x that can be reached from vertex w in $\mathrm{S}(Q)$. Also, every vertex x, that can reach v, has a path to w. Therefore we have $\bar{\mathrm{N}}_Q(w) \subseteq \bar{\mathrm{N}}_Q(v)$.

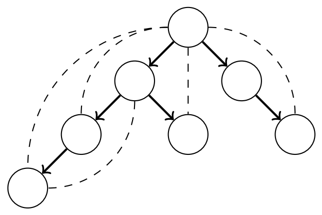


Figure 2.1: Skeleton of a QTG with implicit edges.

2. As two vertices v, w that have the same level in a directed tree cannot reach each other, it follows from the defintion of a skeleton that $\{v, w\} \notin E_Q$.

□

Lemma 2.3. *Iff a* QTG Q *contains a universal vertex* u*, then a skeleton representation* $\mathrm{S}(Q)$ *of* Q *is a tree.*

Proof. If $\mathrm{S}(Q)$ is a tree, then there is a path from its root u to every vertex $v \in V_Q \backslash \{u\}$ in $\mathrm{S}(Q)$. This implies that there is an edge $\{u, v\}$ in Q. Hence, u is a universal vertex of Q. If u is a universal vertex in Q then it has an edge to every $v \in V_Q \backslash \{u\}$. Thus, there is a path from u to v. It follows that the representing forest must be connected. □

Lemma 2.4. *Let* Q *be a quasi-threshold graph with at least one universal vertex. Any skeleton representation* $\mathrm{S}(Q)$ *of* Q *is a tree with root* r *and* r *is a universal vertex of* Q*.*

Proof. We know from Lemma 2.3 that any skeleton representation of a quasi-threshold graph that has a universal vertex is a tree. Further Lemma 2.2 (1) shows that $\mathrm{Depth}_{\mathrm{S}(Q)}(u) \leq \mathrm{Depth}_{\mathrm{S}(Q)}(v)$ for all universal vertices u and non-universal vertices v. Since the root of a tree is the only vertex with a depth of zero, it follows that r must be a universal vertex of Q. □

2.2 Threshold Graphs

Similar to quasi-threshold graphs, *threshold graphs* can be defined by forbidden subgraphs.

Definition 2.5. *A threshold graph (TG) is a graph that does not contain a* P_4*,* C_4 *or* $2K_2$ *as a vertex-induced subgraph.*

The following inductive characterization is equivalent [CH73].

1. A single vertex is a *TG*.
2. Adding an isolated vertex results in a *TG*.
3. Adding a universal vertex results in a *TG*.

Since every threshold graph is also a quasi-threshold graph, it has a skeleton representation. But a skeleton of a threshold graph can be characterized more restrictively. A directed forest that induces a threshold graph cannot have multiple connected components with more than two vertices as this would imply that the corresponding threshold graph would contain $2K_2$ as an induced subgraph. The connected component that consists of more than one vertex is a directed tree in which all vertices are either on the central path or leaves of vertices on the central path. Otherwise the induced graph would contain $2K_2$ as an induced subgrah.

It follows that a threshold skeleton consists of a directed caterpillar and possibly some additional isolated vertices (Figure 2.2). Furthermore, an algorithm for the recognition of threshold graphs can be obtained from an algorithm recognazing quasi-threshold graphs to which we add the test whether the generated skeleton has the strucutre of a caterpillar (plus possibly additional isolated vertices). This can be achieved in linear time.

2.3 Quasi-Threshold to Threshold Deletion

In this section we treat the problem threshold deletion on quasi-threshold graphs. We will give a linear time algorithm that computes a nearest threshold graph for a given quasi-threshold input graph. Recall that the problem of threshold deletion is $\mathcal{NP}$-complete on general graphs.

As isolated vertices form a threshold graph we could delete all edges in Q to obtain a threshold graph from Q. But this is certainly not an optimal solution. Before giving the algorithm we prove two lemmas. They are both not restricted to threshold deletion but also work for the more general case of threshold editing.

Lemma 2.6. *Let Q be a quasi-threshold graph and u a universal vertex in Q. Then u is also a universal vertex in all nearest threshold graphs of Q.*

Proof. Let T be a nearest threshold graph of Q that can be reached by adding or deleting edges of Q. Assume that u is not a universal vertex of T (from which follows that e edges incident to u have been deleted). As threshold graphs are defined by forbidden subgraphs, $T' = T - u$ is also threshold. If we add u as a universal graph to T', the resulting graph T'' is still a threshold graph. The edit distance from Q to T'' is smaller than the distance to T as we do not have to delete e edges. This contradicts the minimality of T. Thus, u has to be a universal vertex in T. □

Lemma 2.7. *Let Q be a quasi-threshold graph. We can add a universal vertex u obtaining a connected quasi-threshold graph Q' that has the same edit distance as Q.*

Proof. As Q is quasi-threshold, it follows from the inductive characterization that Q' is also quasi-threshold. The graph Q' is connected because u is a universal vertex. Let T' be a nearest threshold graph of Q'. It follows from Lemma 2.6 that u is a universal vertex in T'. Removing u results in a threshold graph T with the vertex set of Q. Thus, the edit distance of Q' is an upper bound for that of Q. As adding a universal vertex u to any nearest threshold graph T of Q results in a threshold graph with the vertex set of Q' it follows that the edit distance of Q is also an upper bound for that of Q'.
Hence, the edit distances of Q and Q' are equal. □

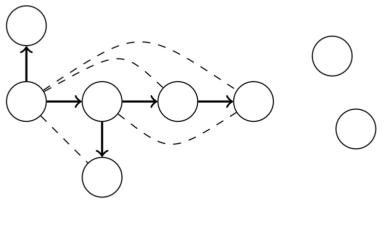


Figure 2.2: Skeleton of a threshold graph with implicit edges. The two vertices on the left are isolated vertices.

In the following we restrict ourselves again to the problem of threshold deletion.

Definition 2.8. *Let* $\mathrm{S}(Q)$ *be a skeleton of a connected quasi-threshold graph, i.e.* $\mathrm{S}(Q)$ *is a tree. We define the following measures for all vertices* v *of* $\mathrm{S}(Q)$*.*

1. ***Number of edges*** $\#\mathrm{Edges}(v)$
   The measure $\#\mathrm{Edges}(v)$ *denotes the number of edges in the subgraph of* Q *induced by the vertices in* $\mathrm{Subtree}_{\mathrm{S}(Q)}(v)$*.*
2. ***Deletion distance*** $\mathrm{Dist}(v)$
   This measure is the deletion distance of the quasi-threshold graph induced by the subgraph of v *in* $\mathrm{S}(Q)$*, i.e. the minimal number of deletions that have to be performed to obtain a threshold graph.*

In the following paragraph we set $C(v) := \mathrm{Children}_{\mathrm{S}(Q)}(v)$. The definition of the skeleton representation directly yields that

$$\#\mathrm{Edges}(v) = \sum_{c \in C(v)} \#\mathrm{Edges}(c) + |\,\mathrm{Subtree}_{\mathrm{S}(Q)}(c)|$$

for every $v \in \mathrm{S}(Q)$.

Lemma 2.9. *Let* Q *be a connected QTG with a skeleton* $\mathrm{S}(Q)$*. For all* $v \in V_Q$ *the following holds:*

$$\mathrm{Dist}(v) = \begin{cases} \min\limits_{c \in C(v)} \left(\left(\sum\limits_{d \in C(v)} \#\mathrm{Edges}(d) \right) - \#\mathrm{Edges}(c) + \mathrm{Dist}(c) \right) & \text{if } C(v) \neq \emptyset \\ 0 & \text{if } C(v) = \emptyset \end{cases}$$

Proof. We prove this lemma by induction on the number of vertices in Q:

Base case $n = 1$**:**
As $C(v)$ is empty and a graph with a single vertex is already threshold, it follows that $\mathrm{Dist}(v) = 0$.

Induction hypothesis:
The claim holds for a connected QTG Q with at most $n \geq 1$ vertices.

Inductive step:
Let Q' be a connected QTG with $n+1$ vertices and $S(Q')$ a skeleton of Q'. The root u of $\mathrm{S}(Q')$ is a universal vertex of Q' (see Lemma 2.4).

We have to consider the following two cases:

1. u **has one child** c**.**
   Let Q be the quasi-threshold graph induced by $\mathrm{S}(Q) := \mathrm{Subtree}_{\mathrm{S}(Q')}(c)$. The formula for the deletion distance holds for all $v \in V_{Q'} \backslash \{u\}$ by the induction hypothesis as

   $$\mathrm{Subtree}_{S(Q')}(v) = \mathrm{Subtree}_{S(Q')-u}(v) = \mathrm{Subtree}_{S(Q)}(v)$$

   and $|V_Q| \leq n$. Thus, we only have to show the formula's correctnes for u. From Lemma 2.7 we know that Q' has the same deletion distance as Q. Therefore

   $$\begin{aligned} \mathrm{Dist}(u) &= \mathrm{Dist}(c) \\ &= \#\mathrm{Edges}(c) - \#\mathrm{Edges}(c) + \mathrm{Dist}(c) \\ &= \min_{c \in C(u)} \left(\left(\sum_{d \in C(u)} \#\mathrm{Edges}(d) \right) - \#\mathrm{Edges}(c) + \mathrm{Dist}(c) \right) \end{aligned}$$

holds. Furthermore, we can calculate a nearest threshold graph of Q' by adding u as a universal vertex to the nearest threshold graph of Q. A skeleton of this graph can be obtained by adding u as a root to a skeleton of a nearest threshold graph of Q.

2. **u has the children c_i, where $i \in \{1, \ldots, n\}$ and $n \geq 2$.**

   Let Q_i be the quasi-threshold graph induced by $\text{Subtree}_{S(Q')}(c_i)$. For all Q_i we find $|Q_i| < n$. Hence, it follows with

   $$\text{Subtree}_{S(Q')}(v) = \text{Subtree}_{S(Q_i)}(v)$$

   for all $v \in V_{Q_i}$, where $i \in \{1, \ldots, n\}$, that the formula holds for all $v \in V_{Q'} \backslash \{u\}$. Now we show the formula for u.

   If all Q_i are single vertices, Q' is already threshold and we have

   $$\begin{aligned} D(u) = 0 &= \min_{c \in C(v)} \left(\left(\sum_{d \in C(v)} 0 \right) - 0 + 0 \right) \\ &= \min_{c \in C(v)} \left(\left(\sum_{d \in C(v)} \#\text{Edges}(d) \right) - \#\text{Edges}(c) + \text{Dist}(c) \right). \end{aligned}$$

   Otherwise, there is at least one Q_i that is not a single vertex. Consider a nearest threshold graph of the quasi-threshold graph $U := \bigcup_i Q_i$. A skeleton of a threshold graph consists of a caterpillar and isolated vertices. In a skeleton of a nearest threshold graph of U the vertices in the caterpillar all belong to one Q_i as we are only allowed to delete edges and no vertex $v \in V_{Q_i}$ is adjacent to a vertex $w \in V_{Q_j}$ for $i \neq j$. Hence, we have to choose as our 'caterpillar graph' the Q_i which causes the lowest costs. The costs are composed of the deletion distance of Q_i and the costs of isolating all other Q_j, i.e. deleting all their edges. Thus, the nearest threshold graph of U has the deletion distance

   $$d = \min_{r \in R} \left(\left(\sum_{r' \in R} \#\text{Edges}(r') \right) - \#\text{Edges}(r) + \text{Dist}(r) \right),$$

   where R is the set that contains all roots of the $\text{Subtree}_{S(Q')}(c_i)$. Since $R = C(u)$ and Q' has the same deletion distance as Q, it follows that the claim holds in this case as well. Furthermore, a nearest threshold graph of Q' can be obtained by adding u as a universal vertex to the nearest threshold graph of the lowest-cost Q_i and isolating the vertices of all the other Q_j with $i \neq j$. A skeleton of this tree can be obtained by adding u as a root to the skeleton of the lowest-cost threshold graph and attaching the vertices from the other subgraphs as leaves to u.

Note that in all cases a skeleton of a nearest threshold graph of Q' can be obtained by adding u as a root to the skeleton of the lowest-cost child and making all vertices in the other subgraphs (if there are any) leaves of u.

□

The proof does not only show the correctness of the formula, but it delivers an algorithm to compute a nearest threshold graph T of a quasi-threshold graph Q. For a given skeleton of a connected QTG we can perform a DFS on this tree. In this DFS we recursively calculate the measures $\#\text{Edges}(c)$, $\text{Dist}(c)$ and the size of $\text{Subtree}(c)$ for all children c of a vertex v. Using these measures we then calculate the measures for v. For each vertex v we store its lowest-cost child. In order to obtain the implicitly computed skeleton we traverse the quasi-threshold skeleton following the path that is given by the stored lowest-cost children, starting from the root. All children and (their descendants in the quasi-threshold skeleton) of a visited vertex v are made leaves of v except for its lowest-cost child. The following pseudo code 2.1 describes this algorithm in detail.

Algorithm 2.1: Threshold Deletion

Input: Array children containing all children of a vertex in the skeleton of a connected quasi-threshold graph Q with root r.

Output: Array parent containing each vertex's parent in a skeleton of a nearest threshold graph of Q. Dist(r) contains the deletion distance of Q.

```
   // Initialization
1  forall $v \in V_Q$ do
2      #Edges($v$) $\leftarrow 0$
3      Dist($v$) $\leftarrow 0$
4      sizeSubtree($v$) $\leftarrow 1$
5      parent($v$) $\leftarrow -1$
6      bestChild($v$) $\leftarrow \perp$

7  CALCULATE(vertex $v$)
8      $s, i, p$, sum $\leftarrow 0$
9      forall $u \in$ children($v$) do
10         CALCULATE($u$)
11         $s \leftarrow s +$ sizeSubtree($u$)
12         $i \leftarrow i +$ #Edges($u$)
13         sum $\leftarrow$ sum + #Edges($u$)
14     #Edges($v$) $\leftarrow i + s$
15     sizeSubtree($v$) $\leftarrow 1 + s$
16     $j \leftarrow \infty$
17     forall $u \in$ children($v$) do
18         if sum$-$#Edges($u$) + Dist($u$) $< j$ then
19             $j \leftarrow$ sum$-$#Edges($u$) + Dist($u$)
20             Dist($v$) $\leftarrow j$
21             bestChild($v$) $\leftarrow u$

22 CALCULATE($r$) // calculate measures
   // create skeleton
23 cur $\leftarrow r$
24 while cur $\neq \perp$ do
25     forall $v \in$ children(cur) and $v \neq$ bestChild(cur) do
26         forall $x \in$ subtree($v$) do
27             parent($x$) $\leftarrow$ cur
28     parent(bestChild(cur)) $\leftarrow$ cur
29     cur $\leftarrow$ bestChild(cur)
```

As we only traverse the skeleton two times, the complexity of this algorithm is linear in the number of vertices of Q. Note that this algorithm can even be sublinear in the number of deletions. The algorithm expects a skeleton of the input graph Q. It follows that the overall complexity is in $\mathcal{O}(|V_Q| + |E_Q|)$ as we have to compute a skeleton of Q first.

2.4 Quasi-Threshold to Threshold Editing

In this section we address the problem of threshold editing. First we will give an example of a quasi-threshold graph Q whose edit distance is smaller than its deletion distance.

The QTG Q consists of two connected components: a clique of size n and a star graph of size $2n$ with inner vertex c (see Figure 2.3). Its edit distance is n as we can insert an edge from c to each vertex in the clique obtaining a treshold graph T. In the context of threshold deletion we have to isolate one of the two connected componets in order to obtain a threshold graph. Since the clique has $n(n-1)/2$ and the star graph has $2n$ edges, the deletion distance of Q is $2n$ if $n \geq 5$. The difference between edit and deletion distance of graphs with this structure is n. Hence, there are graphs for which the deletion distance is twice as large as the edit distance.

Before giving an algorithm for this problem we prove some properties of nearest threshold graphs in the context of threshold editing.

2.4.1 Properties of Nearest Threshold Graphs

We will show that for every QTG Q with a skeleton $\mathrm{S}(Q)$ there is a nearest threshold graph T and a skeleton $\mathrm{S}(T)$ of T such that the quadruple $(Q, S(Q), T, S(T))$ has certain properties. Furthermore, we only consider connected QTGs as we can add a universal vertex to a QTG that is not connected preserving the edit distance (Lemma 2.7). The central result of this section is Theorem 2.10.

Theorem 2.10. *Let Q be a connected* QTG *and* $\mathrm{S}(Q)$ *be a skeleton of Q. There is a nearest threshold graph T (and a skeleton* $\mathrm{S}(T)$ *of this graph) such that the following holds:*

1. *A leaf in* $\mathrm{S}(Q)$ *remains a leaf in* $\mathrm{S}(T)$.
2. *If w is an inner vertex in* $\mathrm{S}(T)$*, then all ancestors v of w in* $\mathrm{S}(Q)$ *are also inner vertices in* $\mathrm{S}(T)$.
3. *For two inner vertices v, w of* $\mathrm{S}(Q)$ *and* $\mathrm{S}(T)$ *if v is an ancestor of w in* $\mathrm{S}(Q)$*, then the same holds for* $\mathrm{S}(T)$.

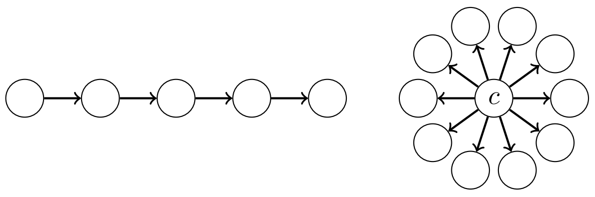


Figure 2.3: Skeleton of a graph Q whose edit distance (5) is smaller than its deletion distance (10).

Before proving this theorem, we want to illustrate it. Figure 2.4 shows a segment of a skeleton of a quasi-threshold graph Q. The leaves of the skeleton are coloured in blue. Let T be a nearest threshold graph of Q with skeleton $\mathrm{S}(T)$ such that propositions (1) - (3) from Theorem 2.10 hold.
Part (1) states that none of the blue coloured vertices can be an inner vertex of $\mathrm{S}(T)$. Part (2) says that if w is an inner vertex of $\mathrm{S}(T)$, then for the vertices $6, 5, v, 0$ the same holds true. The third part of this theorem is that if v and w are inner vertices of $\mathrm{S}(T)$, the vertex w is situated below v in the skeleton $\mathrm{S}(T)$. We will show Theorem 2.10 by using Lemma 2.11, 2.12, 2.13, 2.14 and 2.15, which we prove first.

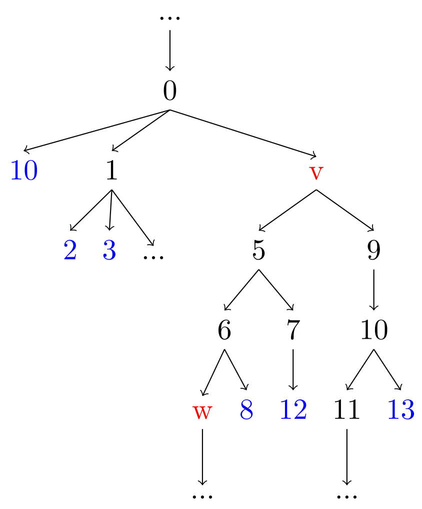


Figure 2.4: Segment of a skeleton of a QTG

Lemma 2.11. *Let Q be a quasi-threshold graph, T a threshold graph of Q with skeleton $\mathrm{S}(T)$ and $v, w \in V_Q$ vertices with $\mathrm{N}_Q(w) \subseteq \bar{\mathrm{N}}_Q(v)$ and $\mathrm{N}_T(v) \subseteq \bar{\mathrm{N}}_T(w)$.*
We denote the skeleton generated by a swap in the positions of v and w in $\mathrm{S}(T)$ by $\mathrm{S}(T')$. It holds that the edit distance from Q to the threshold graph T' induced by $\mathrm{S}(T')$ is at most as great as that to T.

Proof. We define the set X as $X := (\mathrm{N}_T(w) \backslash \mathrm{N}_T(v)) \backslash \{v, w\}$. The set X contains all vertices that are adjacent to w but not to v in T.

The graphs T and T' only differ in edges $\{v, x\}$ or $\{w, x\}$, where $x \in X$. All other edges are identical in both graphs. Now we compare the edit distance from Q to T' to that from Q to T. For the reasons mentionend above we only have to consider the edges $\{v, x\}$ and $\{w, x\}$, where $x \in X$.

By definition of X, for all $x \in X$ the edge $\{v, x\}$ is contained in T' but not in T and the edge $\{w, x\}$ is contained in T but not in T'. We are now accounting for the edits in which T and T' differ:

1. To get from Q to T we have to insert the edges from w to all vertices in $X \backslash \mathrm{N}_Q(w)$ and we have to delete all edges from v to all vertices in $X \cap \mathrm{N}_Q(v)$. Let t denote the number of these edits.

2. To get from Q to T' we have to insert the edges from v to all vertices in $X \backslash \mathrm{N}_Q(v)$ and we have to delete all edges from w to vertices in $X \cap \mathrm{N}(w)$. Let t' denote the number of these edits.

As $\mathrm{N}_Q(w) \subseteq \bar{\mathrm{N}}_Q(v)$ and v is not an element of X, it follows that $|X \backslash \mathrm{N}_Q(v)| \leq |X \backslash \mathrm{N}_Q(w)|$ and $|X \cap \mathrm{N}_Q(w)| \leq |X \cap \mathrm{N}_Q(v)|$. Therefore, we have

$$\begin{aligned} t - t' &= \Big(|X \backslash \mathrm{N}_Q(w)| + |X \cap \mathrm{N}_Q(v)|\Big) - \Big(|X \backslash \mathrm{N}_Q(v)| + |X \cap \mathrm{N}(w)|\Big) \\ &= \underbrace{\Big(|X \backslash \mathrm{N}_Q(w)| - |X \backslash \mathrm{N}_Q(v)|\Big)}_{\geq 0} + \underbrace{\Big(|X \cap \mathrm{N}_Q(v)| - |X \cap \mathrm{N}_Q(w)|\Big)}_{\geq 0} \\ &\geq 0. \end{aligned}$$

This shows that the edit distance from Q to T' is smaller than that from Q to T or has at most the same value. $\square$

Lemma 2.12. *Let Q be a QTG, T a nearest threshold graph of Q and $\mathrm{S}(T)$ a skeleton representation of T. All leaves of $v \in \mathrm{Inner}_{\mathrm{S}(T)}$ are elements of $\mathrm{N}_Q(v)$.*

Proof. Assume that $v \in Q$ is a vertex in $\mathrm{S}(T)$ that has a leaf u and $u \notin \mathrm{N}_Q(v)$. Thus, $\{v, u\}$ must have been added in order to obtain T. By omitting this edge we obtain a threshold graph T' whose distance to the input is smaller than the one of T. The graph T' is threshold because T has this property and omitting the edge $\{u, v\}$ either isolates u or u becomes a leaf of the parent p of v in $\mathrm{S}(T)$. Therefore, v cannot have any leaf that is not in $N_G(v)$. □

Lemma 2.13. *Let Q be a connected QTG and $\mathrm{S}(Q)$ a skeleton representation of this graph. There is a nearest threshold graph T and a skeleton $\mathrm{S}(T)$ such that any leaf in $\mathrm{S}(Q)$ remains a leaf in $\mathrm{S}(T)$.*

Proof. Let v be a leaf of $\mathrm{S}(Q)$. Assume that v is an inner vertex in $\mathrm{S}(T)$. Let B be the subtree of v in $\mathrm{S}(T)$ without its leaves. Note that B can be empty. We have to consider the following cases:

1. **v has a leaf u_j**
   The vertex v can only have leaves u_j with $u_j \in \mathrm{N}_Q(v)$ (see Lemma 2.12). Choose any one of these leaves – we choose u_1 – and interchange the position of v and u_1 in $\mathrm{S}(T)$, thereby generating a skeleton $\mathrm{S}(T')$ for a threshold graph T'. The edit distance of T' is at most as great as that of T(see Lemma 2.11). Note that u_1 cannot be a leaf in $\mathrm{S}(Q)$.

2. **v has no leaves**

   a) $B = \emptyset$
   This is not possible as v would be a leaf in $\mathrm{S}(T)$ in this situation.

   b) **v has no Q-neighbour in B**
   We delete all edges to the vertices in B. This way v becomes a leaf in $\mathrm{S}(T')$ or an isolated vertex if v is already the root of $\mathrm{S}(T)$. This saves $|B|$ edits.

   c) **All Q-neighbours of v are part of the central path in $\mathrm{S}(T)$**
   We insert v as a leaf under the lowest of its neighbours in the path in $\mathrm{S}(T)$, saving as many edits as there are non-Q neighbour leaves in B.

   d) **At least one Q-neighbour u is a leaf in B**
   As u is a Q-neighbour of v and v a leaf in $\mathrm{S}(Q)$, u is an inner vertex in Q and situated above v. Therefore we have $\mathrm{N}_Q(v) \subseteq \bar{\mathrm{N}}_Q(u)$. Since v is situated on the path from w to the root in $\mathrm{S}(T)$, it follows that $\mathrm{N}_T(u) \subseteq \bar{\mathrm{N}}_T(v)$. Thus, we can interchange u and v in $\mathrm{S}(T)$ with at most zero costs.

Either we have saved edits in this process or the edit distance of the generated tree T' is at most as great as that of T. In the former case T is not a nearest threshold graph to Q contradicting the assumption that T is such a graph, whereas in the latter case we have obtained a new nearest threshold graph T' and a skeleton $S(T')$ where the number of violations of the claimed property is reduced by one compared to T and $\mathrm{S}(T)$. Thus, we can reapply the procedure until all violations have been removed. □

The following lemma will help to prove Theorem 2.10 (2). Figure 2.5 illustrates the situation in this lemma.

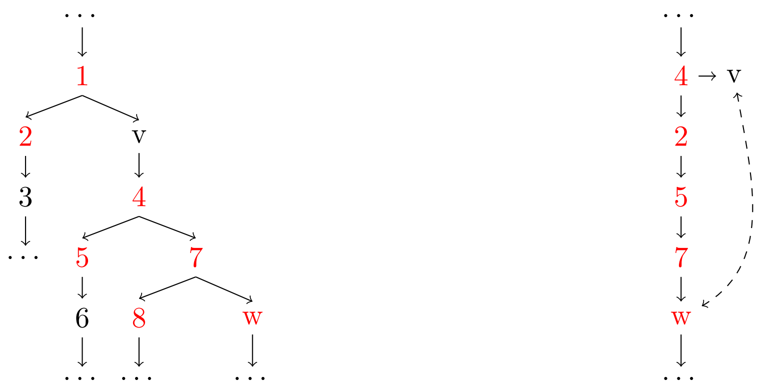


Figure 2.5: Skeleton of a QTG without leaves and corresponding skeleton of a threshold graph T without leaves except for v. Only red labelled vertices are inner vertices in both skeletons. The dashed arrow depicts the interchange that can occur in the proof of Lemma 2.14

Lemma 2.14. *Let Q be a QTG and $\mathrm{S}(Q)$ a skeleton of Q. There is nearest threshold graph T and a skeleton $\mathrm{S}(T)$ such that the following holds:*

1. *$(T, \mathrm{S}(T))$ fulfils Theorem 2.10 (1).*
2. *For two vertices $v, w \in V_G$ with $v, w \in \mathrm{Inner}_{\mathrm{S}(Q)}$ and $w \in \mathrm{Subtree}_{\mathrm{S}(Q)}(v)$ it holds that $w \in \mathrm{Inner}_{\mathrm{S}(T)}$ implies $v \in \mathrm{Inner}_{\mathrm{S}(T)}$.*

Proof. From Lemma 2.13 we know that there is a nearest $(T, \mathrm{S}(T))$ such that Theorem 2.10 (1) holds. We define the following set

$$A_T := \Big\{ v \mid v \in \big(\mathrm{Inner}_{\mathrm{S}(Q)} \setminus \mathrm{Inner}_{\mathrm{S}(T)}\big) \ \wedge\ \exists w \in \big(\mathrm{Subtree}_{\mathrm{S}(Q)}(v) \cap \mathrm{Inner}_{\mathrm{S}(T)}\big) \Big\}.$$

It contains all vertices v that are inner vertices in $\mathrm{S}(Q)$ but not in $\mathrm{S}(T)$ and for which exists a 'violation-partner' w such that the pair (v, w) contradicts claim (2).

$$B_T(v) := \Big\{ w \mid w \in \big(\mathrm{Inner}_{\mathrm{S}(Q)} \cap \mathrm{Inner}_{\mathrm{S}(T)}\big) \ \wedge\ w \in \mathrm{Subtree}_{\mathrm{S}(Q)}(v) \Big\}.$$

The set $B_T(v)$ contains all 'violation-partners' w of v.
Assume that A_T is not empty, let v be the vertex with minimal $\mathrm{Depth}_{\mathrm{S}(Q)}$ in A_T and w the vertex with maximal $\mathrm{Depth}_{\mathrm{S}(Q)}$ in $B_T(v)$ (see Figure 2.5).
With v being an ancestor of w in $\mathrm{S}(Q)$, we have that $\mathrm{N}_{\mathrm{S}(Q)}(w) \subseteq \bar{\mathrm{N}}_{\mathrm{S}(Q)}(v)$ and $\mathrm{N}_{\mathrm{S}(T)}(v) \subseteq \bar{\mathrm{N}}_{\mathrm{S}(T)}(w)$ because v is a leaf in $\mathrm{S}(T)$ whereas w is an inner vertex. Interchanging v and w in $\mathrm{S}(T)$ generates a skeleton $\mathrm{S}(T')$. We denote the threshold graph induced by $\mathrm{S}(T')$ by T'. We know from Lemma 2.11 that the edit distance from Q to T' is at most as great as that to T. Furthermore, it holds that $(T', \mathrm{S}(T'))$ fulfils Theorem 2.10 (1) as we did not make any leaf in $\mathrm{S}(Q)$ to an inner vertex in $\mathrm{S}(T')$. The case that the edit distance to T' is smaller than that to T cannot occur, as T is a nearest threshold graph of Q.

What we have shown so far is that we can interchange the vertices v and w while preserving the threshold property, equal editing distance and property (1). Now we prove that $A_{T'} \subseteq A_T \setminus \{v\}$. In order to do so we show that $A_{T'} \setminus A_T$ is empty.

$A_{T'} \setminus A_T$ is empty.

Assume that there is a $v' \in A_{T'} \setminus A_T$. Then there must be a w' such that (v', w') violates the claim. Hence, w' has to be a descendant of v' in $\mathrm{S}(Q)$ and an inner vertex in $\mathrm{S}(Q)$ and

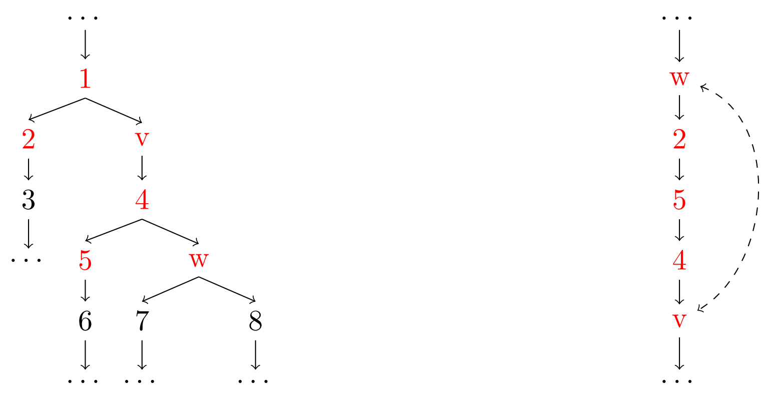


Figure 2.6: Skeleton of a QTG without leaves and corresponding skeleton of a threshold graph T without leaves. The dashed arrow depicts an interchange that can occur in the proof of Lemma 2.15.

$\mathrm{S}(T')$, whereas v' must be an inner vertex in $\mathrm{S}(Q)$ and a leaf in $\mathrm{S}(T')$. As we do not touch $\mathrm{S}(Q)$, either v' must have been an inner vertex in $\mathrm{S}(T)$ and is now a leaf (1) or w' must have been a leaf in $\mathrm{S}(T)$ and is now an inner vertex in $\mathrm{S}(T')$ (2).

1. **Situation (1) cannot occur.**
   The only vertex that has become a leaf is w. This implies $v' = w$. It follows that there has to be a w' in the subtree of w in $\mathrm{S}(Q)$ that is an inner vertex in $\mathrm{S}(T')$. But all $w_i \in \mathrm{Subtree}_{\mathrm{S}(Q)}(w)$ are leaves in $\mathrm{S}(T)$ (and so in $\mathrm{S}(T')$). Otherwise, $w_i \in B_T(v)$ would hold but this would contradict the maximal $\mathrm{Depth}_{\mathrm{S}(Q)}$ of w in $B_T(v)$. Hence, a pair (w, w_i) that does not fullfil the claim cannot exists in T'.

2. **Situation (2) cannot occur.**
   The only vertex that was a leaf in $\mathrm{S}(T)$ and is an inner vertex in $S(T')$ is v. Assume that $v = w'$. Then v' has to be an ancestor of v in $\mathrm{S}(Q)$. But all $v'' \in \mathrm{Ancestors}_{\mathrm{S}(Q)}(v)$ have to be inner vertices in $\mathrm{S}(T)$ and so in $\mathrm{S}(T')$. Otherwise, (v'', w) would have also been a violation of the claim in T what contradicts the minimal $\mathrm{Depth}_{\mathrm{S}(Q)}$ of v in A_T.

We have proved that neither (1) nor (2) can occur. Therefore, we can conclude that there is no pair (v', w') and $A_{T'} \backslash A_T$ is empty.

As v is an inner vertex in $\mathrm{S}(T')$, it cannot be part of $A_{T'}$. It follows that $A_{T'} \subseteq A_T \backslash \{v\}$.

Hence, we can reduce the set of violations successively and obtain a final threshold graph that has the claimed property. □

Note that the lemma above shows that there is always a nearest threshold T with a skeleton $\mathrm{S}(T)$ such that Theorem 2.10 (1) and (2) hold.

Lemma 2.15. *Let Q be a quasi-threshold graph and $\mathrm{S}(Q)$ a skeleton of Q. There is a nearest threshold graph T and a skeleton $\mathrm{S}(T)$ such that the following holds:*

1. *$(T, \mathrm{S}(T))$ fulfils Theorem 2.10 (1) and (2).*

2. *Let $v, w \in V_Q$ be inner vertices in $\mathrm{S}(Q)$ and $\mathrm{S}(T)$. If v is an ancestor of w in $\mathrm{S}(Q)$, the vertex v is also an ancestor of w in $\mathrm{S}(T)$.*

Proof. The general idea of this proof is – as in the previous proof – that we can remove all possible violations by iteratively interchanging two vertices in the threshold skeleton. We know from Lemma 2.14 that there is a nearest $(T, \mathrm{S}(T))$ such that Theorem 2.10 (1) and (2) hold. We define $I := \mathrm{Inner}_{\mathrm{S}(Q)} \cap \mathrm{Inner}_{\mathrm{S}(T)}$, which contains all vertices that are inner vertices in both $\mathrm{S}(Q)$ and $\mathrm{S}(T)$. Furthermore, we define

$$A_T := \left\{ v \mid v \in I \wedge \left(\exists w \in I : v \in \mathrm{Ancestors}_{\mathrm{S}(Q)}(w) \wedge w \in \mathrm{Ancestors}_{\mathrm{S}(T)}(v) \right) \right\}.$$

This set contains all vertices v for which a w exists such that the pair (v, w) does not fulfil the claimed property (2). For a vertex $v \in A_T$ we define

$$B_T(v) := \left\{ w \mid w \in I \wedge v \in \mathrm{Ancestors}_{\mathrm{S}(Q)}(w) \wedge w \in \mathrm{Ancestors}_{\mathrm{S}(T)} \right\}.$$

The set consists of all w such that (v, w) contradicts the claim. Note that all w are situated above v in $\mathrm{S}(T)$.

Assume that A_T is not empty. Then we choose $v \in A_T$ with minimal $\mathrm{Depth}_{\mathrm{S}(Q)}$ and $w \in B_T(v)$ with minimal $\mathrm{Depth}_{\mathrm{S}(T)}(w)$. It follows from Lemma 2.2 that $\mathrm{N}_Q(w) \subseteq \bar{\mathrm{N}}_Q(v)$ and $\mathrm{N}_T(v) \subseteq \bar{\mathrm{N}}_T(w)$. With Lemma 2.11 we can interchange v and w in $\mathrm{S}(T)$ generating a threshold graph T' with an edit distance from Q that is at most as great as those of T. Since T is a nearest threshold graph of Q, we have that the edit distances to T and T' are equal. There is a skeleton $\mathrm{S}(T')$ of T' that is equal to $\mathrm{S}(T)$ except for the interchange of v and w. As we do not make a leaf in $\mathrm{S}(Q)$ an inner vertex in $\mathrm{S}(T')$ and both v and w remain inner vertices in $\mathrm{S}(T')$, Theorem 2.10 (1) and (2) also holds for $(T', \mathrm{S}(T'))$ (see Figure 2.6).

We show that for all $x \in A_{T'} \backslash A_T$ it holds that $\mathrm{Depth}_{\mathrm{S}(Q)}(x) > \mathrm{Depth}_{\mathrm{S}(Q)}(v)$ and $v \notin A_{T'}$.

1. **v is not in $A_{T'}$.**
   As w has minimal depth in $B_T(v)$, all vertices w' above w in $\mathrm{S}(T)$ cannot be descendants of v in $\mathrm{S}(Q)$. We only interchange the positions of v and w, so this also holds for $\mathrm{S}(T')$. Since the position of v in $\mathrm{S}(T')$ is the old one of w, it follows that $v \notin A_{T'}$.

2. **For all $x \in A_{T'} \backslash A_T$ it holds that $\mathrm{Depth}_{\mathrm{S}(Q)}(x) > \mathrm{Depth}_{\mathrm{S}(Q)}(v)$.**
   Assume that there is an $x \in A_{T'} \backslash A_T$ with $\mathrm{Depth}_{\mathrm{S}(Q)}(x) \le \mathrm{Depth}_{\mathrm{S}(Q)}(v)$. There must be a y such that (x, y) is a violation of the claim. As we do not touch $\mathrm{S}(Q)$ but $x \notin A_T$, there are only two possibilities.
   (1) The vertices x or y are leaves in $\mathrm{S}(T)$ and inner vertices in $\mathrm{S}(T')$. This is not possible, however, as we only interchange two inner vertices and do not touch other vertices.
   (2) The relative positions of x or y have changed; more formally $\mathrm{Depth}_{\mathrm{S}(T)}(x) < \mathrm{Depth}_{\mathrm{S}(T')}(x)$ or $\mathrm{Depth}_{\mathrm{S}(T)}(y) > \mathrm{Depth}_{\mathrm{S}(T')}(y)$. The vertex w is the only vertex whose depth has increased. But from $\mathrm{Depth}_{\mathrm{S}(Q))}(x) \le \mathrm{Depth}_{\mathrm{S}(Q)}(v) < \mathrm{Depth}_{\mathrm{S}(Q)}(w)$ it follows that $x \neq w$.
   Thus, $\mathrm{Depth}_{\mathrm{S}(T)}(y) > \mathrm{Depth}_{\mathrm{S}(T')}(y)$ must be true. The only vertex that has decreased its depth is v. It follows that (x, v) is the only possible pair for a violation. But then (x, w) must have been a violation in $\mathrm{S}(T)$ as well, as x has not changed its position and v is now at the position of w. This contradicts the assumption that $x \notin A_T$.

It follows that we can delete v from the set A_T and only add vertices with greater depth in $\mathrm{S}(Q)$. This implies that the depth in $A_{T'}$ increases and thus reapplying this procedure will terminate as the maximal depth in $\mathrm{S}(Q)$ is finite. □

With Lemma 2.15 we have shown that for every $(Q, \mathrm{S}(Q))$ there is always a nearest threshold graph T with skeleton $\mathrm{S}(T)$ that fulfils Theorem 2.10 (1) - (3), which completes the proof of Theorem 2.10.

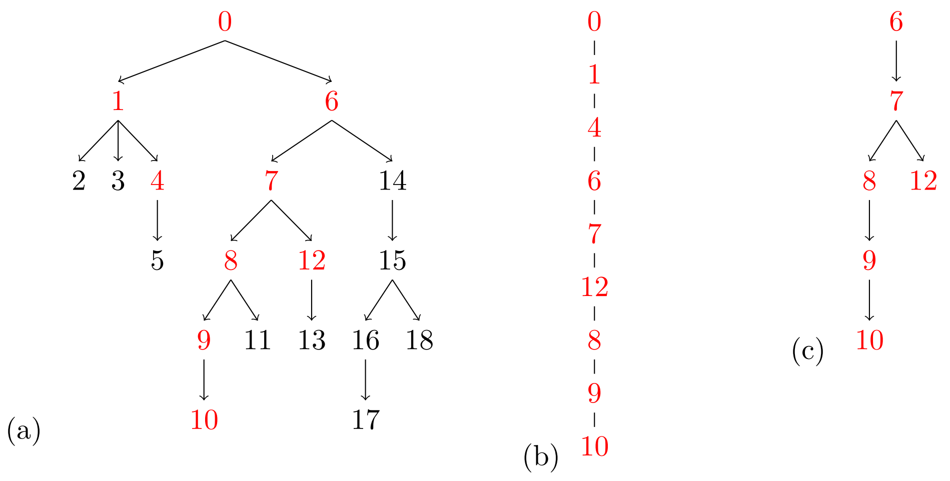


Figure 2.7: A skeleton $\mathrm{S}(Q)$ of a QTG Q without leaves and a skeleton $\mathrm{S}(T)$ of a nearest threshold graph T also without leaves. The red labelled vertices in (a) form $\mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$. Tree (c) is the subtree of vertex 6 in $\mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$.

Let $(Q, \mathrm{S}(Q))$ be a QTG with skeleton and $(T, \mathrm{S}(T))$ a nearest threshold graph such that Theorem 2.10 (1) - (3) are fulfilled. We define $\mathrm{Inner} := \mathrm{Inner}_{\mathrm{S}(Q)} \cap \mathrm{Inner}_{\mathrm{S}(T)}$, i.e. Inner contains exactly these vertices that are inner vertices in $\mathrm{S}(Q)$ and $\mathrm{S}(T)$. From property (2) it follows that the subgraph of $\mathrm{S}(Q)$ induced by Inner is a tree (see Figure 2.7). We call this tree the reduced tree of $\mathrm{S}(Q)$ and $\mathrm{S}(T)$, in short $\mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$.

Now we will show a last property of nearest threshold graphs $(T, \mathrm{S}(T))$ of $(Q, \mathrm{S}(Q))$ that fulfil Theorem 2.10 (1) - (3) before introducing an algorithm that computes them.

In the following we only consider $\mathrm{S}(T)$ without leaves, i.e. the central path of the caterpillar. Note that $\mathrm{S}(T)$ cannot have isolated vertices as Q is connected. We can divide the central path into intervals. Consider a vertex v and its child w in $\mathrm{S}(T)$. Both belong to the same interval if and only if w is a child of v in $\mathrm{S}(Q)$, too. Otherwise, w is the starting vertex of a new interval. In order to illustrate this term we consider the central path in Figure 2.7. There are three intervals: $I_1 = (0, 1, 4)$, $I_2 = (6, 7, 12)$ and $I_3 = (8, 9, 10)$.

The minimal length of these intervals is determined by the vertices above them in the central path. In order to specify this proposition we introduce for two inner vertices v, w of $\mathrm{S}(T)$, i.e. vertices on the central path of $\mathrm{S}(T)$, with $w \in \mathrm{Ancestors}_{\mathrm{S}(T)}(v)$ the measures $\mathrm{Score}(v, w)$, $\mathrm{Score}_{\max}(v)$ and $\mathrm{Score}_{\max}\mathrm{holder}(v)$. The following definition is more general as it defines these measures for any directed path that consists of inner vertices of $\mathrm{S}(Q)$.

Definition 2.16. *Let $(Q, \mathrm{S}(Q))$ be a connected* QTG *and P a directed path with $V_P \subseteq \mathrm{Inner}(\mathrm{S}(Q))$.*

1. $\mathrm{Score}(v, w)$
Let v, w be two vertices of P with $w \in \mathrm{Ancestors}_P(v)$, i.e. w is situated above v in the path. Let $p_{w,\mathrm{parent}(v)}$ be the path from w to the parent of v in P, i.e. $p_{w,\mathrm{parent}(v)} = (w, \ldots, \mathrm{parent}(v))$. The following illustrates the situation:

$$P = (\cdots, \mathrm{parent}(w), \underbrace{w, \cdots, \mathrm{parent}(v)}_{p_{w,\mathrm{parent}(v)}}, v, \cdots)$$

Then we define $\mathrm{Score}(v, w)$ as the number of vertices that are not Q-neighbours of v in $p_{w,\mathrm{parent}(v)}$ minus the number of v's Q-neighbours in $p_{w,\mathrm{parent}(v)}$.

Score(v, w)	‘10’	‘9’	‘8’	‘12’	‘7’	‘6’	‘4’	‘1’	‘0’	Score$_{\max}$	Score$_{\max}$holder
‘8’	-	-	-	1	0	-1	0	1	0	1	‘12’
‘6’	-	-	-	-	-	-	1	2	1	2	‘1’
‘0’	-	-	-	-	-	-	-	-	-	0	‘0’

Table 2.1: Values of Score and Score$_{\max}$ for some vertices in Figure 2.7 (b).

2. Score$_{\max}(v)$
   Let v be a vertex in P. The set A consists of all vertices w that are situated above v in P. We define

$$\text{Score}_{\max}(v) := \begin{cases} \max\limits_{w \in A} \left(\text{Score}(v, w)\right) & \textit{if } A \neq \emptyset \\ 0 & \textit{if } A = \emptyset \end{cases}$$

3. Score$_{\max}$holder(v)
   Score$_{\max}$holder(v) *is the first vertex w above v such that* Score$_{\max}(v) :=$ Score(v, w). *If there is no such w above v then we set* Score$_{\max}$holder$(v) := v$.

The idea behind these measures is that a vertex v with Score$_{\max}(v) > 1$ cannot have a leaf in the skeleton of a nearest threshold graph (see Lemma 2.18). Score$_{\max}$holder(v) is important because its parent can be chosen as parent of vertices that are leaves of v in S(Q) in this case (see Section 2.4.2).

To give an example, we calculate these measures for the starting vertices of the three intervals from Figure 2.7 (b). The results can be found in Table 2.1. Note that the vertex w with Score$(v, w) =$ Score$_{\max}(v)$ cannot be an ancestor of v in S(Q).
The proposition that we want to show is that the length of an interval with starting vertex v has to be greater than Score$_{\max}(v)$. The central path in Figure 2.7 (b) has this property as $|I_1| = 3 > 0 =$ Score$_{\max}$(‘0’), $|I_2| = 3 > 2 =$ Score$_{\max}$(‘6’) and $|I_3| = 3 > 1 =$ Score$_{\max}$(‘8’).

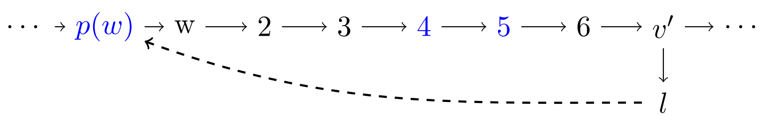
Theorem 2.17. *Let Q be a connected* QTG *with a skeleton* S(Q) *and T a nearest threshold graph with skeleton* S(T) *such that Theorem 2.10 (1) - (3) holds. Let I be an interval with starting vertex v in* S(T). *It holds that $|I| >$* Score$_{\max}(v)$.

In order to prove this theorem we use the following Lemma 2.18 (see Figure 2.8 for an illustration).

Lemma 2.18. *In the situation of Theorem 2.17 a vertex v' with* Score$_{\max}(v') > 1$ *cannot have a leaf in* S(T).

Proof. Let v' be a vertex with Score$_{\max}(v') > 1$. By definition of this measure we know that there is a vertex w above v' in the central path of S(T) with Score$(v, w) =$ Score$_{\max}(v')$. This vertex w is not a S(Q)-ancestor of v' and the parent of w in S(T) is a S(Q)-ancestor of v'; otherwise Score(v, w) would not be a maximal value. Note that there must be a parent(w) in S(T) as Q is connected and thus the root of S(T) is a universal vertex.

Assume that v' has a leaf l in S(T). We know from Lemma 2.12 that l must be in the neighbourhood of v' in Q. Hence, it is also a Q-neighbour of all $x \in$ Ancestors$_{S(Q)}(v')$. All these x are part of the central path of S(T) and situated above v' as a consequence of Theorem 2.10 (2) and (3). Since l is a Q-neighbour of v' and all ancestors of v' in S(Q) are situated above v' in S(T), l has to be a descendant of v' in S(Q). Therefore, we have



$\cdots \to p(w) \to w \to 2 \to 3 \to 4 \to 5 \to 6 \to v' \to \cdots$
l

Figure 2.8: The vertices ‘4’,‘5’ and $p(w)$ are $\mathrm{S}(Q)$-ancestors of v. Suppose that all vertices on the left of w are $\mathrm{S}(Q)$-ancestors of v. As $\mathrm{Score}_{\max}(v') = 2$ and $\mathrm{Score}(v', w) = 2$, vertex l becomes a leaf of $p(wY)$.

$\mathrm{N}_Q(l) \subseteq \bar{\mathrm{N}}_Q(v')$.
Let $p_{w,v'}$ be the path from w to v' in $\mathrm{S}(T)$, i.e. $p_{w,v'} = (w, \dots, v')$. If we want to make l a leaf of $\mathrm{parent}(w)$ in $\mathrm{S}(T)$, we have to delete all edges to Q-neighbours of l in $p_{w,v'}$ and do not have to insert edges to the vertices that are not Q-neighbours in $p_{w,v'}$. As all Q-neighbours of l in $p_{w,v'} - v'$ are $\mathrm{S}(Q)$-ancestors of v' and all non-Q-neighbours of l in $p_{v',w} - v'$ are not $\mathrm{S}(Q)$-ancestors of v' and $\mathrm{Score}_{\max}(v') > 1$, we know that the number of non-$\mathrm{S}(Q)$-neighbours of l in $p_{v',w} - v'$ minus the number of $\mathrm{S}(Q)$-neighbours in this path is at least two. Since we also have to delete the edge $\{l, v\}$, we can save at least one edit if we make l a leaf of $\mathrm{parent}(w)$. But this contradicts the assumption that T is a nearest threshold graph of Q. □

Recall the definition of the reduced tree $\mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$. It contains all vertices that are inner vertices in $\mathrm{S}(Q)$ and $\mathrm{S}(T)$. Now we consider the subtree of v in $\mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$. Figure 2.7 (c) is an example of such a subtree.
Let w be a vertex with depth $k < \mathrm{Score}_{\max}(v) - 1$ in this subtree. Therefore, there are k vertices above w in the subtree. As $\mathrm{S}(T)$ has the properties Theorem 2.10 (2) and (3), it follows that the only Q-neighbours of w on the path from v to $\mathrm{parent}(w)$ in $\mathrm{S}(T)$ are exactly those k vertices. Therefore, $\mathrm{Score}_{\max}(w)$ is reduced at most by k. Hence, we have

$$\mathrm{Score}_{\max}(w) \geq \mathrm{Score}_{\max}(v) - k > \mathrm{Score}_{\max}(v) - (\mathrm{Score}_{\max}(v) - 1) = 1.$$

This implies that such a vertex w cannot have a leaf in $\mathrm{S}(T)$. Now we can show Theorem 2.17.

Proof. Assume that there is an interval I with starting vertex v and $|I| \leq \mathrm{Score}_{\max}(v)$. We define $R := \mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$. We consider $\mathrm{Subtree}_R(v)$ (see Figure 2.9 (a)). It follows from Lemma 2.18 that no vertex with a depth smaller than $\mathrm{Score}_{\max}(v) - 1$ in this subtree can have a leaf in $\mathrm{S}(T)$. If a vertex with depth $\mathrm{Score}_{\max}(v) - 1$ has a leaf in $\mathrm{S}(T)$, we can make this leaf a leaf of the parent of w, where w is chosen to be a vertex with $\mathrm{Score}(v, w) = \mathrm{Score}_{\max}(v)$. This can be achieved using the same procedure as in Lemma 2.18 with zero cost. Thus, let us assume that no vertex with a depth of at most $\mathrm{Score}_{\max}(v) - 1$ in this subtree has a leaf in $\mathrm{S}(T)$. Note that this assumption does not change the set $\mathrm{Inner}_{\mathrm{S}(T)}$ and therefore does not change $\mathrm{Reduced}_{\mathrm{S}(Q),\mathrm{S}(T)}$. If there was a vertex of depth $\mathrm{Score}_{\max}(v) - 1$ that would become a leaf by moving its leaves up, then we could make this vertex itself a leaf of the parent of w saving an edit. But in this case T could not be a nearest threshold graph.

We have to consider the following two cases.

1. **A vertex $v' \in \mathrm{Subtree}_R(v)$ with a depth of $\mathrm{Score}_{\max}(v)$ or greater has a leaf in $\mathrm{S}(T)$.**

   Let v' be the vertex with minimal depth in $\mathrm{S}(T)$ among all such vertices.

   Claim: All vertices on the path $p_{v,\mathrm{parent}(v')} = (v, \dots, \mathrm{parent}(v'))$ in $\mathrm{S}(T)$ are ancestors of v' in $\mathrm{S}(Q)$.

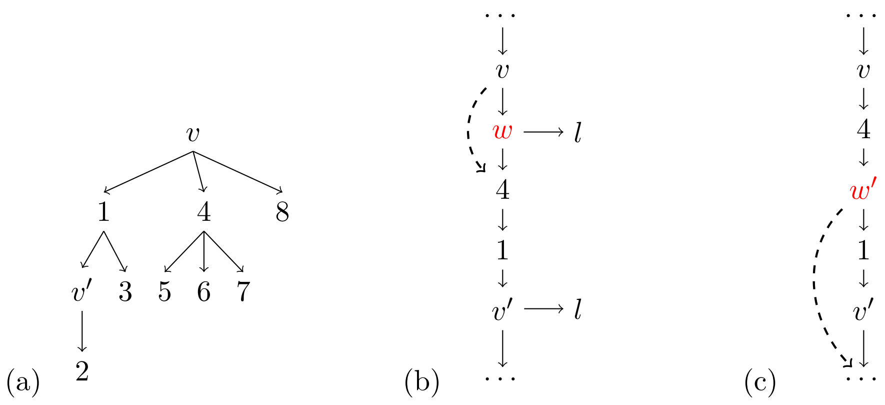


Figure 2.9: Illustration of the first case in the proof of Theorem 2.17. The subtree of v in $\text{Reduced}_{\text{S}(Q),\text{S}(T)}$ is shown in (a). In (b) and (c) w is not an ancestor of v' in $\text{S}(Q)$. Note that in this example $\text{Score}_{\max}(v)$ must be at least two.

Assume that there is a $w \in p_{v,\text{parent}(v')}$ that is not an ancestor of v'. We have to differentiate two cases, w does or does not have a leaf.

a) The vertex w has a leaf in $\text{S}(T)$. In this case it cannot be part of $\text{Subtree}_R(v)$ as this would contradict the minimality of v'. But then we can transfer all $x \in \text{Subtree}_R(v)$ that are situated above w in $\text{S}(T)$ to positions below this vertex as no vertex x has leaves in $\text{S}(T)$ (see Figure 2.9 (b)). This saves the insertion of the edges from the vertices x to the leaf of w. It follows that T is not a nearest threshold graph. Hence, such a vertex w cannot have a leaf in $\text{S}(T)$.

b) Vertex w does not have a leaf in $\text{S}(T)$. If there are multiple vertices w that are not Q-ancestors of v' in $p_{v,\text{parent}(v')}$, let w' be the vertex with a greatest depth in $\text{S}(T)$ among them. We can transfer w' to the position below v' (we have shown above that w' cannot have a leaf in $\text{S}(T)$) saving the insertion of the edges from w' to the leaves of v' (see Figure 2.9 (c)). This again contradicts the minimality of T.

As both cases lead to contradiction, the assumption that there is a vertex w that is not a $\text{S}(Q)$-ancestor of v' cannot be true.

From this observation follows that $p_{v,\text{parent}(v')} \cup \{v'\}$ forms an interval and therefore I contains at least $|p_{v,\text{parent}(v')}| + 1 \geq \text{Score}_{\max}(v) + 1$ vertices, which contradicts the assumption that $|I| \leq \text{Score}_{\max}(v)$.

2. **No $v' \in \text{Subtree}_R(v)$ has a leaf in $\text{S}(T)$.**
Let $v' \in \text{Subtree}_R(v)$ be the vertex with maximal $\text{Depth}_{\text{S}(T)}$ among all vertices in $\text{Subtree}_R(v)$. It follows that there are no vertices $x \in N_Q(v')$ below v' in $\text{S}(T)$. Yet there must be another inner vertex y of $\text{S}(T)$ below v' in $\text{S}(T)$, or otherwise v' would be a leaf in $\text{S}(T)$ and thus $v' \notin \text{Subtree}_R(v)$. Then we make v' a leaf of the next vertex v'' above v' in $\text{S}(T)$, where v'' is an ancestor of v' in $\text{S}(Q)$. This saves the insertion of the edge to y and its descendants in $\text{S}(T)$. But this contradicts the minimality of T. So this case cannot occur.

It follows that $|I| \leq \text{Score}_{\max}(v)$ cannot be true. Note that the proof even shows that in an interval at least one vertex must have a leaf. □

2.4.2 Exact Algorithm

Now we want to outline an algorithm that computes for every QTG Q a nearest threshold graph T. More precisely: we take a skeleton $\mathrm{S}(Q)$ of Q as input and compute a skeleton $\mathrm{S}(T)$ of T. First of all we know from Lemma 2.7 that we can add a universal vertex to a QTG without changing its editing distance. Therefore, we will only consider connected QTGs. This implies that the skeleton of the nearest threshold graph of the input is a caterpillar (it cannot have isolated vertices as the input graph is connected). A caterpillar is a sequence S of vertices and a set of vertices that are leaves in the sequence S.

Consider the following problem: We are given a sequence S of vertices that form the central path of a caterpillar. Now we shall compute the positions of all leaves $l \in V_Q \backslash S$ within the sequence so that the threshold graph induced by this caterpillar has minimal editing distance from Q among all threshold graphs that are induced by a caterpillar with exactly this sequence as central path. We will show later that under certain conditions this problem can be solved in linear time in the size of the sequence.

The idea of the algorithm is to enumerate all possible sequences S, to calculate the best positions of the remaining vertices as leaves and to keep a skeleton whose induced threshold graph has minimal editing distance to Q. A sequence S can consist of up to $n := |V_Q|$ vertices. There are $\binom{n}{i}$ possibilities to choose a subset of i vertices out of a set of n vertices and we can order these i vertices in $i!$ different ways. It follows that the number of all possible sequences is

$$\sum_{i=1}^{n} \binom{n}{i} \cdot i!, \; n = |V_Q|.$$

This is far too big for any reasonable computation. But we can restrict the number of possible sequences. We have shown that for every $(Q, \mathrm{S}(Q))$ there is a nearest threshold graph T with a skeleton $\mathrm{S}(T)$ so that Theorem 2.10 (1) - (3) holds. We try to find such a threshold graph or, more precisely, its skeleton $\mathrm{S}(T)$. The central path of this skeleton cannot contain leaves of $\mathrm{S}(Q)$ (Theorem 2.10 (1)). Therefore, we only have to consider the subgraph of $\mathrm{S}(Q)$ that is induced by its inner vertices. Furthermore, the vertices in the central path of $\mathrm{S}(T)$ have to fulfil the properties Theorem 2.10 (2) and (3), from which follows that the sequence of these vertices must consist of intervals of coherent vertices in $\mathrm{S}(Q)$. This allows us to perform a kind of DFS to find all such sequences. The following lemma summarizes the properties that a possible sequence for the central path of $\mathrm{S}(T)$ must have.

Corollary 2.19. *Let* $(Q, \mathrm{S}(Q))$ *be a connected* QTG *and* $(T, \mathrm{S}(T))$ *a nearest threshold graph of Q such that Theorem 2.10 (1) - (3) holds. A possible sequence* seq *of vertices for the central path of* $\mathrm{S}(T)$ *must have the following properties:*

1. *It does not contain any leaves of* $\mathrm{S}(Q)$*.*
2. *If w is an element of* seq*, then all ancestors v of w in $S(Q)$ are also in* seq*.*
3. *Let v, w be elements of* seq*. If v is an ancestor of w in* $\mathrm{S}(Q)$*, the same holds in* seq*.*
4. *Let I be an interval of* seq *with starting vertex v. It holds that* $|I| > \mathrm{Score}_{\max}(v)$*.*

We call a sequence that has all these properties valid*.*

Proof. This corollary follows directly from Theorem 2.10 and Theorem 2.17. □

2.4.2.1 Enumerating-Algorithm

Here we present and prove the correctness of Algorithm 2.2 that enumerates all valid sequences. Since we do not have to consider the leaves of $\mathrm{S}(Q)$ in order to find the correct sequence, the algorithm operates on the subtree of $\mathrm{S}(Q)$ induced by its inner vertices. We denote this subtree by $\mathrm{S(Q)}_{\mathrm{inner}}$. The algorithm works recursively. Each function call has three arguments: v, seq and super. The list seq contains the current sequence, super all vertices in $\mathrm{S(Q)}_{\mathrm{inner}}$ that are adjacent to a vertex in the current sequence in $\mathrm{S(Q)}_{\mathrm{inner}}$ but not contained in the latter and v is the vertex that is added to the sequence seq during the current function call. In order to keep super correct, we delete v from this list and insert all children of v into it. If the new sequence is valid, the algorithm prints the sequence. If super is not empty, we call the function successively on all vertices in super. In the case that the sequence is not valid, i.e. the last interval in it is too short, we call the function only on children of v rather than on all vertices in super. In each function call first we test whether the longest path in $\mathrm{Subtree}_{\mathrm{S(Q)}_{\mathrm{inner}}}(v)$ is greater than $\mathrm{Score}_{\mathrm{max}}(v)$. Otherwise we return (see Corollary 2.19 (4)).

After all subsequent calls have returned, seq and super are restored to the state they had before entering the current function call. Then the function returns. The initial function call is made with the root of $\mathrm{S}(Q)$ as its first argument and two empty lists for seq and super.

In order to prove the correctness of Algorithm 2.2 we use the following Lemma 2.20.

Lemma 2.20. *Let* `calculate`$(v, \mathrm{seq}, \mathrm{super})$ *be a subsequent function call of* `calculate` $(r, \perp, \perp)$. *Then the following holds: When the call has executed line 30*

1. $\mathrm{scoreMax}(v)$ *and* $\mathrm{scoreMaxHolder}(v)$ *are correct for all* $v \in \mathrm{seq}$, *i.e. they equal* $\mathrm{Score}_{\mathrm{max}}(v)$ *and* $\mathrm{Score}_{\mathrm{max}}\mathrm{holder}(v)$, *respectively.*
2. position *contains the correct index (starting at zero) for each vertex v in* seq.
3. super *contains exactly those vertices that are adjacent to a vertex in* seq *but that are not contained in the sequence.*
4. *The sequence in* seq *fulfils the properties of Corollary 2.19 (1) - (3).*

Proof. We prove this by a structural induction on the depth of a call on the stack. Before beginning, we want to point out an important detail. During the execution of the algorithm the parameters seq and super are used in subcalls of `calculate` (lines 33 and 37). These two lists have the same value after returning from the subcall as before its execution. This holds because every subcall that modifies the lists undoes its changes of the two lists in lines 38 - 41. Also, the entries in the arrays scoreMax, scoreMaxHolder and position belonging to vertices in the current sequence do not change as the algorithm does not touch entries pertaining to vertices that are in seq and the sequence only grows in subsequent calls.

1. **Base case** $n = 1$
   In this case `calculate`$(v, \mathrm{seq}, \mathrm{super})$ is directly called by `calculate`$(r, \perp, \perp)$ in line 37. Therefore, super contains all children of r in $\mathrm{S(Q)}_{\mathrm{inner}}$ and seq only contains r. As $v \neq r$, f is set to zero ($\mathrm{position}(r) = 0$). Thus, we get $\mathrm{scoreMax}(v) = \mathrm{scoreMax}(r) - 1 = -1$ and $\mathrm{Score}_{\mathrm{max}}\mathrm{holder}(v) = \mathrm{parent}(v) = r$. The algorithm proceeds to line 26, inserts all children of v in super and deletes v from super. Hence, super now contains all vertices adjacent to $r - v$ in $\mathrm{S(Q)}_{\mathrm{inner}}$ and none of the vertices in seq. Furthermore, $\mathrm{position}(v)$ is set to 1 and seq contains r and v. Consequently, all claims of the lemma are fulfilled.

Algorithm 2.2: ENUMERATION OF ALL VALID SEQUENCES

Input: Array children containing all children of a vertex in a skeleton $S(Q)$ of a QTG Q with root vertex r.

Output: All sequences that have to be considered

```
forall $v \in S(Q)_{inner}$ do
    position($v$) ← $-1$
    scoreMax($v$) ← $0$
    scoreMaxHolder($v$) ← $r$
    inner($v$) ← INNER_VERTICES($v$, children) // returns the children of $v$ that are inner vertices
    longestPath($v$) ← CALC_LONGEST_PATH($v$, inner) // returns the length of the longest path in the subgraph of $S(Q)_{inner}$ with $v$ as root
CALCULATE($r$,⊥,⊥)
// definition of the function CALCULATE
CALCULATE(vertex $v$, list seq, list super)
    if $v \neq r$ then
        $f$ ← (seq.SIZE $- 1$) $-$ position(parent($v$))
        if $f = 0$ then
            scoreMax($v$) ← max(scoreMax(parent($v$)) $- 1, -1$)
            if scoreMax($v$) $= -1$ then
                scoreMaxHolder($v$) ← parent($v$)
            else
                scoreMaxHolder($v$) ← scoreMaxHolder(parent($v$))
        else
            if scoreMax(parent($v$)) $- 1 > 0$ then
                scoreMax($v$) ← scoreMax(parent($v$)) $- 1 + f$
                scoreMaxHolder($v$) ← scoreMaxHolder(parent($v$))
            else
                scoreMax($v$) ← $f$
                scoreMaxHolder($v$) ← seq(position(parent($v$) + 1))
    if longestPath($v$) $\leq$ scoreMax($v$) then
        return
    forall $u \in$ inner($v$) do
        super.INSERT($u$) // insert $u$ directly behind $v$
    super.DELETE($v$)
    position($v$) ← seq.SIZE()
    seq.INSERT($v$)
    if scoreMax($v$) $> 0$ then
        forall $c \in$ inner($v$) do
            CALCULATE ($c$, seq, super)
    else
        PRINT (seq) // print current sequence
        forall $n \in$ super do
            CALCULATE ($n$, seq, super)
    seq.DELETE($v$)
    super.INSERT($v$) // insert $v$ after its former predecessor in super
    forall $u \in$ inner($v$) do
        super.DELETE($u$)
```

2. **Induction hypothesis**
   The claim holds for all calls with a depth on the stack of at most n.

3. **Inductive step**
   Let c be a subsequent function call of c' that has a depth of n. Furthermore, let seq′ and super′ be the states of seq and super directly after line 30 in call c'. As $v \neq r$, scoreMax(v) and scoreMaxHolder(v) are computed in lines 9 - 23. The list seq′ fulfils the claimed properties as the corresponding function call c' has depth n. From property (4) it follows that the subgraph of $\mathrm{S(Q)}_{\mathrm{inner}}$ induced by the vertices in seq′ is a tree. In combination with (3) this implies that the parent of v in $\mathrm{S(Q)}_{\mathrm{inner}}$ must be part of seq′. Otherwise, v could not be part of super′.
   As seq′ fulfils Corollary 2.19 (2), the number of elements between parent(v) and v in seq′ $+v$ that are not ancestors of v in $\mathrm{S(Q)}_{\mathrm{inner}}$ is

   $$(\mathrm{seq}'.\mathtt{size}() - 1) - \mathrm{position}(\mathrm{parent}(v)).$$

   If $f = 0$, i.e. v will be added directly after its parent in the sequence, then we have $\mathrm{Score}_{\max}(v) = \max(\mathrm{Score}_{\max}(\mathrm{parent}(v)) - 1, -1)$, as either we have to consider not-Q-neighbours before parent(v) or not. If $\mathrm{Score}_{\max}(v) = -1$, clearly, $\mathrm{Score}_{\max}\mathrm{holder}(v) = \mathrm{parent}(v)$ holds. Otherwise we have $\mathrm{Score}_{\max}\mathrm{holder}(v) = \mathrm{Score}_{\max}\mathrm{holder}(\mathrm{parent}(v))$.

   If $f > 0$ we have to differentiate two cases:

   a) $\mathrm{Score}_{\max}(\mathrm{parent}(v)) - 1 > 0$
      In this case $\mathrm{Score}_{\max}(v) = \mathrm{Score}_{\max}(\mathrm{parent}(v)) - 1 + f$ holds, as we have to take into account the vertices that are not Q-neighbours of parent(v) (and therefore not Q-neighbours of v) before parent(v) in the sequence. This also implies that $\mathrm{Score}_{\max}\mathrm{holder}(v) = \mathrm{Score}_{\max}\mathrm{holder}(\mathrm{parent}(v))$.

   b) $\mathrm{Score}_{\max}(\mathrm{parent}(v)) - 1 \leq 0$
      In this case the highest value of $\mathrm{Score}(v, \cdot)$ is obtained if we only consider the f not-Q-neighbours of v that are situated after parent(v) in the sequence. Thus, we get $\mathrm{Score}_{\max}(v) = f$ and $\mathrm{Score}_{\max}\mathrm{holder}(v)$ is the first one of these vertices in the sequence.

   It follows that the computation of scoreMax(v) and scoreMaxHolder(v) is correct, i.e. $\mathrm{scoreMax}(v) = \mathrm{Score}_{\max}(v)$ and $\mathrm{scoreMaxHolder}(v) = \mathrm{Score}_{\max}\mathrm{holder}(v)$.

   If $\mathrm{longestPath}(v) \leq \mathrm{scoreMax}(v)$ the call returns. In this case the claim trivially holds as we do not reach line 30. Otherwise, we insert all children of v into super and delete v. Thus, (3) is true for seq′ $+v$. As position(v) is set before inserting v in seq′, this value is also correct. Hence, the claimed properties (1) - (3) hold for c. We have added v to seq′ and, as stated above, the parent of v in $\mathrm{S(Q)}_{\mathrm{inner}}$ is part of seq′. Thus, seq′ $+v$ also fulfils Corollary 2.19 (1) - (3) and therefore property (4).

   This shows that the claimed properties also hold for a call of depth $n + 1$.

□

Using Lemma 2.20 we will show that Algorithm 2.2 enumerates all valid sequences.

Theorem 2.21. *Algorithm 2.2 enumerates all valid sequences, i.e. sequences that fulfil the properties Corollary 2.19 (1) - (4).*

Proof. We prove this theorem by induction on the length of a valid sequence.

- **Base case $n = 1$**
  The only valid sequence that has length one is the root itself. Consider the function call `calculate`$(r, \perp, \perp)$. As scoreMax(r) is zero but longestPath(r) is at least one, we proceed to line 34 and print this sequence.

- **Induction hypothesis**
  The algorithm enumerates all valid sequences of length at most n.

- **Inductive step**
  Let S be a valid sequence with a length of $n + 1$. At first, we consider the case that the sequence S' consisting of the first n vertices of S is a valid sequence. Let v be the last vertex in S, i.e. the vertex that is not contained in S'.

  Let c' be the function call of `calculate` that prints S'. With seq$'$ and super$'$ we denote the state of seq and super after processing line 30 in c'. As S is a valid sequence, v cannot be the starting vertex of a new interval. Therefore, its parent in S is also its parent in $\mathrm{S(Q)}_{\mathrm{inner}}$ and v is part of super$'$ (Lemma 2.20 (3)). Thus, there is a call c of `calculate` in line 37 with v as its first argument. From Lemma 2.20 we know that scoreMax(v) is correct for seq$' + v = S + v$ and because S is a valid sequence scoreMax$(v) < 0$ must hold. Note that otherwise the last interval in S would be too short. Therefore, we proceed to line 34 in call c and print seq$' + v = S$.

  Now we consider the case that $S - v$ is not a valid sequence, where v is the last vertex in S. We have $(u_1, u_2, \ldots, u_n, v) = S$. There must be an $i > 0$ such that $(u_1, \ldots, u_i)$ is a valid sequence. Let S' be $(u_1, \ldots, u_i)$ with maximal i such that S' is valid. Then u_{i+1} is the starting vertex of the interval in sequence S that contains v. Let c' be the function call that prints S' and seq$'$ and super$'$ be the states of seq and super directly after line 30 in c'. The sequence S' is valid, so the subgraph of $\mathrm{S(Q)}_{\mathrm{inner}}$ induced by the vertices in the sequence must be a tree t. Sequence S is also valid and the tree induced by its vertices contains t. As the vertices $(u_{i+1}, \ldots, u_n, v)$ form an interval in S, they have to be a path in $\mathrm{S(Q)}_{\mathrm{inner}}$. Therefore, u_{i+1} must be adjacent (in $\mathrm{S(Q)}_{\mathrm{inner}}$) to one of the vertices in t. It follows that u_{i+1} is contained in super$'$ and that there is a call c of `calculate` with u_{i+1} as its first argument in line 37.
  The values of $\mathrm{Score}_{\max}(u_j)$, where $i < j \leq n$, are all positive as it holds that $\mathrm{Score}_{\max}(u_j) = \mathrm{Score}_{\max}(v) + n - j + 1$ and $\mathrm{Score}_{\max}(v) = 0$. Since S is a valid sequence, longestPath$(v) \geq 1$ and $\mathrm{Score}_{\max}(v) = 0$, we have that longestPath$(u_{i+1}) >$ $\mathrm{Score}_{\max}(u_j)$. Therefore, we pass the test in line 25 and proceed to line 33 (recall that $\mathrm{Score}_{\max}(\cdot) = \mathrm{scoreMax}(\cdot)$ as stated in Lemma 2.20). Here `calculate` with u_{i+2} as its first argument is called. This process continues until the function call with u_n as its first argument calls `calculate`$(v, \ldots)$. The value of $\mathrm{Score}_{\max}(v)$ in S is zero, therefore the same holds for scoreMax(v). Thus, we proceed to line 34 in which S is printed.

$\square$

We have shown that Algorithm 2.2 enumerates all valid sequences. Moreover, it enumerates only valid sequences. We want to justify why this is true.
An illegal sequence fails to fulfil either Corollary 2.19 (1) - (3) or (4). As the algorithm only works on inner vertices of $\mathrm{S}(Q)$ and as super only contains vertices whose parents are in seq, the propositions (1) - (3) are fulfilled. In each step the last interval in the current sequence is extended and not printed if $\mathrm{Score}_{\max}(w) > 0$ (w is the last vertex in the current sequence). Hence, for a sequence that is printed in line 35 an ending vertex w of an interval I has a $\mathrm{Score}_{\max}$ of at most zero. This implies that $|I| \geq \mathrm{Score}_{\max}(v)$ for the starting vertex v of an interval I.

The following Lemma 2.22 helps to determine an upper bound for the number of calls of function `calculate` during the execution of Algorithm 2.2.

Lemma 2.22. *During the execution of the algorithm for any two function calls* `calculate`$(v, \mathrm{seq}, \mathrm{super})$ *and* `calculate`$(v', \mathrm{seq}', \mathrm{super}')$*, it holds that either $v \neq v'$ or* $\mathrm{seq} \neq \mathrm{seq}'$*.*

Proof. Assume that there are two function calls c_n and $c'_{n'}$ of `calculate` with $v = v'$ and $\mathrm{seq} = \mathrm{seq}'$ during the execution of the algorithm. By c_{i-1} we denote the function call that has invoked call c_i. We find $n = n'$ as the depth of a call equals the size of its second argument and we have $\mathrm{seq} = \mathrm{seq}'$. It follows directly that the first and the second argument of the calls c_{n-1} and c'_{n-1} are equal as the second argument is modified only in line 30 before invoking a subsequent call. Inductively, it follows that this must be true for every pair (c_i, c'_i), where $i \in \{0, \cdots, n-1\}$. Note that $c_0 =$ `calculate`$(r, \perp, \perp)$. Since in `calculate` only one subsequent call is invoked per vertex v as first argument, it holds that $c_i = c'_i$ for every $i \in \{0, \cdots n\}$. This contradicts the assumption $c_n \neq c'_n = c'_{n'}$. □

There are $\binom{n}{i+1} \cdot (i+1)!$ different ways to choose a sequence of length i and a single vertex from a set of n vertices. In combination with Lemma 2.22 follows that there can be at most

$$\sum_{i=1}^{n} \binom{n}{i} \cdot i!$$

different function calls of `calculate` during the execution of the algorithm, where n is the number of inner leaves of the input skeleton. Note that this number is not a sharp upper bound.

2.4.2.2 Leaf-Positioning-Algorithm

As mentioned in the introduction of this subsection, we need to calculate the best position of the leaves for a given sequence S. The leaves are all vertices of Q that are not in S. Algorithm 2.3 solves this task for a valid sequence S. The idea of this algorithm is the following: it iterates over the sequence beginning at its end. For each vertex v the algorithm calculates its potential leaves in the skeleton induced by S. These are all leaves of v in $\mathrm{S}(Q)$ and vertices in the subtrees of children of v that are not part of the sequence. Then, based on the measure $\mathrm{Score}_{\max}(v)$, the algorithm decides whether these vertices should be made leaves of v or of the parent of $\mathrm{Score}_{\max}\mathrm{holder}(v)$ in S.

The algorithms's complexity is in $\mathcal{O}(|S| + |V_Q \backslash S|) = \mathcal{O}(|V_Q|)$. The summand $|V_Q \backslash S|$ is due to the fact that the algorithm calculates a best position for every $v \in V_Q \backslash S$. Thus, we have to touch every vertex in $V_Q \backslash S$. If we are only interested in the editing costs of a nearest threshold graph T with sequence S as its central path, there is no need to know the position of each leaf in $\mathrm{S}(T)$. In this case we can replace leavesSeq by an array that only contains the number of leaves for each vertex in S. This avoids the costly operations in lines 10 and 15. Instead of copying all leaves we only have to add their number. This can be performed in $\mathcal{O}(1)$. Thus, the running time of this version is in $\mathcal{O}(|S|)$.

Theorem 2.23. *Given a valid sequence S, Algorithm 2.3 computes a skeleton $S(T)$ of a threshold graph that fulfils Theorem 2.10 (1) - (3) and that has a minimal editing distance among all threshold graphs induced by skeletons whose central path is S.*

Let S be a valid sequence and $\mathrm{S}(T)$ the skeleton of a nearest S-induced threshold graph of Q, i.e. a threshold graph induced by a skeleton with S as central path that has a minimal edit distance from Q among all threshold graphs induced by skeletons with S as a central

Algorithm 2.3: Leaf-Positioning

```
Input:
Array seq, containing a valid sequence of vertices.
Array children, containing all children of a vertex in the skeleton $S(Q)$ of a QTG $Q$.
Array depth, containing the depth of a vertex in $S(Q)$.
Array #edges, containing the costs for isolating the subtree of a vertex in $S(Q)$.
Array subtree, containing the vertices in the vertex's subtree in $S(Q)$.
Arrays scoreMax and scoreMaxHolder containing $\text{Score}_{\max}$ and $\text{Score}_{\max}$holder for
all vertices in seq.
Output: Leaves of each vertex in sequence seq. Editing costs in del + adds
1  del $\leftarrow 0$ // number of deleted edges
2  adds $\leftarrow 0$ // number of inserted edges
3  for $v \in$ seq do
4      leavesSeq($v$) $\leftarrow \emptyset$
5  for $n \leftarrow$ seq.SIZE() $- 1$ to $0$ do
6      $v \leftarrow$ seq($n$)
7      for $c \in$ children($v$) do
8          if $c \notin$ seq then
9              del $\leftarrow$ del+#edges($c$) // add isolation costs for this subtree
10             leavesSeq($v$) $\leftarrow$ leavesSeq($v$) $\cup$ subtree($v$) // add vertices in this
                   subtree
11     if scoreMax($v$) $> 1$ then
12         pos $\leftarrow$ position(scoreMaxHolder($v$)) $- 1$ // position of future parent
13         $p \leftarrow$ seq(pos)
14         nbAncestor $\leftarrow$ depth($v$) $-$ depth($p$)
15         leavesSeq($p$) $\leftarrow$ leavesSeq($p$) $\cup$ leavesSeq($v$)
16         del $\leftarrow$ del + nbAncestor $\cdot$|leavesSeq($v$)|
17         leavesSeq($v$) $\leftarrow \emptyset$
18     adds $\leftarrow$ adds + ($n -$ depth($v$)) $\cdot$ (|leavesSeq($v$)| + 1)
```

path. Let l be a leaf in $S(T)$, i.e. l is part of the set $V_Q \backslash S$. In the following we analyse the set of edges that have to be deleted or inserted in order to obtain T. All edges $\{v, w\}$ in Q that are not incident to a vertex in S must be deleted. Otherwise, v would be an ancestor of w in $S(T)$ or vice versa, but this is not possible as $v, w \notin S$ and leaves cannot be ancestors of other vertices. Edges that are inserted also have to be incident to a vertex in S for the same reason. But this implies that positioning a leaf within the sequence is completely independent from the positioning of any other leaves.

Hence, iterating over all $l \in V_Q \backslash S$ and assigning them to a parent such that the number of edits incident to l is minimal yields a skeleton of a nearest S-induced threshold graph. In order to find an optimal position for a leaf l, we define the following measure:

Definition 2.24. *Let S be a valid sequence of $S(Q)$ and $l \in V_Q \backslash S$. We denote the first vertex in S by r and its last vertex by w. For every v in S we define the two sets $p_1(v)$, which contains v, and all ancestors of v in S and $p_2(v)$, which contains all descendants of v in S.*

$$S = (\underbrace{r, \ldots, \text{pred}(v), v}_{p_1(v)}, \underbrace{\text{succ}(v), \ldots, w}_{p_2(v)})$$

Note that $p_2(v)$ is empty if v is the last vertex in S.
Let $\text{cost}(v, l)$ be the number of vertices in $p_1(v)$ that are not Q-neighbours of l plus the number of Q-neighbours in $p_2(v)$.

It is clear that $\text{cost}(v, l)$ is the number of edits that have to be performed in order to insert l as a leaf of v in the sequence (ignoring the deletions incident to l that have been necessary for isolating l from the other leaves, but these cannot be omitted). Thus, inserting every $l \in V_Q \backslash S$ as a leaf of a vertex v with a minimal value of $\text{cost}(v, l)$ among all vertices in S results in a skeleton of a nearest S-induced threshold graph of Q.

As we only consider valid sequences S, for every leaf $l \in V_Q \backslash S$ there is a $\text{S}(Q)$-ancestor of l in S. The vertex with maximal depth in S among all these ancestors of v plays an important role in the positioning process of l.

Lemma 2.25. *Let S be a valid sequence of $\text{S}(Q)$ and l be an element of $V_Q \backslash S$. Let v be the $\text{S}(Q)$-ancestor of l with maximal depth in S. It holds that $\text{cost}(v, l) - \text{cost}(v', l) = \text{Score}(v, v')$ for every $\text{S}(Q)$-ancestor v' of v in S.*

Proof. First we introduce $\#\text{N}_P(x)$ as the number of vertices in a set of vertices P that are Q-neighbours of x and $\#\text{F}_P(x)$ as the number of vertices in P that are not Q-neighbours of x. Note that $|P \backslash \{x\}| = \#\text{N}_P(x) + \#\text{F}_P(v)$.

We have

$$S = (\underbrace{r, \dots, \text{pred}(v')}_{p_1}, v', \underbrace{\text{succ}(v'), \dots, \text{pred}(v)}_{p_2}, v, \underbrace{\text{succ}(v) \dots, w}_{p_3}).$$

From the definition of $\text{cost}(\cdot, l)$ follows:

$$\begin{aligned} \text{cost}(v', l) = \#\text{F}_{p_1+v'}(l) + \#\text{N}_{p_2+v+p_3}(l) &\overset{(1)}{=} \#\text{F}_{p_1}(l) + \#\text{N}_{p_2+v} \\ &\overset{(2)}{=} \#\text{F}_{p_1} + \#\text{N}_{p_2} + 1. \end{aligned}$$

The notation $p_i + x$ is an abbreviation for $p_i \cup \{x\}$. Step (1) is valid as v' is a Q-neighbour of l and p_3 does not contain Q-neighbours of l; (2) follows as v is a Q-neighbour of l. Now we analyse $\text{cost}(v, l)$.

$$\begin{aligned} \text{cost}(v, l) = \#\text{F}_{p_1+v'+p_2}(l) + \#\text{N}_{p_3}(l) &\overset{(3)}{=} \#\text{F}_{p_1+v'+p_2}(l) \\ &\overset{(4)}{=} \#\text{F}_{p_1}(l) + \#\text{F}_{p_2}(l). \end{aligned}$$

Step (3) is valid as p_3 does not contain a Q-neighbour of l and (4) follows from v' being a Q-neighbour of l. But then we have for $\text{cost}(v, l) - \text{cost}(v', l)$:

$$\begin{aligned} \text{cost}(v, l) - \text{cost}(v', l) &= \#\text{F}_{p_1}(l) + \#\text{F}_{p_2}(l) - (\#\text{F}_{p_1}(l) + \#\text{N}_{p_2}(l) + 1) \\ &= \#\text{F}_{p_2}(l) - (\#\text{N}_{p_2}(l) + 1) \\ &\overset{(5)}{=} \#\text{F}_{p_2+v'}(l) - (\#\text{N}_{p_2+v'}(l)) \\ &\overset{(6)}{=} \text{Score}(v, v') \end{aligned}$$

Step (5) is valid as v' is a Q-neighbour of l; (6) is justified as a vertex in S is an Q-neighour of l iff it is a Q-neighbour of v. □

Lemma 2.25 implies the following:

1. $\text{Score}_{\max}(v) < 1$
   For all vertices w above v in the sequence we have $\text{Score}(v, w)$ at most zero. But all vertices that maximise $\text{Score}(v, \cdot)$ cannot be Q-neighbours of v and therefore they are not Q-neighbours of l either. Consequentely, they cannot be parents of l. But then the value of $\text{Score}(v', v)$ of a potential parent v' is negative, so inserting l as a leaf of v' is more costly than inserting it as a leaf of v. We do not have to consider vertices below v in the sequence as they are not Q-neighbours of l.

2. $\text{Score}_{\max}(v) = 1$
   Here the same argument as in the first case applies with the only difference that there is a potential parent v' such that inserting l under v' is as costly as inserting l under v. But as both costs are equal we can insert l as a leaf of v.

3. $\text{Score}_{\max}(v) > 1$
   There is a vertex w with $\text{Score}(v, w) = \text{Score}_{\max}(v) \geq 2$. Its predecessor v' in S must be a Q-neighbour of v; otherwise, $\text{Score}(v, w)$ would not be maximal. Note that such a parent always exists as we only regard connected QTGs. Therefore, the root of S is a Q-neighbour of all vertices.
   In this case we have $\text{Score}(v, v') \geq 1$ and v' has a maximal score among all Q-neighbours of v. Hence, inserting l as a leaf of v' is a best choice for l as a position within S.

Now we can prove some propositions about Algorithm 2.3. The term *best parent* of an $l \in V_Q \backslash S$ means that inserting l as a leaf of this vertex causes a least number of edits incident to l.

Lemma 2.26. *Let v_i be the vertex in* seq *with index i. After iteration i of the loop in line 5 the following holds:*

1. *All vertices of $V_Q \backslash S$ that are in* $\text{leavesSeq}(v_j)$ *for a v_j with $j \geq i$ are placed with a minimal number of edits.*
2. *For a vertex that is not in* $\text{leavesSeq}(v_j)$*, where $j \geq i$, v_j cannot be a better parent than its current parent.*
3. *All vertices in $V_Q \backslash S$ for which v_i is the Q-neighbour with greatest depth in S but not a best parent are transfered into* $\text{leavesSeq}(p)$*, and p is a best parent of them.*
4. *$del + adds$ equals the number of the following edits:*
   - *All edits incident to v_j and vertices in* $\text{leavesSeq}(v_j)$*, where $j \geq i$.*
   - *All deletions incident to vertices in* $\text{leavesSeq}(v_k)$*, where $k > i$. (These are the vertices that have been moved above.)*

Proof. We show this by induction on the loop counter i:

- **The claim holds for the first iteration $i :=$ seq.`size`$() - 1$.**

  1. All possible leaves of v_i have to be neighbours of this vertex in Q. Therefore, iterating over all children (lines 7 - 10) of v_i and adding all vertices in the subtrees of those children that are not in S inserts all possible leaves of v_i into $\text{leavesSeq}(v_i)$ as S is a valid sequence. If $\text{Score}_{\max}(v_i) \leq 1$, v_i is a best parent for all these vertices. If $\text{Score}_{\max}(v_i) > 1$, the predecessor p of $w := \text{Score}_{\max}\text{holder}(v)$ in S is a best parent (see Lemma 2.25). Thus, transferring all vertices in $\text{leavesSeq}(v_i)$ to p (line 15) makes them leaves of a best parent. As $\text{Score}_{\max}(p) \leq 1$ (otherwise w would not have a maximal $\text{Score}(v, w)$), the vertices will remain there.

2. We have shown in (1.) that all possible leaves are added initially to leavesSeq(v_i) and only if v_i is not a best parent for them, they are transferred to a best parent above v_i.

3. If $\text{Score}_{\text{max}}(v) > 1$ all vertices in leavesSeq(v_i) are transferred to the predecessor of $\text{Score}_{\text{max}}\text{holder}(v_i)$ and this p is a best parent for them.

4. After line 10 the variable del contains the number of deletions of all edges of a possible leaf to vertices that are not in S. If these vertices remain leaves of v_i, no more deletions incident to them will occur. If they do not, we have to delete all edges to any of their Q-neighbours in the path from v_i to their best parent p. This number is equal to $\text{level}(v_i) - \text{level}(p)$ as S is a valid sequence. In this case these edits are added to del (line 16). After line 17 leavesSeq(v_i) only contains vertices for which v_i is a best parent. Therefore, no additional deletions incident to them are necessary. What remains is to count the edges that have to be inserted incident to these vertices and to v_i itself in order to get the threshold structure. These are the edges from vertices situated above v in the sequence that are not Q-neighbours of v_i in S. As S is a valid sequence, this number is equal to $i - \text{level}(v_i)$.

- **Induction hypothesis**
  The claim holds for all v_j with $j \geq i$ and $i < \text{seq}$.`size`$() - 1$.

- **Inductive step**
  As we do not change leavesSeq(v_j) for any v_j, where $j > i - 1$ in iteration $i - 1$, the claim holds for all these vertices due to the induction hypothesis.

  1. If leavesSeq(v_{i-1}) already contains leaves before the execution of line 7, we know that v_{i-1} is a best parent for them. In this case $\text{Score}_{\text{max}}(v_{i-1}) \leq 1$ and v_{i-1} will keep them. Independent of this, all vertices of $V_Q \backslash S$ for which v_{i-1} is the Q-neighbour with greatest depth in S are added to leavesSeq(v_{i-1}) and kept as leaves iff $\text{Score}_{\text{max}}(v_{i-1}) \leq 1$. Hence, for all vertices in leavesSeq v_{i-1} vertex v_{i-1} is a best parent.

  2. This is true for all vertices in $V_Q \backslash S$ for which a v_j is the Q-neighbour with greatest depth and $j > i - 1$ because of part (2) of the induction hypothesis. The same holds for all vertices for which v_{i-1} is the Q-neighbour with greatest depth in S as they are inserted to leavesSeq(v_{i-1}) in line 10 and kept as leaves iff $\text{Score}_{\text{max}}(v_{i-1}) \leq 1$. All vertices that will be initially inserted to a vertex above v_{i-1} in S are descendants of this vertex in S(Q). Thus, v_{i-1} cannot be a Q-neighbour of them as S is a valid sequence, and so it cannot be a best parent either.

  3. If $\text{Score}_{\text{max}}(v_{i-1}) > 1$, v_{i-1} is not a best parent for the vertices in leavesSeq(v_{i-1}) and they are transfered to a best parent above v_{i-1} in line 15. This vertex p must have a $\text{Score}_{\text{max}}(p) \leq 1$ so they will remain there.

  4. Before the execution of line 7 we know by the induction hypothesis that all edits incident to v_j and vertices in leavesSeq(v_j), where $j > i - 1$, have already been taken into account. Similarly all deletions for vertices in leavesSeq(v_{i-1}) have already been counted. The deletions for isolating vertices for which v_{i-1} is the Q-neighbour in S with greatest depth are counted in line 9. If $\text{Score}_{\text{max}}(v_{i-1}) > 1$, the additional deletions for changing the leaves in leavesSeq(v_{i-1}) into leaves of a best parent are counted in line 16. Finally, all edges that have to be inserted for v_{i-1} and all leaves for which it is a best parent are counted in line 18.

Hence, the claim is also true for iteration $i-1$. This shows that the lemma holds for all iterations of the main loop.

□

Lemma 2.26 shows that the vertices in leavesSeq(v_i) for $v_i \in S$ are placed in such a way that the resulting threshold graph is a nearest S-induced threshold graph of the input graph Q. Hence, it shows the correctness of Theorem 2.23. Also, the complexity of $\mathcal{O}(|S| + |V_Q \backslash S|)$ is correct as every $l \in V_Q \backslash S$ is moved up at least once (if its initial position is not correct, it is moved directly to a best parent and will remain there).

If we insert this algorithm (or rather its $(|\mathcal{S}|)$-version) into Algorithm 2.2 in line 35 instead of printing the current sequence, we obtain the edit distance of a nearest threshold graph induced by the current sequence. In doing so, we can identify the sequence S that induces a threshold graph that has minimal edit distance among all valid sequences S'. In a last step we execute the $\mathcal{O}(|S| + |V_Q \backslash S|)$-version of Algorithm 2.3 on sequence S in order to obtain a skeleton of this nearest threshold graph of Q.

2.4.2.3 Running Time

Given a skeleton S(Q) of a quasi threshold graph Q, where n is the number of inner vertices of S(Q), we know from Lemma 2.22 that

$$\sum_{i=1}^{n} \binom{n}{i} \cdot i! < n! \cdot \sum_{i=1}^{n} \binom{n}{i} < 2^{n(\log n+1)}$$

is an upper bound for the number of calls of the function `calculate` in Algorithm 2.2. As in each call of `calculate` at most once Algorithm 2.3 is executed, we obtain a complexity of $\mathcal{O}(n \cdot 2^{n\cdot(\log n+1)}) = \mathcal{O}(2^{n\cdot(\log n+1)+\log n})$. If we take into account that the algorithms requires a skeleton of the input graph that has to be calculated and that we must executed the $\mathcal{O}(|V_Q|)$-version of Algorith 2.3 one time in order to obtain a nearest threshold graph, the overall complexity is in $\mathcal{O}(2^{n\cdot(\log n+1)+\log n} + |V_Q| + |E_Q|)$. Note that this is not a sharp upper bound for the complexity of the algorithm.

2.4.3 Heuristic

After introducing an algorithm that solves the threshold editing problem for quasi-threshold graphs exactly, we will present a heuristic for this problem. It is based on the Quasi-Threshold Mover presented in [BHSW15], but instead of editing to quasi-threshold graphs we edit to threshold graphs. Let Q be a quasi-threshold graph and S(Q) a skeleton of Q. The algorithm works in rounds. We start with an initial threshold graph skeleton of Q. This initial threshold skeleton can be obtained by arranging the vertices of Q to form an arbitrary caterpillar or by using a more advanced technique. In each round the algorithm iterates over the vertices v of Q in a random order. For each vertex v a position that causes the least editing costs within the current skeleton is found and v is moved to this position. One round of the algorithm presented in the paper needs $\mathcal{O}(|V_Q| + |E_Q| \log \Delta)$ time, where Δ is the maximum degree. We will present a simple version that has a quadratic running time per round as this running time is sufficient in order to compare the heuristic to the exact algorithm. The description in Algorithm 2.4 shows one round of this algorithm.

The main calculation is performed in the functions `CostsInsertAsLeafOf`() (line 11) and `CostsAdoptChildrenOf`() (line 12). We will briefly explain how these functions work. Suppose that the central path of curSkel has the following structure:

$$(\underbrace{r, \ldots, \mathrm{pred}(p)}_{p_1}, p, \underbrace{\mathrm{succ}(p), \ldots, w}_{p_2}).$$

Algorithm 2.4: ONE-ROUND-THRESHOLD-MOVING

```
Input: Quasi-threshold-graph $Q$.
Initial threshold skeleton Inital of $Q$, i.e. a skeleton of a threshold graph that has the same vertex set as $Q$.
Output: Skeleton of a threshold graph of $Q$.
permutation ← randomPermutation($V_Q$)
curSkel ← Inital
centralPath ← CalcCentralPath(curSkel)
for $v_m \in$ permutation do
    curSkel.remove($v_m$)
    $\text{pos}_{\text{opt}} \leftarrow -1$
    $\text{score}_{\text{opt}} \leftarrow \infty$
    leaf ← true
    for $i \leftarrow 1$ to centralPath.size() do
        $p \leftarrow$ centralPath($i$)
        $\text{edits}_{\text{leaf}} \leftarrow$ CostsInsertAsLeafOf($p$)
        $\text{edits}_{\text{adopt}} \leftarrow$ CostsAdoptChildrenOf($p$) // If a child of p is adopted, centralPath($i$+1) must be adopted, too

        if $\min(\text{edits}_{\text{leaf}}, \text{edits}_{\text{adopt}}) < \text{score}_{\text{opt}}$ then
            $\text{score}_{\text{opt}} \leftarrow \min(\text{edits}_{\text{leaf}}, \text{edits}_{\text{adopt}})$
            $\text{pos}_{\text{opt}} \leftarrow p$
            if $\text{edits}_{\text{leaf}} \leq \text{edits}_{\text{adopt}}$ then
                leaf ← true
            else
                leaf ←false
    curSkel.insert($v_m$, $\text{pos}_{\text{opt}}$, leaf)
```

1. `CostsInsertAsLeafOf`(p)

   This function calculates the costs for inserting v_m as a leaf of p. In order to do this we have to

   a) insert edges from all vertices in $p_1 + p$ that are not Q-neighbours of v_m.

   b) delete edges from all Q-neighbours of v_m that are not on the path $p_1 + p$.

2. `CostsAdoptChildrenOf`(p)

   The function calculates the cost if we want to adopt a child of p. Note that we have to adopt at least succ(p) (if $p \neq w$). Adopting another child of p but not succ(p) would destroy the caterpillar structure of the skeleton. Therefore we must adopt succ(p) and all other leaves of p that are Q-neighbour of v_m. The edge operations are as follows

   a) inserting edges from all vertices in $p_1 + p$ that are not Q-neighbours of v_m.

   b) deleting the edges from all Q-neighbours of v_m that are not on the path $p_1 + p$ or that are not in the adopted subtrees.

   c) inserting edges to all vertices that are not Q-neighbours of v_m and that are situated in one of the adopted subtrees.

For the computation of the requested values both functions need information on the number of Q-neighbours and vertices in the path above p, the total number of Q-neighbours and vertices above p that are not in the central path, as well as the number of Q-neighbours/vertices below p. All these values can be kept up-to-date while iterating over the central path of curSkel. Additionally, we need a test in $\mathcal{O}(1)$ as to whether a vertex w is adjacent to v_m in Q. This can be realized using a $n \times 1$ boolean array indicating the neighbourhood of v_m. This array can be updated in every iteration of the outer loop in $\mathcal{O}(\text{degree}(v_m))$, where $\text{degree}(v_m)$ is the degree of v_m in Q.

In our implementation we do not have a fixed upper bound for the number of rounds. Instead, the algorithm terminates when it cannot improve the current threshold graph in three consecutive rounds.

3. Experimental Evaluation

In this chapter we will experimentally evaluate the proposed algorithms. We will discuss differences between the exact algorithm for threshold editing (algorithm E), the exact algorithm for threshold deletion (algorithm D) and the heuristic for threshold editing (algorithm H). In particular we will consider the following two aspects:

- differences in the number of edits that are calculated by the algorithms E, D and H for an input G.
- differences in the running time of the three algorithms for an input G.

Note that the first aspect is rather a comparison between the problems threshold editing and threshold deletion than a comparison between algorithms E and D as both calculate the minimal number of edits that are necessary in order to make a quasi-threshold graph G a threshold graph in the context of threshold editing and threshold deletion, respectively. For the heuristic H we use a tree in which all vertices are children of the root as the initial skeleton.

For the evaluation we use generated quasi-threshold graphs. In doing so we can easily obtain input graphs of different size. An important question is how we generate these quasi-threshold graphs. We have decided to use the following procedure in order to obtain a tree of size n. This tree induces a connected quasi-threshold graph of the same size.

Algorithm 3.1: Random Trees

```
    Input: $n$, size of the arbitrarily generated tree.
    Output: Array parent containing each vertex's parent.
1   parent(1) $\leftarrow -1$ // vertex 1 is the root of the skeleton
2   forall $i = 2$ to $n$ do
3       parent($i$) $\leftarrow$ random number between 1 and $i - 1$
```

The algorithm can generate every connected quasi-threshold graph of size n. However, we should keep in mind that the generated graphs are not uniformly distributed. We have implemented the algorithms in C++11. All experiments are executed on an Intel Core i5-4690 CPU with 8GB RAM.

data set		deviations in the number of edits					
		E vs. D		E vs. H		E vs. H^*	
id	n	#dev.	max. diff.	#dev.	max. diff.	#dev.	max. diff.
D_1	100	30	24	53	32	18	21
D_2	200	50	55	68	92	24	43
D_3	300	51	67	80	187	26	45

Table 3.1: Number of graphs in each data set in which the results of D, H and H^* differ from the optimal solution calculated by E and the maximal difference among these deviations.

3.1 Number of Edits

In this section we compare the number of edits that are calculated by the algorithms E, D and H for input graphs G. We evaluate the algorithms on three data sets: D_1, D_2 and D_3. Each data set consists of 100 quasi-threshold graphs that have 100, 200 and 300 vertices respectively. These input graphs have been generated by Algorithm 3.1.

Figures 3.1, 3.2 and 3.3 show the results of D_1, D_2 and D_3. Each number between 1 and 100 on the x-axis represents a quasi-threshold graph G of the considered data set. In this experiment we calculate the ratios

$$r_E = \frac{\#\text{edits by}\, E}{\#\text{edits by}\, E} = 1,\; r_D = \frac{\#\text{edits by}\, D}{\#\text{edits by}\, E},\; r_H = \frac{\#\text{edits by}\, H}{\#\text{edits by}\, E}$$

for each input graph G. The ratios r_D (blue) and r_H (black) are the factors by which the results of algorithms D and H differ from the optimal solution calculated by E. The green bars represent the running time of algorithm E on input graph G. The graphs are ordered increasingly by the number of edits calculated by E.

In all three Figures there is no identifiable link between the number of edits calculated by E and the running time of this algorithm. Furthermore, the results imply that r_D and r_E are independent of E's running time.

In the data sets D_1 and D_2, r_D is always smaller than 1.15 and in D_3 it is even smaller than 1.1. At least for these randomly generated graphs there is only a small mean difference between the edit and deletion distances. In our sample, r_H tends to be greater than r_D. In order to improve the heuristic one could use the result of D as H's initial skeleton. Since D is a linear-time algorithm, this does not interfere with H's complexity. We denote this version of the heuristic by H^*. In the Tables 3.1 and 3.2, H^* is compared to the other algorithms. To contrast the ratios r_E, r_D and r_H, Figure 3.4 shows the absolute numbers of edits of the three algorithms on data set D_3. The graphs showing the absolute number of edits in D_1 and D_2 are of similar shape.

Table 3.1 shows the number of graphs in which the result of D, H and H^* differs from the optimal solution calculated by E for all three data sets. Additionally, the highest difference in each data set is given. We can see that algorithm H^* can reduce the number of graphs in which the calculated solution is not equal to the result of algorithm E significantly compared to the algorithms D and H. Also, the maximum difference is reduced.

Table 3.2 shows the average number of edits for all algorithms on the data sets and its standard deviation. The latter is large, therefore the average is only of limited significance. But in combination with the results displayed in Table 3.1 we can conclude that H^* computes significantly better results than H. In our sample, it can reduce the average difference between the results calculated by E and those calculated by D by about 50%.

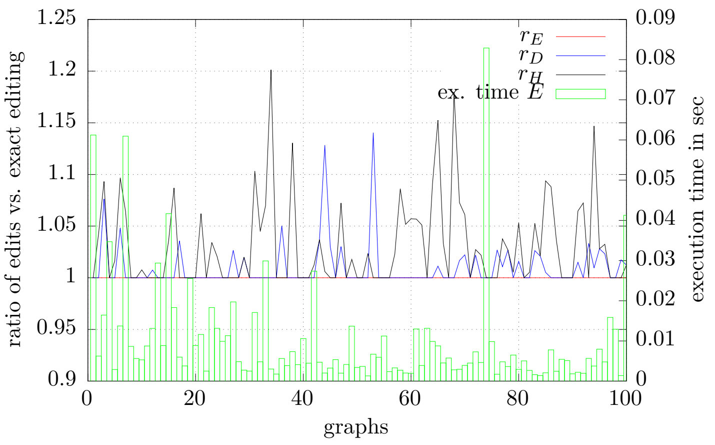


Figure 3.1: Relative number of edits in data set D_1: 100 graphs of size 100.

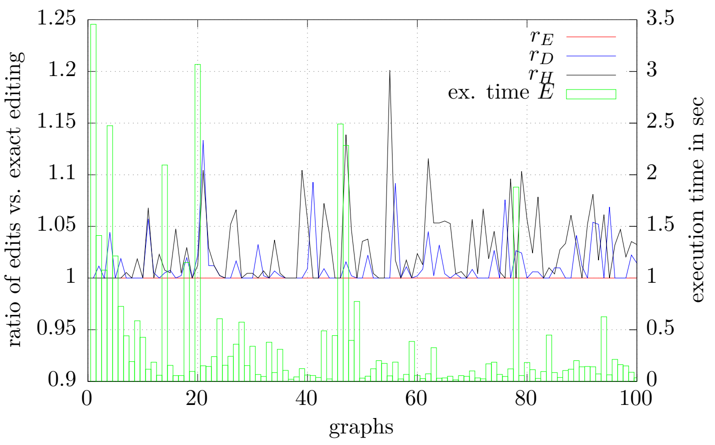


Figure 3.2: Relative number of edits in data set D_2: 100 graphs of size 200.

3.2 Running Time

Figure 3.5 gives an overview of the running time of all three algorithms on input graphs of different sizes (x-axis). For each considered input size n, we let the algorithm run on 50 random quasi-threshold graphs of size n. The plot shows the average running time of the algorithms on these sets. Note that the y-axis is logarithmically scaled. We can see that the running time of algorithm E grows exponentially. However, the plot is not monotone; this – like Figures 3.1, 3.2 and 3.3 – shows again that the running time of E has a large variance and depends strongly on the structure of the input graph.

data set		number of edits							
		E		D		H		H^*	
id	n	mean	std.	mean	std.	mean	std.	mean	std.
D_1	100	167.71	27.16	169.36	27.78	172.88	29.20	168.62	27.83
D_2	200	451.24	48.02	456.83	50.34	464.29	55.21	453.86	48.73
D_3	300	790.67	74.06	798.53	77.78	816.52	89.29	794.23	76.03

Table 3.2: Mean edit distance of the algorithms E,D,H and H^* on the three data sets.

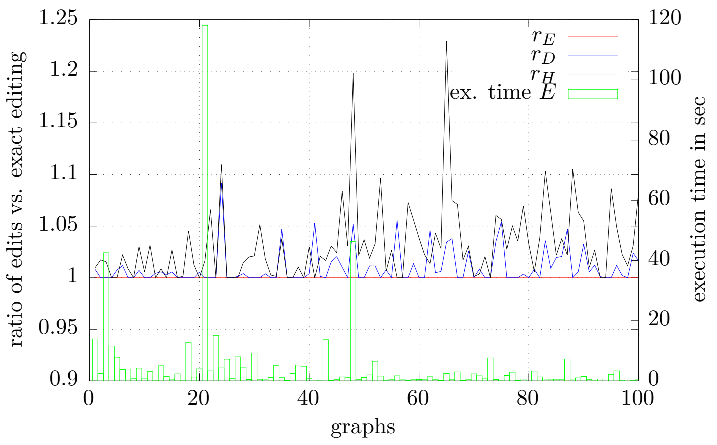


Figure 3.3: Relative number of edits in data set D_3: 100 graphs of size 300.

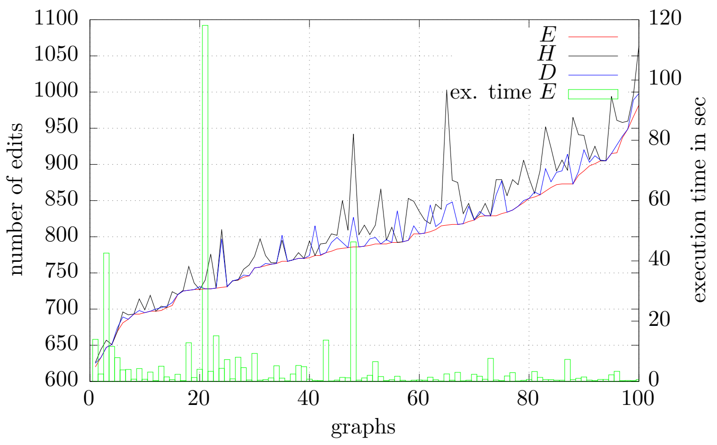


Figure 3.4: Absolute number of edits in data set D_3: 100 graphs of size 300.

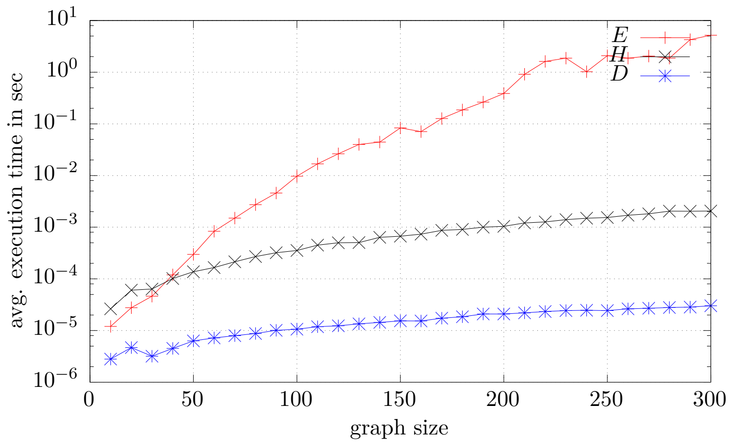


Figure 3.5: Running time on graphs of different size.

3.3 Other Graphs

We want to complete the picture by giving some examples in which our above findings do not hold. In the case of the differences in the number of edits between threshold editing and threshold deletion we have seen that the average difference (on the selected data sets) was small compared to the total number of edits necessary for a given input graph. However, for every $n \in \mathbb{N}$ one can generate a quasi-threhold graph Q such that its edit distance is n while its deletion distance equals $2n$ resulting in $r_D = 2$ (see Section 2.4).

Concerning running time, the results show that algorithm E runs in feasible time on the graphs in our sample with a size of up to 300 vertices. However, there are special graphs on which algorithm E does not perform as well as on the graphs in the data sets analyzed so far. One of these 'critical' graph classes are quasi-threshold graphs which are induced by a skeleton that consists of one universal vertex u and two paths of size n (or of size n and $n-1$ if the number of vertices of the graph is even) that are subtrees of u. Figure 3.6 shows the running time of the algorithms on these graphs. The running time of algorithm E is significantly greater than its running time on random graphs generated by Algorithm 3.1. However, the edit and deletion distance are equal on these graphs (namely $n(n-1)/2$). This is large value compared to the edit distance of the randomly generated graphs in our samples. This suggests, that also the ftp-based algorithm from [DDLS15] would have difficulties in handling this kind of graphs.

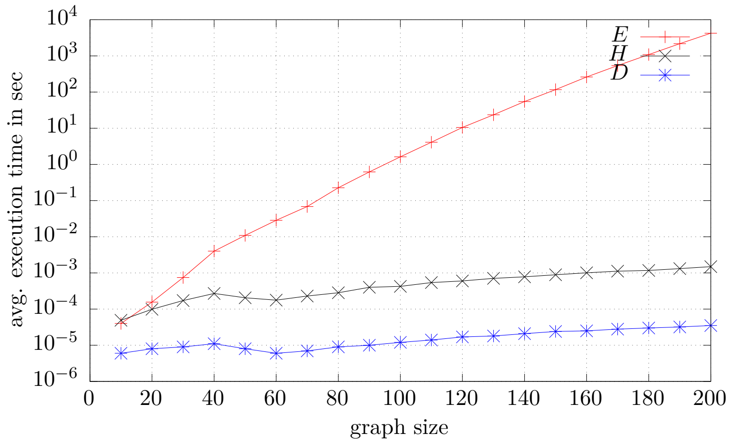


Figure 3.6: Running time of a QTG induced by a skeleton with a universal u and two paths as subtrees of u.

It seems that the mean difference between the edit and deletion distances of randomly generated graphs is not very large although the size of this difference is not bounded. Furthermore, algorithm E seems to perform better on randomly generated graphs than on the graphs introduced in this section.

4. Conclusion

In the first part of this thesis, we have given a linear time algorithm solving the problem of threshold deletion on quasi-threshold graphs, which is $\mathcal{NP}$-complete on general graphs. The algorithm takes a skeleton of its input graph and calculates a skeleton of a nearest threshold graph of the input.

In the second part we have presented an algorithm for the problem of threshold editing on quasi-threshold graphs, which is also $\mathcal{NP}$-complete on general graphs (and even on split graphs). The algorithm is based on the following observation: given a skeleton $\mathrm{S}(Q)$ of a quasi-threshold graph Q, there is a skeleton $\mathrm{S}(T)$ of a nearest threshold graph T of Q such that all leaves in $\mathrm{S}(Q)$ are also leaves in $\mathrm{S}(T)$. Furthermore, among all skeletons $\mathrm{S}(T)$ with this property we can find a skeleton $\mathrm{S}(T')$ such that a certain 'structure' of $\mathrm{S}(Q)$ is maintained in $\mathrm{S}(T')$. We could show that given a sequence of vertices, we can calculate in linear time a threshold graph with minimal edit distance from the input among all threshold graphs that have this sequence as their central path and the same vertex set as the input. Therefore, the main problem is to find the right sequence of inner vertices. From the observation described above it follows that we can ignore all leaves in a skeleton of Q. This and the maintained 'structure' of the input skeleton reduces the number of sequences that we have to test significantly. But it is still exponential in the number of inner vertices of the input skeleton.
Further, we have adapted an existing heuristic for the problem of quasi-threshold editing to work for the problem of threshold editing.

In a last step we have evaluated all three algorithms. On our randomly generated input graphs with a size of up to 300 the exact algorithm for threshold editing works in feasible time. The generated quasi-threshold graphs of this size (in our sample) have a mean edit distance of about 800. There are fpt-based approaches for the threshold editing problem with a complexity in $2^{\mathcal{O}(\sqrt{k}\log k)} + \mathrm{poly}(n)$, where k is the edit distance and n the number of vertices of the input graph. Our results suggest that our algorithm can handle input graphs for which the running time of the fpt-based algorithm would not be feasible. Another result of our evaluation is that the mean difference between the edit and deletion distance of quasi-threshold graphs seems to be relatively small. This suggests that calculating (in linear time) a threshold graph with minimal deletion distance from an input graph might be a good alternative to the potentially time-consuming calculation of a threshold graph with minimal edit distance.

Future work. The $\mathcal{NP}$-completeness of threshold editing on quasi-threshold graphs remains an open problem.

One might try to improve the running time of our exact algorithm for threshold editing by identifying further rules that help to reduce the number of possible sequences that have to be tested. We noticed that quasi-threshold graphs with a small number of leaves in their skeletons are likely to have identical edit and deletion distances. We think that this could lead to a rule excluding many sequences that cannot be part of an optimal solution.

Further, one might implement a linear time version of the heuristic using the ideas in [BHSW15].

Bibliography

[BB13] Sebastian Böcker and Jan Baumbach. *Cluster Editing*, pages 33–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser, and Dorothea Wagner. *Fast Quasi-Threshold Editing*, pages 251–262. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[BLS99] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes: a survey*, volume 3. Siam, 1999.

[Cai96] Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58(4):171 – 176, 1996.

[CH73] Vašek Chvátal and Peter Hammer. Set packing and threshold graphs. *Univ. Waterloo Res. Report*, pages 73–21, 1973.

[DDLS15] Pål Grønås Drange, Markus Sortland Dregi, Daniel Lokshtanov, and Blair D Sullivan. On the threshold of intractability. In *Algorithms-ESA 2015*, pages 411–423. Springer, 2015.

[JHJJC96] Yan Jing-Ho, Chen Jer-Jeong, and Gerard J. Chang. Quasi-threshold graphs. *Discrete Applied Mathematics*, 69(3):247 – 255, 1996.

[Mar94] François Margot. Some complexity results about threshold graphs. *Discrete Applied Mathematics*, 49(1):299 – 308, 1994.

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