# Fast Quasi-Threshold Editing*

Ulrik Brandes[1], Michael Hamann[2], Ben Strasser[2], and Dorothea Wagner[2]

[1] Computer and Information Science, University of Konstanz, Germany
ulrik.brandes@uni-konstanz.de
[2] Faculty of Informatics, Karlsruhe Institute of Technology, Germany
{michael.hamann,strasser,dorothea.wagner}@kit.edu

**Abstract.** We introduce Quasi-Threshold Mover (QTM), an algorithm to solve the quasi-threshold (also called trivially perfect) graph editing problem with a minimum number of edge insertions and deletions. Given a graph it computes a quasi-threshold graph which is close in terms of edit count, but not necessarily closest as this edit problem is NP-hard. We present an extensive experimental study, in which we show that QTM performs well in practice and is the first heuristic that is able to scale to large real-world graphs in practice. As a side result we further present a simple linear-time algorithm for the quasi-threshold recognition problem.

## 1 Introduction

Quasi-Threshold graphs, also known as *trivially perfect* graphs, are defined as the $P_4$- and $C_4$-free graphs, i.e., the graphs that do not contain a path or cycle of length 4 as node-induced subgraph [20]. They can also be characterized as the transitive closure of rooted forests [19], as illustrated in Fig. 1. These forests can be seen as skeletons of quasi-threshold graphs. Further a constructive characterization exists: Quasi-threshold graphs are the graphs that are closed under disjoint union and the addition of isolated nodes and nodes connected to every existing node [20].

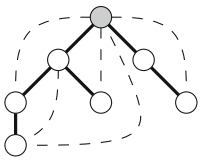


**Fig. 1.** Quasi-thres. graph with thick skeleton, grey root and dashed transitive closure.

Linear time quasi-threshold recognition algorithms were proposed in [20] and in [9]. Both construct a skeleton if the graph is a quasi-threshold graph. Further, [9] also finds a $C_4$ or $P_4$ if the graph is no quasi-threshold graph.

Nastos and Gao [14] observed that components of quasi-threshold graphs have many features in common with the informally defined notion of communities in social networks. They propose to find a quasi-threshold graph that is close to a given graph in terms of edge edit distance in order to detect the communities of that graph. Motivated by their insights we study the quasi-threshold graph editing problem in this paper. Given a graph $G = (V, E)$ we want to find a quasi-threshold graph $G' = (V, E')$ which is closest to $G$, i.e., we want to minimize the number $k$ of edges in the symmetric difference of $E$ and $E'$. Figure 2 illustrates

* This work was supported by the DFG under grants BR 2158/6-1, WA 654/22-1, and BR 2158/11-1.


N. Bansal and I. Finocchi (Eds.): ESA 2015, LNCS 9294, pp. 251–262, 2015.
DOI: 10.1007/978-3-662-48350-3_22

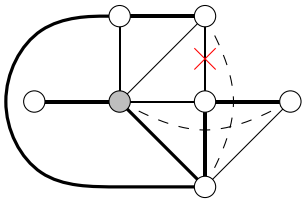


**Fig. 2.** Edit example with solid input edges, dashed inserted edges, a crossed deleted edge, a thick skeleton with grey root.

an edit example. Unfortunately, the quasi-threshold graph editing problem is NP-hard [14]. However, the problem is fixed parameter tractable (FPT) in $k$ as it is defined using forbidden subgraphs [7]. A basic bounded search tree algorithm which tries every of the 6 possible edits of a forbidden subgraph has a running time in $O(6^k \cdot (|V| + |E|))$. In [11] a polynomial kernel of size $O(k^7)$ was introduced. Unfortunately, our experiments show that real-world social networks have a prohibitively large amount of edits. We prove lower bounds on real-world graphs for $k$ on the scale of $10^4$ and $10^5$. A purely FPT-based algorithm with the number of edits as parameter can thus not scale in practice. The only heuristic we are aware of was introduced by Nastos and Gao [14]. It greedily picks edits that result in the largest decrease in the number of induced $C_4$ and $P_4$ in the graph. Unfortunately, it examines all $\Theta(|V|^2)$ possible edits in each step and thus needs $\Omega(k \cdot |V|^2)$ running time. Even though this running time is polynomial it is still prohibitive for large graphs. In this paper we fill this gap by introducing Quasi-Threshold Mover (QTM), the first scalable quasi-threshold editing heuristic. The final aim of our research is to determine whether quasi-threshold editing is a useful community detection algorithm. Designing an algorithm able to solve the quasi-threshold editing problem on large real-world graphs is a first step in this direction.

### 1.1 Our Contribution

Our main contribution is Quasi-Threshold Mover (QTM), a scalable quasi-threshold editing algorithm. We provide an extensive experimental evaluation on synthetic as well as a variety of real-world graphs. We further propose a simplified certifying quasi-threshold recognition algorithm. QTM works in two phases: An initial skeleton forest is constructed by a variant of our recognition algorithm, and then refined by moving one node at a time to reduce the number of edits required. The running time of the first phase is dominated by the time needed to count the number of triangles per edge. The best current triangle counting algorithms run in $O(|E|\alpha(G))$ [8,15] time, where $\alpha(G)$ is the arboricity. These algorithms are efficient and scalable in practice on the considered graphs. One round of the second phase needs $O(|V| + |E| \log \Delta)$ time, where $\Delta$ is the maximum degree. We show that four rounds are enough to achieve good results.

### 1.2 Outline

Our paper is organized as follows: We begin by describing how we compute lower bounds on the number of edits. We then introduce the simplified recognition algorithm and the computation of the initial skeleton. The main algorithm is

described in Sect. 4. The remainder of the paper is dedicated to the experimental evaluation. An extended version of this paper is available on arXiv [6].

### 1.3 Preliminaries

We consider simple, undirected graphs $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. For $v \in V$ let $N(v)$ be the adjacent nodes of $v$. Let $d(v) := |N(v)|$ for $v \in V$ be the degree of $v$ and $\Delta$ the maximum degree in $G$. Whenever we consider a skeleton forest, we denote by $p(u)$ the parent of a node $u$.

## 2 Lower Bounds

A lot of previous research has focused on FPT-based algorithms. To show that no purely FPT-based algorithm parameterized in the number of edits can solve the problem we compute lower bounds on the number of edits required for real-world graphs. The lower bounds used by us are far from tight. However, the bounds are large enough to show that any algorithm with a running time superpolynomial in $k$ can not scale.

To edit a graph we must destroy all forbidden subgraphs $H$. For quasi-threshold editing $H$ is either a $P_4$ or a $C_4$. This leads to the following basic algorithm: Find forbidden subgraph $H$, increase the lower bound, remove all nodes of $H$, repeat. This is correct as at least one edit incident to $H$ is necessary. If multiple edits are needed then accounting only for one is a lower bound. We can optimize this algorithm by observing that not all nodes of $H$ have to be removed. If $H$ is a $P_4$ with the structure $A - B - C - D$ it is enough to remove the two central nodes $B$ and $C$. If $H$ is a $C_4$ with nodes $A$, $B$, $C$, and $D$ then it is enough to remove two adjacent nodes. Denote by $B$ and $C$ the removed nodes. This optimization is correct if at least one edit incident to $B$ or $C$ is needed. Regardless of whether $H$ is a $P_4$ or a $C_4$ the only edit not incident to $B$ or $C$ is inserting or deleting $\{A, D\}$. However, this edit only transforms a $P_4$ into a $C_4$ or vice versa. A subsequent edit incident to $B$ or $C$ is thus necessary.

$H$ can be found using the recognition algorithm. However, the resulting running time of $O(k(n + m))$ does not scale to the large graphs. In the extended version [6] we describe a running time optimization to accelerate computations.

## 3 Linear Recognition and Initial Editing

The first linear time recognition algorithm for quasi-threshold graphs was proposed in [20]. In [9], a linear time certifying recognition algorithm based on lexicographic breadth first search was presented. However, as the authors note, sorted node partitions and linked lists are needed, which result in large constants behind the big-O. We simplify their algorithm to only require arrays but still provide negative and positive certificates. Further we only need to sort the nodes once to iterate over them by decreasing degree. Our algorithm constructs

the forest skeleton of a graph $G$. If it succeeds $G$ is a quasi-threshold graph and outputs for each node $v$ a parent node $p(v)$. If it fails it outputs a forbidden subgraph $H$.

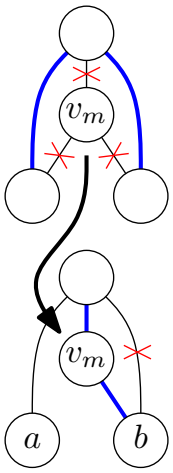
To simplify our algorithm we start by adding a super node $r$ to $G$ that is connected to every node and obtain $G'$. $G$ is a quasi-threshold graph if and only if $G'$ is one. As $G'$ is connected its skeleton is a tree. A core observation is that higher nodes in the tree must have higher degrees, i.e., $d(v) \leq d(p(v))$. We therefore know that $r$ must be the root of the tree. Initially we set $p(u) = r$ for every node $u$. We process all remaining nodes ordered decreasingly by degree. Once a node is processed its position in the tree is fixed. Denote by $u$ the node that should be processed next. We iterate over all non-processed neighbors $v$ of $u$ and check whether $p(u) = p(v)$ holds and afterwards set $p(v)$ to $u$. If $p(u) = p(v)$ never fails then $G$ is a quasi-threshold graph as for every node $x$ (except $r$) we have that by construction that the neighborhood of $x$ is a subset of the one of $p(x)$. If $p(u) \neq p(v)$ holds at some point then a forbidden subgraph $H$ exists. Either $p(u)$ or $p(v)$ was processed first. Assume without lose of generality that it was $p(v)$. We know that no edge $(v, p(u))$ can exist because otherwise $p(u)$ would have assigned itself as parent of $v$ when it was processed. Further we know that $p(u)$'s degree can not be smaller than $u$'s degree as $p(u)$ was processed before $u$. As $v$ is a neighbor of $u$ we know that another node $x$ must exist that is a neighbor of $p(u)$ but not of $u$, i.e., $(u, x)$ does not exist. The subgraph $H$ induced by the 4-chain $v - u - p(u) - x$ is thus a $P_4$ or $C_4$ depending on whether the edge $(v, x)$ exists. We have that $u$, $v$ and $p(u)$ are not $r$ as $p(v)$ was processed before them and $r$ was processed first. As $x$ has been chosen such that $(u, x)$ does not exist but $(u, r)$ exist $x \neq r$. $H$ therefore does not use $r$ and is contained in $G$.

*From Recognition to Editing.* We modify the recognition algorithm to construct a skeleton for arbitrary graphs. This skeleton induces a quasi-threshold graph $Q$. We want to minimize $Q$'s distance to $G$. Note that all edits are performed implicitly, we do not actually modify the input graph for efficiency reasons. The only difference between our recognition and our editing algorithm is what happens when we process a node $u$ that has a non-processed neighbor $v$ with $p(u) \neq p(v)$. The recognition algorithm constructs a forbidden subgraph $H$, while the editing algorithm tries to resolve the problem. We have three options for resolving the problem: we ignore the edge $\{u, v\}$, we set $p(v)$ to $p(u)$, or we set $p(u)$ to $p(v)$. The last option differs from the first two as it affects all neighbors of $u$. The first two options are the decision if we want to make $v$ a child of $u$ even though $p(u) \neq p(v)$ or if we want to ignore this potential child. We start by determining a preliminary set of children by deciding for each non-processed neighbor of $u$ whether we want to keep or discard it. These preliminary children elect a new parent by majority. We set $p(u)$ to this new parent. Changing $u$'s parent can change which neighbors are kept. We therefore reevaluate all the decisions and obtain a final set of children for which we set $u$ as parent. Then the algorithm simply continues with the next node.

What remains to describe is when our algorithm keeps a potential child. It does this using two edge measures: The number of triangles $t(e)$ in which an edge

```
1  foreach v_m-neighbor u do
2      push u;
3  while queue not empty do
4      u ← pop;
5      determine child_close(u) by DFS;
6      x ← max over score_max of reported u-children;
7      y ← ∑ over child_close of close u-children;
8      if u is v_m-neighbor then
9          score_max(u) ← max{x, y} + 1;
10     else
11         score_max(u) ← max{x, y} − 1;
12     if child_close(u) > 0 or score_max(u) > 0 then
13         report u to p(u);
14         push p(u);
15 Best v_m-parent corresponds to score_max(r);
```

(a) Pseudo-Code for moving $v_m$




(b) Moving $v_m$ example

**Fig. 3.** In Fig. 3b the drawn edges are in the skeleton. By moving $v_m$, crossed edges are removed and thick blue edges are inserted. $a$ is not adopted while $b$ is.

$e$ participates and a pseudo-$C_4$-$P_4$-counter $p_c(e)$, which is the sum of the number of $C_4$ in which $e$ participates and the number of $P_4$ in which $e$ participates as central edge. Computing $p_c(x, y)$ is easy given the number of triangles and the degrees of $x$ and $y$ as $p_c(\{x, y\}) = (d(x) - 1 - t(\{x, y\})) \cdot (d(y) - 1 - t(\{x, y\}))$ holds. Having a high $p_c(e)$ makes it likely that $e$ should be deleted. We keep a potential child only if two conditions hold. The first is based on triangles. We know by construction that both $u$ and $v$ have many edges in $G$ towards their current ancestors. Keeping $v$ is thus only useful if $u$ and $v$ share a large number of ancestors as otherwise the number of induced edits is too high. Each common ancestor of $u$ and $v$ results in a triangle involving the edge $\{u, v\}$ in $Q$. Many of these triangles should also be contained in $G$. We therefore count the triangles of $\{u, v\}$ in $G$ and check whether there are at least as many triangles as $v$ has ancestors. The other condition uses $p_c(e)$. The decision whether we keep $v$ is in essence the question of whether $\{u, v\}$ or $\{v, p(v)\}$ should be in $Q$. We only keep $v$ if $p_c(\{u, v\})$ is not higher than $p_c(\{v, p(v)\})$. The details of the algorithm can be found in the extended version [6]. The time complexity of this editing heuristic is dominated by the triangle counting algorithm as the rest is linear.

## 4 The Quasi-Threshold Mover Algorithm

The Quasi-Threshold Mover (QTM) algorithm iteratively increases the quality of a skeleton $T$ using an algorithm based on local moving. Local moving is a technique that is successfully employed in many heuristic community detection algorithms [2,12,16]. As in most algorithm based on this principle, our algorithm

works in rounds. In each round it iterates over all nodes $v_m$ in random order and tries to move $v_m$. In the context of community detection, a node is moved to a neighboring community such that a certain objective function is increased. In our setting we want to minimize the number of edits needed to transform the input graph $G$ into the quasi-threshold graph $Q$ implicitly defined by $T$. We need to define the set of allowed moves for $v_m$ in our setting. Moving $v_m$ consists of moving $v_m$ to a different position within $T$ and is illustrated in Fig. 3b. We need to chose a new parent $u$ for $v_m$. The new parent of $v_m$'s old children is $v_m$'s old parent. Besides choosing the new parent $u$ we select a set of children of $u$ that are *adopted* by $v_m$, i.e., their new parent becomes $v_m$. Among all allowed moves for $v_m$ we chose the move that reduces the number of edits as much as possible. Doing this in sub-quadratic running time is difficult as $v_m$ might be moved anywhere in $G$. By only considering the neighbors of $v_m$ in $G$ and a few more nodes per neighbor in a bottom-up scan in the skeleton, our algorithm has a running time in $O(n + m \log \Delta)$ per round. While our algorithm is not guaranteed to be optimal as a whole we can prove that for each node $v_m$ we choose a move that reduces the number of edits as much as possible. Our experiments show that given the result of the initialization heuristic our moving algorithm performs well in practice. They further show that in practice four rounds are good enough which results in a near-linear total running time.

*Basic Idea.* Our algorithm starts by isolating $v_m$, i.e., removing all incident edges in $Q$. It then finds a position at which $v_m$ should be inserted in $T$. If $v_m$'s original position was optimal then it will find this position again. For simplicity we will assume again that we add a virtual root $r$ that is connected to all nodes. Isolating $v_m$ thus means that we move $v_m$ below the root $r$ and do not adopt any children. Choosing $u$ as parent of $v_m$ requires $Q$ to contain edges from all ancestors of $u$ to $v_m$. Further if $v_m$ adopts a child $w$ of $u$ then $Q$ must have an edge from every descendant of $w$ to $v_m$. How good a move is depends on how many of these edges already exist in $G$ and how many edges incident to $v_m$ in $G$ are not covered. To simplify notation we will refer to the nodes incident to $v_m$ in $G$ as *$v_m$-neighbors*. We start by identifying which children a node should adopt. For this we define the *child closeness* $\text{child}_{\text{close}}(u)$ of $u$ as the number of $v_m$-neighbors in the subtree of $u$ minus the non-$v_m$-neighbors. A node $u$ is a *close child* if $\text{child}_{\text{close}}(u) > 0$. If $v_m$ chooses a node $u$ as new parent then it should adopt all close children. A node can only be a close child if it is a neighbor of $v_m$ or when it has a close child. Our algorithm starts by computing all close children and their closeness using many short DFS searches in a bottom up fashion. Knowing which nodes are good children we can identify which nodes are good parents for $v_m$. A potential parent must have a close child or must be a neighbor of $v_m$. Using the set of close children we can easily derive a set of parent candidates and an optimal selection of adopted children for every potential parent. We need to determine the candidate with the fewest edits. We do this in a bottom-up fashion.To implement the described moving algorithm we need to put $O(d_G(v_m))$ elements into a priority queue. The running time is thus amortized $O(d_G(v_m) \log d_G(v_m))$ per move or $O(n + m \log \Delta)$ per round.

We analyze the running time complexity using tokens. Initially only the $v_m$-neighbors have tokens. The tokens are consumed by the short DFS searches and the processing of parent nodes. The details of the analysis are complex and are described in the extended version [6].

*Close Children.* To find all close children we attach to each node $u$ a DFS instance that explores the subtree of $u$. Note that every DFS instance has a constant state size and thus the memory consumption is still linear. $u$ is close if this DFS finds more $v_m$-neighbors than non-$v_m$-neighbors. Unfortunately we can not fully run all these searches as this requires too much running time. Therefore a DFS is aborted if it finds more non-$v_m$-neighbors than $v_m$-neighbors. We exploit that close children are $v_m$-neighbors or have themselves close children. Initially we fill a queue of potential close children with the neighbors of $v_m$ and when a new close child is found we add its parent to the queue. Let $u$ denote the current node removed from the queue. We run $u$'s DFS and if it explores the whole subtree then $u$ is a close child. We need to take special care that every node is visited only by one DFS. A DFS therefore looks at the states of the DFS of the nodes it visits. If one of these other DFS has run then it uses their state information to skip the already explored part of the subtree. To avoid that a DFS is run after its state was inspected we organize the queue as priority queue ordered by tree depth. If the DFS of $u$ starts by first inspecting the wrong children then it can get stuck because it would see the $v_m$-neighbors too late. The DFS must first visit the close children of $u$. To assure that $u$ knows which children are close every close child must report itself to its parent when it is detected. As all children have a greater depth they are detected before the DFS of their parent starts.

*Potential Parents.* Consider the subtree $T_u$ of $u$ and a potential parent $w$ in $T_u$. Let $X_w$ be the set of nodes given by $w$, the ancestors of $w$, the close children of $w$ and the descendants of the close children of $w$. Moving $v_m$ below $w$ requires us to insert an edge from $v_m$ to every non-$v_m$-neighbor in $X_w$. Likewise, not including $v_m$-neighbors in $X_w$ requires us to delete an edge for each of them. We therefore want $X_w$ to maximize the number of $v_m$-neighbors minus the number of non-$v_m$-neighbors. This value gives us a score for each potential parent in $T_u$. We denote by $\text{score}_{\max}(u)$ the maximum score over all potential parents in $T_u$. Note that $\text{score}_{\max}(u)$ is always at least -1 as we can move $v_m$ below $u$ and not adopt any children. We determine in a bottom-up fashion all $\text{score}_{\max}(u)$ that are greater than 0. Whether $\text{score}_{\max}(u)$ is -1 or 0 is irrelevant because isolating $v_m$ is never worse. The final solution will be in $\text{score}_{\max}(r)$ of the root $r$ as its "subtree" encompasses the whole graph. $\text{score}_{\max}(u)$ can be computed recursively. If $u$ is a best parent then the value of $\text{score}_{\max}(u)$ is the sum over the closenesses of all of $u$'s close children $\pm 1$. If the subtree $T_w$ of a child $w$ of $u$ contains a best parent then $\text{score}_{\max}(u) = \text{score}_{\max}(w) \pm 1$. The $\pm 1$ depends on whether $w$ is a $v_m$-neighbor. Unfortunately not only potential parents $u$ have a $\text{score}_{\max}(u) > 0$. However, we know that every node $u$ with $\text{score}_{\max}(u) > 0$ is a $v_m$-neighbor or has a child $w$ with $\text{score}_{\max}(w) > 0$. We can therefore process all $\text{score}_{\max}$ values in a similar bottom-up way using a tree-depth ordered priority

queue as we used to compute $\text{child}_{\text{close}}$. As both bottom-up procedures have the same structure we can interweave them as optimization and use only a single queue. The algorithm is illustrated in Fig. 3a in pseudo-code form.

## 5 Experimental Evaluation

We evaluated the QTM algorithm on the small instances used by Nastos and Gao [14], on larger synthetic graphs and large real-world social networks and web graphs. We measured both the number of edits needed and the required running time. For each graph we also report the lower bound $b$ of necessary edits that we obtained using our lower bound algorithm. We implemented the algorithms in C++ using NetworKit [17]. All experiments were performed on an Intel Core i7-2600K CPU with 32GB RAM. We ran all algorithms ten times with ten different random node id permutations.

*Comparison with Nastos and Gao's Results.* Nastos and Gao [14] did not report any running times, we therefore re-implemented their algorithm. Our implementation of their algorithm has a complexity of $O(m^2 + k \cdot n^2 \cdot m)$, the details can be found in the extended version [6]. Similar to their implementation we used a simple exact bounded search tree (BST) algorithm for the last 10 edits. In Table 1 we report the minimum and average number of edits over ten runs. Our implementation of their algorithm never needs more edits than they reported[1]. For two of the graphs (dolphins and lesmis) our implementation needs slightly less edits due to different tie-breaking rules.

For all but one graph QTM is at least as good as the algorithm of Nastos and Gao in terms of edits. QTM needs only one more edit than Nastos and Gao for the grass_web graph. The QTM algorithm is much faster than their algorithm, it needs at most 2.5 milliseconds while the heuristic of Nastos and Gao needs up to 6 seconds without bounded search tree and almost 17 seconds with bounded search tree. The number of iterations necessary is at most 5. As the last round only checks whether we are finished four iterations would be enough.

*Large Graphs.* For the results in Table 2 we used two Facebook graphs [18] and five SNAP graphs [13] as social networks and four web graphs from the 10th DIMACS Implementation Challenge [1,3,4,5]. We evaluate two variants of QTM. The first is the standard variant which starts with a non-trivial skeleton obtained by the heuristic described in Section 3. The second variant starts with a trivial skeleton where every node is a root. We chose these two variants to determine which part of our algorithm has which influence on the final result. For the standard variant we report the number of edits needed before any node is moved. With a trivial skeleton this number is meaningless and thus we report the number of edits after one round. All other measures are straightforward and are explained in the table's caption.

[1] Except on Karate, where they report 20 due to a typo. They also need 21 edits.

**Table 1.** Comparison of QTM and [14]. We report $n$ and $m$, the lower bound $b$, the number of edits (as minimum, mean and standard deviation), the mean and maximum of number of QTM iterations, and running times in ms.

| Name | $n$ | $m$ | $b$ | Algorithm | Edits | | | Iterations | | Time [ms] | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | min | mean | std | mean | max | mean | std |
| dolphins | 62 | 159 | 24 | QTM | 72 | 74.1 | 1.1 | 2.7 | 4.0 | 0.6 | 0.1 |
| | | | | NG w/ BST | 73 | 74.7 | 0.9 | - | - | 15 594.0 | 2 019.0 |
| | | | | NG w/o BST | 73 | 74.8 | 0.8 | - | - | 301.3 | 4.0 |
| football | 115 | 613 | 52 | QTM | 251 | 254.3 | 2.7 | 3.5 | 4.0 | 2.5 | 0.4 |
| | | | | NG w/ BST | 255 | 255.0 | 0.0 | - | - | 16 623.3 | 3 640.6 |
| | | | | NG w/o BST | 255 | 255.0 | 0.0 | - | - | 6 234.6 | 37.7 |
| grass_web | 86 | 113 | 10 | QTM | 35 | 35.2 | 0.4 | 2.0 | 2.0 | 0.5 | 0.1 |
| | | | | NG w/ BST | 34 | 34.6 | 0.5 | - | - | 13 020.0 | 3 909.8 |
| | | | | NG w/o BST | 38 | 38.0 | 0.0 | - | - | 184.6 | 1.2 |
| karate | 34 | 78 | 8 | QTM | 21 | 21.2 | 0.4 | 2.0 | 2.0 | 0.4 | 0.1 |
| | | | | NG w/ BST | 21 | 21.0 | 0.0 | - | - | 9 676.6 | 607.4 |
| | | | | NG w/o BST | 21 | 21.0 | 0.0 | - | - | 28.1 | 0.3 |
| lesmis | 77 | 254 | 13 | QTM | 60 | 60.5 | 0.5 | 3.3 | 5.0 | 1.4 | 0.3 |
| | | | | NG w/ BST | 60 | 60.8 | 1.0 | - | - | 16 919.1 | 3 487.7 |
| | | | | NG w/o BST | 60 | 77.1 | 32.4 | - | - | 625.0 | 226.4 |

Even though for some of the graphs the mover needs more than 20 iterations to terminate, the results do not change significantly compared to the results after round 4. In practice we can thus stop after 4 rounds without incurring a significant quality penalty. It is interesting to see that for the social networks the initialization algorithm sometimes produces a skeleton that induces more than $m$ edits (e.g. in the case of the "Penn" graph) but still the results are always slightly better than with a trivial initial skeleton. This is even true when we do not abort moving after 4 rounds. For the web graphs, the non-trivial initial skeleton does not seem to be useful for some graphs. It is not only that the initial number of edits is much higher than the finally needed number of edits, also the number of edits needed in the end is slightly higher than if a trivial initial skeleton was used. This might be explained by the fact that we designed the initialization algorithm with social networks in mind. Initial skeleton heuristics built specifically for web graphs could perform better. While the QTM algorithm needs to edit between approximately 50 and 80% of the edges of the social networks, the edits of the web graphs are only between 10 and 25% of the edges. This suggests that quasi-threshold graphs might be a good model for web graphs while for social networks they represent only a core of the graph that is hidden by a lot of noise. Concerning the running time one can clearly see that QTM is scalable and suitable for large real-world networks.

As we cannot show for our real-world networks that the edit distance that we get is close to the optimum we generated synthetic graphs by generating quasi-threshold graphs and applying random edits to these graphs. The details of the

**Table 2.** Results for large real-world and synthetic graphs. Number of nodes $n$ and edges $m$, the lower bound $b$ and the number of edits are reported in thousands. Column "I" indicates whether we start with a trivial skeleton or not. • indicates an initial skeleton as described in Section 3 and ∘ indicates a trivial skeleton. Edits and running time are reported for a maximum number of 0 (respectively 1 for a trivial initial skeleton), 4 and ∞ iterations. For the latter, the number of actually needed iterations is reported as "It". Edits, iterations and running time are the average over the ten runs.

| | Name | $n$ [K] / $m$ [K] | $b$ [K] | I | Edits [K] 0/1 | Edits [K] 4 | Edits [K] ∞ | It ∞ | Time [s] 0/1 | Time [s] 4 | Time [s] ∞ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Social Networks | Caltech | 0.77 / 16.66 | 0.35 | • | 15.8 | 11.6 | 11.6 | 8.5 | 0.0 | 0.0 | 0.1 |
| | | | | ∘ | 12.6 | 11.7 | 11.6 | 9.4 | 0.0 | 0.0 | 0.1 |
| | amazon | 335 / 926 | 99.4 | • | 495 | 392 | 392 | 7.2 | 0.3 | 5.5 | 9.3 |
| | | | | ∘ | 433 | 403 | 403 | 8.9 | 1.3 | 4.9 | 10.7 |
| | dblp | 317 / 1 050 | 53.7 | • | 478 | 415 | 415 | 7.2 | 0.4 | 5.8 | 9.9 |
| | | | | ∘ | 444 | 424 | 423 | 9.0 | 1.4 | 5.2 | 11.5 |
| | Penn | 41.6 / 1 362 | 19.9 | • | 1 499 | 1 129 | 1 127 | 14.4 | 0.6 | 4.2 | 13.5 |
| | | | | ∘ | 1 174 | 1 133 | 1 129 | 16.2 | 1.0 | 3.7 | 14.4 |
| | youtube | 1 135 / 2 988 | 139 | • | 2 169 | 1 961 | 1 961 | 9.8 | 1.4 | 31.3 | 73.6 |
| | | | | ∘ | 2 007 | 1 983 | 1 983 | 10.0 | 7.1 | 28.9 | 72.7 |
| | lj | 3 998 / 34 681 | 1 335 | • | 32 451 | 25 607 | 25 577 | 18.8 | 23.5 | 241.9 | 1 036.0 |
| | | | | ∘ | 26 794 | 25 803 | 25 749 | 19.9 | 58.3 | 225.9 | 1 101.3 |
| | orkut | 3 072 / 117 185 | 1 480 | • | 133 086 | 103 426 | 103 278 | 24.2 | 115.2 | 866.4 | 4 601.3 |
| | | | | ∘ | 106 367 | 103 786 | 103 507 | 30.2 | 187.9 | 738.4 | 5 538.5 |
| Web Graphs | cnr-2000 | 326 / 2 739 | 48.7 | • | 1 028 | 409 | 407 | 11.2 | 0.8 | 12.8 | 33.8 |
| | | | | ∘ | 502 | 410 | 409 | 10.7 | 3.2 | 11.8 | 30.8 |
| | in-2004 | 1 383 / 13 591 | 195 | • | 2 700 | 1 402 | 1 401 | 11.0 | 7.9 | 72.4 | 182.3 |
| | | | | ∘ | 1 909 | 1 392 | 1 389 | 13.5 | 16.6 | 65.0 | 217.6 |
| | eu-2005 | 863 / 16 139 | 229 | • | 7 613 | 3 917 | 3 906 | 13.7 | 6.9 | 90.7 | 287.7 |
| | | | | ∘ | 4 690 | 3 919 | 3 910 | 14.5 | 22.6 | 85.6 | 303.5 |
| | uk-2002 | 18 520 / 261 787 | 2 966 | • | 68 969 | 31 218 | 31 178 | 19.1 | 200.6 | 1 638.0 | 6 875.5 |
| | | | | ∘ | 42 193 | 31 092 | 31 042 | 22.3 | 399.8 | 1 609.6 | 8 651.8 |
| Synthetic | Gen. 160K | 100 / 930 | 42 | • | 200 | 158 | 158 | 4.6 | 0.2 | 3.5 | 4.1 |
| | | | | ∘ | 193 | 158 | 158 | 6.1 | 1.0 | 3.3 | 4.9 |
| | Gen. 0.4K | 1 000 / 10 649 | 0.391 | • | 1.161 | 0.395 | 0.395 | 3.0 | 3.3 | 43.8 | 43.8 |
| | | | | ∘ | 182 | 5.52 | 5.52 | 6.1 | 15.9 | 52.9 | 78.8 |

generation process are described in the extended version [6]. In Table 2 we report the results of two of these graphs with 400 and 160 000 random edits. In both cases the number of edits the QTM algorithm finds is below or equal to the generated editing distance. If we start with a trivial skeleton, the resulting edit distance is sometimes very high, as can be seen for the graph with 400 edits. This shows that the initialization algorithm from Section 3 is necessary to achieve good quality on

graphs that need only few edits. As it seems to be beneficial for most graphs and not very bad for the rest, we suggest to use the initialization algorithm for all graphs.

*Case Study: Caltech.* The main application of our work is community detection. While a thorough experimental evaluation of its usefulness in this context is future work we want to give a promising outlook. Figure 4 depicts the edited Caltech university Facebook network from [18]. Nodes are students and edges are Facebook-friendships. The dormitories of most students are known. We colored the graph accordingly. The picture clearly shows that our algorithm succeeds at identifying most of this structure.

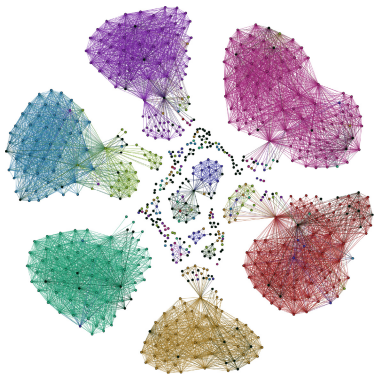


**Fig. 4.** Edited Caltech network, edges colored by dormitories of endpoints.

## 6 Conclusion

We have introduced Quasi-Threshold Mover (QTM), the first heuristic algorithm to solve the quasi-threshold editing problem in practice for large graphs. As a side result we have presented a simple certifying linear-time algorithm for the quasi-threshold recognition problem. A variant of our recognition algorithm is also used as initialization for the QTM algorithm. In an extensive experimental study with large real world networks we have shown that it scales very well in practice. We generated graphs by applying random edits to quasi-threshold graphs. QTM succeeds on these random graphs and often even finds other quasi-threshold graphs that are closer to the edited graph than the original quasi-threshold graph. A surprising result is that web graphs are much closer to quasi-threshold graphs than social networks, for which quasi-threshold graphs were introduced as community detection method. A logical next step is a closer examination of the detected quasi-threshold graphs and the community structure they induce. Further our QTM algorithm might be adapted for the more restricted problem of threshold editing which is NP-hard as well [10] or extended with an improved initialization algorithm, especially for web graphs.

**Acknowledgment.** We thank James Nastos for helpful discussions.

## References

1. Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D.: Graph Partitioning and Graph Clustering: 10th DIMACS Implementation Challenge, vol. 588. American Mathematical Society (2013)

2. Blondel, V., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10) (2008)
3. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: A scalable fully distributed web crawler. Software - Practice and Experience 34(8), 711–726 (2004)
4. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In: Proceedings of the 20th International Conference on World Wide Web (WWW 2011), pp. 587–596. ACM Press (2011)
5. Boldi, P., Vigna, S.: The WebGraph framework I: Compression techniques. In: Proceedings of the 13th International Conference on World Wide Web (WWW 2004), pp. 595–602. ACM Press (2004)
6. Brandes, U., Hamann, M., Strasser, B., Wagner, D.: Fast quasi-threshold editing (2015), http://arxiv.org/abs/1504.07379
7. Cai, L.: Fixed-parameter tractability of graph modification problems for hereditary properties. Information Processing Letters 58(4), 171–176 (1996)
8. Chiba, N., Nishizeki, T.: Arboricity and subgraph listing algorithms. SIAM Journal on Computing 14(1), 210–223 (1985)
9. Chu, F.P.M.: A simple linear time certifying lbfs-based algorithm for recognizing trivially perfect graphs and their complements. Information Processing Letters 107(1), 7–12 (2008)
10. Drange, P.G., Dregi, M.S., Lokshtanov, D., Sullivan, B.D.: On the threshold of intractability. In: Proceedings of the 23rd Annual European Symposium on Algorithms (ESA 2015). LNCS. Springer (2015)
11. Drange, P.G., Pilipczuk, M.: A polynomial kernel for trivially perfect editing. In: Proceedings of the 23rd Annual European Symposium on Algorithms (ESA 2015). LNCS. Springer (2015)
12. Görke, R., Kappes, A., Wagner, D.: Experiments on density-constrained graph clustering. ACM Journal of Experimental Algorithmics 19, 1.6:1.1–1.6:1.31 (2014)
13. Leskovec, J., Krevl, A.: Snap datasets: Stanford large network dataset collection (June 2014), http://snap.stanford.edu/data
14. Nastos, J., Gao, Y.: Familial groups in social networks. Social Networks 35(3), 439–450 (2013)
15. Ortmann, M., Brandes, U.: Triangle listing algorithms: Back from the diversion. In: Proceedings of the 16th Meeting on Algorithm Engineering and Experiments (ALENEX 2014), pp. 1–8. SIAM (2014)
16. Rotta, R., Noack, A.: Multilevel local search algorithms for modularity clustering. ACM Journal of Experimental Algorithmics 16, 2.3:2.1–2.3:2.27 (2011)
17. Staudt, C., Sazonovs, A., Meyerhenke, H.: Networkit: An interactive tool suite for high-performance network analysis (2014), http://arxiv.org/abs/1403.3005
18. Traud, A.L., Mucha, P.J., Porter, M.A.: Social structure of facebook networks. Physica A: Statistical Mechanics and its Applications 391(16), 4165–4180 (2012)
19. Wolk, E.S.: A note on "the comparability graph of a tree". Proceedings of the American Mathematical Society 16(1), 17–20 (1965)
20. Yan, J.H., Chen, J.J., Chang, G.J.: Quasi-threshold graphs. Discrete Applied Mathematics 69(3), 247–255 (1996)