A Parallel 2k Kernel for the Cluster Editing Problem

Frank Boogaard

Master Thesis, ICA-5727731

Supervisor: Prof. Dr. H.L. Bodlaender

Utrecht University
Department of Information and Computing Sciences

July 14, 2020

Abstract

The Cluster Editing problem asks us to edit a graph and turn it into a disjoint union of cliques. We are allowed to both add and remove any edges from the graph to achieve this. The goal is to make as few edits as possible. We will present a parallel version of a $2k$ Cluster Editing kernel by Cheng and Meng [6]. Our parallel version will use $O(n \log n)$ parallel time and $O(mn)$ work. We will show that the rules of the kernel can be applied in parallel without conflicts. Unfortunately this does not guarantee that we apply the rules a constant number of times. There exist certain graphs that still require us to apply the rules a linear number of times. We also provide an implementation of the parallel kernel on a GPU. Our implementation can reduce graphs with a few million edges in just a few seconds. We will show that the rules are applied a constant number of times in practice. One feature of our implementation is that after each rule application we will use the same amount of memory or less, even when we add edges to the graph.

Contents

1 Introduction

Cluster Editing is a well known Fixed Parameter Tractable (FPT) problem. It has mainly been studied in the sequential setting. No research has been done yet that specifically focussed on parallel algorithms for the Cluster Editing problem. In this thesis we will present a parallel kernel for the Cluster Editing problem of size $2k$. Our kernel is based on the sequential $2k$ kernel by Cheng and Meng [6], which uses $O(mn)$ time. We turned it into a parallel kernel that uses $O(n \log n)$ time and $O(mn)$ work. We show that you can apply as many rules as possible at once without conflicts. Unfortunately there are however still situations were we have to apply the rules $O(n)$ times, but this happens rarely in practice. We implemented our algorithm on a GPU. Our machine can reduce graphs with a few million edges in just a few seconds. One feature of our implementation is that it does not add any edges to memory. Even when the kernel tells us to add many edges to the graph, we will use less memory after each rule application.

The $2k$ kernel by Cheng and Meng [6] uses critical cliques and their editing degree to reduce the graph. The kernel looks at groups of vertices that almost form a disjoint clique. It uses the editing degree to find these critical cliques. The original kernel by Cheng and Meng [6] recalculates the critical cliques each time after a rule application. We came up with a way to update the critical cliques instead. This makes the kernel more efficient in practice. Existing algorithms for finding critical cliques are mostly theoretical algorithms that do not work very well in practice. By updating instead of recalculating we can use a theoretically less efficient algorithm that works well in practice for finding the initial critical cliques.

We will first discuss the relevance of the Cluster Editing problem in Section 2. Section 3 will discuss all definitions and notations we use throughout this thesis. In Section 4 we will discuss related work and we will explain the sequential kernel of Cheng and Meng [6] in more detail. We give a short overview of the parallel kernel in Section 5. Algorithms for finding critical cliques will be discussed in Section 6. Section 7 will describe how we can efficiently calculate the editing degrees. We prove in Section 8 that all rules can be applied in parallel and in Section 9 we will discuss the complexity of the parallel kernel. We discuss the results of our implementation in Section 10 and conclude our work in Section 11.

2 Relevance

The Cluster Editing problem is defined as follows. Given: an undirected graph $G = (V, E)$ and an integer k. Question: Can we transform G, by inserting and deleting at most k edges, into a graph that consists of a disjoint union of cliques? The smallest value for k for which this is possible is called the *Cluster Edit Distance*.

Cluster Editing is a problem that is used in many areas of research. It can for example be used to fix inconsistencies in research data. In order to make research data consistent, we want to edit as few things as possible. Biologists use Cluster Editing to cluster gene expression data for tissue classification or to cluster biological entities such as proteins [22, 23]. It is also used for gene regulatory network analysis [10].

Cluster Editing can also be used in the field of phylogenetics [8]. Here we are interested in constructing a tree showing the evolutionary relationships of different species. The tree is constructed by looking at the similarities and dissimilarities between the biological entities. We identify the different species by finding clusters in the biological data. These species then form the leafs of the phylogenetic tree.

In the context of machine learning Cluster Editing is also called Correlation Clustering. Here we want to classify the data in an optimum number of clusters. Unlike other clustering algorithms, we however do not know the amount of clusters we are looking for. We are given a weighted graph that encodes the relation between the entities. Similar entities are connected with a positive weighted edge. If two entities are dissimilar, they are connected with a negative weighted edge. We then want to maximize the sum of positive weighted edges within a cluster plus the absolute value of the sum of the negative edges between the clusters. CORRELATION CLUSTERING can for example be used for classifying webpages or documents [1].

3 Definitions and Notation

In order to make our figures more clear we decided to simplify them. We draw two types of edges. If an edge is connected to two vertices then it just connects these two edges. If and edge hovers between two groups of vertices then it represents all the edges between those groups. All vertices of the group are connected to all vertices of the other group. See Figure 1 for an example.

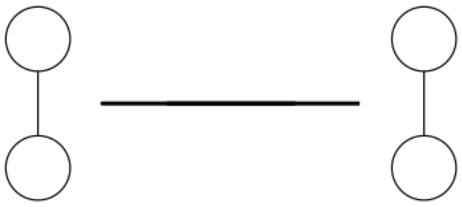

Figure 1: Example of how we illustrate graphs. This is a complete graph. The floating edge denotes that all edges between the vertices on the left and the vertices on the right are there.

We will be parallelizing a CLUSTER EDITING kernel that uses *critical cliques* and their *editing degree* to reduce the graph. A critical clique is a maximal set of fully connected vertices with the same neighbors. The editing degree of a critical clique will tell us how many edits we need in order to turn the critical clique and its neighbors into a disjoint clique. We will be using the same definition for critical cliques and their editing degree as Cheng and Meng [6].

Definition 1. A *critical clique* K in a graph G is a clique such that for all vertices v and w in K, $N(v) \setminus K = N(w) \setminus K$ and K is maximal under this condition.

Definition 2. Let K be a critical clique in a graph G and let $v \in N(K)$. The *editing degree* $p_K(v)$ of v with respect to K is defined to be the number of vertex pairs $\{v, w_1\}$, where $w_1 \in N(K) \setminus \{v\}$ and $w_1 \notin N(v)$, plus the number of edges $[v, w_2]$, where $w_2 \notin K \cup N(K)$.

Every vertex in a graph belongs to a unique critical clique. We can thus construct a *critical clique graph* G_c for a graph G where the vertices of G_c correspond to the critical cliques. An edge is present in G_c if the union of the two critical cliques form a larger clique in G. In Figure 2 we illustrate a graph and its corresponding critical clique graph. We have also calculated the editing degree for each critical clique in Figure 2.

In our algorithm we use the *augmented neighborhood* of a vertex to find critical cliques in a graph. The augmented neighborhood is defined as follows:

Definition 3. The *augmented neighborhood* of a vertex v is the set $N(v) \cup \{v\}$.

A kernelization algorithm is based on the repeated application of a number of rules. Each rule is applied to a graph G and integer k and emits a smaller graph G' and possibly also a smaller integer k'. All rules are safe which means that G can be turned into a cluster graph with at most k edits if and only if G' can be turned into a cluster graph with at most k' edits. Each rule of Cheng and Meng's kernel is given a critical clique K as an additional input. These rules are given in Section 4 and will be numbered. For now we would like to introduce *r-applicable* critical cliques.

Definition 4. A critical clique K is *r-applicable* if Rule r can be applied to K.

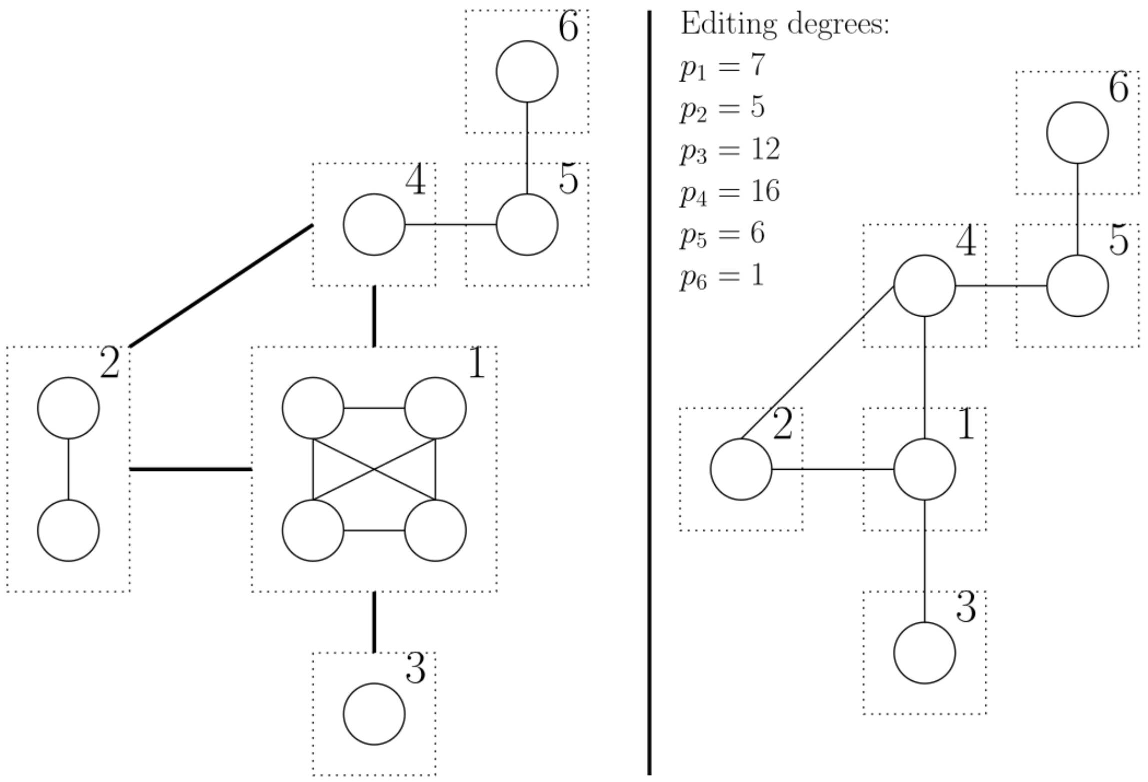

Figure 2: Example of a critical clique graph. On the left you can see the original graph and on the right the corresponding critical clique graph. Note that missing edges in $N(K)$ are counted twice when we calculate p_K.

We will use the following notations regarding graphs, critical cliques and editing degrees:

Notation	Meaning
n	The number of vertices in the graph.
m	The number of edges in the graph.
$d(v)$	The degree of a vertex v.
k	The parameter of our FPT problem. We are allowed to do at most k edits.
k_{opt}	The Cluster Edit Distance (the smallest value of k that gives us a solution).
K	The set of vertices in a critical clique.
$N(K)$	The set of vertices that are a neighbor of critical clique K.
$N_2(K)$	The set of vertices adjacent to any vertex in $N(K)$ (excluding K).
$p_K(v)$	The editing degree of vertex v with respect to critical clique K.
p_K	$\sum_{v \in N(K)} p_K(v)$. The total editing degree of critical clique K.
$p_v(w)$	The editing degree of vertex w with respect to the critical clique vertex v is in.

4 Related Work and the Sequential Kernel by Chen and Meng

Downey and Fellows [12] introduced the notion of *Fixed Parameter Tractability* (FPT). Consider a parameterized problem $Q \subseteq \Sigma^* \times \mathbb{N}$. Q is parameter tractable if there exists an algorithm that can decide for a given input $(x, k) \in \Sigma^* \times \mathbb{N}$ whether $(x, k) \in Q$ in time $O(f(k) \cdot n^{O(1)})$. Here n is the size of the input x, k is called the *parameter* and f is a computable function. FPT algorithms are always polynomial in n, but they can be exponential in k. The parameter k usually tells us something about the size of the solution. For the k-Vertex Cover problem we are for example only interested in vertex covers of size at most k.

A *kernel* K is a polynomial computable function that reduces the problem instance to an equivalent instance of size $s_K(k)$. s_K can be any computable function. The original input has a solution if and only if the reduced instance also has a solution. A problem is FPT if and only if it has a kernel. Because the size of the problem only depends on k after the kernelization, we can exhaustively search the remaining input for a solution.

In this section we will give a short overview of existing sequential kernels for the Cluster Editing problem. We will explain the $4k$ kernel by Guo [16] and the $2k$ kernel by Cheng and Meng [6] in more detail, because our parallel kernel is based on their ideas.

(Non-)Common neighbors: $2k^2 + k$

Gramm et al. [15] were the first to give a kernel for the Cluster Editing problem. Their kernel has a size of $2k^2 + k$ vertices and $2k^3 + k^2$ edges. It runs in $O(n^3)$ time. The kernel looks at the amount of common and non-common neighbors of two vertices. If the two endpoints of an edge have more than k non-common neighbors, then we will only find a solution if we remove that edge from the graph. Similarly if a pair of not connected vertices have more than k common neighbors, then we will only find a solution if we create an edge between those vertices.

Crown Decomposition: $24k$

Fellows et al. [14] were the first to come up with a linear kernel for the Cluster Editing problem. Given a solution of the Cluster Editing problem, we can classify the vertices into three types. Type A vertices have at least one neighboring edge addition in the solution. Type B vertices are not of type A and have at least one neighboring edge deletion in the solution. The remaining vertices are of type C. Fellows et al. [14] noted that if a clique of the solution has enough vertices of type C, then a reduction rule might apply. As it turns out we can use cluster crown decompositions to do just this.

A *cluster crown decomposition* of a graph $G = (V, E)$ partitions V in four disjoint subsets (C, H, N, X). C is a cluster in the graph and all vertices in C are adjacent to all vertices in H. H is a cut-set of the graph (there are no edges between C and $V \setminus H$). N is the set of vertices that are adjacent to the vertices in H (excluding vertices in C). And X contains all other vertices.

Fellows et al. [14] used a 4-approximation algorithm for the Cluster Editing problem to find a cluster crown decomposition. Picking vertices of type C from a single clique in the solution will give you the C partition. Vertices of type A and B within that same clique will form the H partition. From there you can easily deduce the vertices that are part of partition N and X as well.

The kernel only has one rule given a cluster crown decomposition (C, H, N, X) of the graph:

1. If $|C| \geq |H| + |N| - 1$, then remove C and H from the graph and set $k' = k - e - f$. Here e denotes the number of edges needed to turn $C \cup H$ into a clique and f denotes the number of edges between H and N.

Fellows et al. [14] show that this kernel has a size of $24k$ by using the fact that the 4-approximation algorithm has at most $8k$ vertices of type A or B. The number of vertices in a H or N partition is then at most $8k$. Which means that if the reduction rule does not apply, the number of vertices in a C partition is less than $16k$. The graph therefore has a size smaller than $16k + 8k = 24k$. Later Fellows et al. [13] found a way to calculate the cluster crown decompositions in polynomial time. This resulted in a $6k$ kernel.

Critical Cliques: $4k$

Guo [16] came up with a kernel of size $4k$. The kernel is based on the idea of *critical cliques*. Critical cliques are similar to the cluster crown decompositions: a critical clique K (see Definition 1) is a clique and all vertices in this clique have the same neighbors. K has to be maximal under this condition.

The set of unaffected vertices contained in a single clique of the optimal solution forms a critical clique in the original graph (the type C vertices of the $6k$ kernel). This suggest that working with critical cliques might be easier than working with the input graph. This also means that we can use the critical clique graph as a weighted input for FPT algorithms, because we will never edit an edge within a critical clique (the definition of a type C vertex).

Guo [16] showed that the number of vertices in a critical clique graph is bounded by $4k_{opt}$, where k_{opt} is the smallest value for k for which we will find a solution. This means that just calculating the critical clique graph already gives us a generalized kernel of size $4k$ when we can use a weighted FPT algorithm for the Cluster Editing problem. Fortunately Böcker [4] designed a fast algorithm for the Weighted Cluster Editing problem which is also currently the fastest algorithm for the Cluster Editing problem. However when Guo designed the $4k$ kernel, such a weighted algorithm did not exist yet. In order to obtain a $4k$ kernel for the regular Cluster Editing problem Guo [16] designed the following set of rules:

1. Remove all vertices that are part of an isolated critical clique from the input graph.

2. If $|K| > |N(K)| + |N^2(K)|$, then remove $K \cup N(K)$ from the graph and reduce k by the amount of graph edits you need to turn $K \cup N(K)$ into a disjoint clique.

Applying these first two rules will yield a kernel of size $6k$ [16]. The first two rules are essentially the same as the rules for the $6k$ kernel we discussed previously.

3. Suppose $|K| \geq |N(K)|$ and let K' be a neighboring critical clique of K. If the set of edges between K' and $N^2(K)$ is not empty, and the number of edges between K' and $N^2(K)$ plus the number of edges between K' and $N(K)$ is smaller than $|K| \cdot |K'|$, then we can remove all the edges between K' and $N^2(K)$ and decrease k accordingly.

4. If $|K| \geq |N(K)|$ and $N^2(K) = \emptyset$, then remove the vertices in $N(K)$ from the input graph and decrease k by the number of edges needed to turn $K \cup N(K)$ into a disjoint clique.

Combining Rule 1 with Rule 3 and 4 will yield a kernel of size $4k$ [16].

Critical Cliques: $2k$

Chen and Meng [6] came up with the smallest known Cluster Editing kernel to date. It has a size of $2k$ and a complexity of $O(mn)$. It also uses the concept of critical cliques. In addition, this kernel uses the *editing degree* (see Definition 2) of the critical cliques to improve on the bound of the $4k$ kernel by Guo [16]. We will be parallelizing this $2k$ kernel.

The editing degree $p_K(v)$ is equal to the amount of edges we have to add in order to connect v with all vertices in $N(K)$ plus the amount of edges we have to remove in order to disconnect v from all vertices that are not in $K \cup N(K)$. The editing degree of a critical clique K is then defined as follows: $p_K = \sum_{v \in N(K)} p_K(v)$. Note that with this definition p_K counts all edges that have to be added to the graph twice.

The $2k$ kernel has the following 5 rules:

Rule 1. If $|K| > k$, then turn $K \cup N(K)$ into a disjoint clique, remove it from the graph and adjust k accordingly.

Rule 2. If $|K| \geq |N(K)|$ and $|K| + |N(K)| > p_K$, then turn $K \cup N(K)$ into a disjoint clique, remove it from the graph and adjust k accordingly.

Rule 3. If $|K| < |N(K)|$ and $|K| + |N(K)| > p_K$, and if there is no vertex in $N_2(K)$ which is adjacent to more than $(|K| + |N(K)|)/2$ vertices in $N(K)$, then turn $K \cup N(K)$ into a disjoint clique, remove it from the graph and adjust k accordingly.

Rule 4. If $|K| < |N(K)|$ and $|K| + |N(K)| > p_K$, and if there is a vertex $u \in N_2(K)$ which is adjacent to more than $(|K| + |N(K)|)/2$ vertices in $N(K)$, then make $K \cup N(K)$ a clique and remove the edges between $N(K)$ and $N_2(K) \setminus \{u\}$. Decrease k accordingly.

Rule 5. If $|K| < |N(K)|$ and $|K| + |N(K)| > p_K$, and Rule 1-4 can not be applied, then call the Pendulum algorithm on K.

The Pendulum algorithm mentioned in Rule 5 is also a reduction algorithm. Cheng and Meng showed that when Rules 1-4 can not be applied anymore, $N_2(K)$ will contain a single vertex when $|K| + |N(K)| > p_K$. The reduction rules of the Pendulum algorithm uses this fact. The Pendulum algorithm has two rules:

1. If $|K| \geq |N(K)|$, then turn $K \cup N(K)$ into a disjoint clique and adjust k accordingly.
2. If $|K| < |N(K)|$, then choose $|K|$ vertices from $N(K)$ and remove them together with K from the graph. Decrease k by $|K|$.

Since $N_2(K)$ contains only a single vertex, we know that when we can apply the first rule of the Pendulum algorithm, we could also apply Rule 2. We will thus only need the second rule of the Pendulum algorithm.

With these rules we iterate at most $O(n + k)$ times, because the rules reduce the number of vertices by at least one or reduce k by at least one. Each iteration we have to calculate a critical clique graph, which is linear in the number of edges [19] as we will discuss in Section 6. This results in a time complexity of $O(mn)$.

The editing degrees of the critical cliques can change each time we apply a rule. Cheng and Meng [6] however do not give us much detail on how we can calculate and update the editing degree efficiently. In Section 7 we will further explain how the editing degree can be calculated to fit the time complexity of $O(mn)$.

Later Cao and Chen [5] came up with an improved $2k$ kernel. They managed to obtain a running time of $O(n^2)$ by using graph edge-cuts.

5 The Parallel Kernel

The goal of our parallel kernel is to apply as many rules at once. Each time we want to apply a Rule r, we look for all r-applicable critical cliques in the graph and apply Rule r to all of them at once. In this way we have to recalculate the critical cliques and editing degrees less often. In order to further improve the speed of the kernel we decided to update the critical cliques and editing degrees instead of recalculating them. This way we will avoid recalculating editing degrees of critical cliques of which we already know that they will stay the same. Another benefit is that we can theoretically get away with using a less efficient algorithm for the initial critical clique graph construction.

Algorithm 1 describes our kernel. We first sort the neighbors of all vertices in the graph, because it will make finding critical cliques easier and it will also help with updating the editing degrees. We then find all the critical cliques in the graph. The next step is to calculate the editing degrees. Then we keep looping over all the rules. After each rule application we will update the critical cliques and their editing degrees.

Rule 2, 3 and 4 are very similar to each other and could be applied at once. We will go into more detail on this in Section 8. We however chose to apply Rule 2 separately from Rules 3 and 4, because Rule 2 is much faster on its own. For Rule 3 and 4 we need to do some additional calculations that are not needed for Rule 2. On a graph with 4900 vertices and almost 3 million edges Rule 2 will take roughly 0.08 seconds whereas Rule 3 and 4 take roughly 0.49 seconds. We hope that by first applying Rule 2 we can already remove some edges and vertices from the graph, which can make the calculations of Rule 3 and 4 faster as well.

Algorithm 1: The Parallel Kernel

```
Input: (G): An undirected graph G
Output: All edits for the $2k$ kernel.
G.SortNeighbors();
G.FindCriticalCliques();
G.CalculateEditingDegree();
while true do
    for $i = 1; i <= 5; i++$ do
        if $i == 5$ && no-edits-from-rule1-4 then
            G.ApplyRule(5);
        else
            G.ApplyRule(i);
        end
        G.UpdateEditingDegree();
        G.UpdateCriticalCliques();
        if G.empty || !G.rule-can-be-applied then
            return;
        end
    end
end
```

6 Finding Critical Cliques in Parallel

In this section we will discuss how you can efficiently find critical cliques in parallel. We will first discuss our theoretical results in Section 6.1. In Section 6.2 we will focus on practical aspects of finding critical cliques in parallel.

6.1 Theoretical Algorithms for Finding Critical Cliques in Parallel

Critical cliques are closely related to modular decompositions. A *modular decomposition* partitions the vertices of an undirected graph $G = (V, E)$ in modules. All vertices within a module $S \subset V$ have the same set of neighbors. That is $\forall v \in S : N(S) = N(v) \setminus S$. A module is a bit more flexible than a critical clique, because it also allows vertices within a module to *not* be connected to each other. There are three types of modules: *Type I* modules are modules that also form a clique, *Type II* modules are modules that are connected, but not of Type I and *Type III* modules are independent sets with the same neighbors. We are only interested in the Type I modules, because they are the same as the critical cliques we are looking for.

Hsu and Ma [19] gave an $O(n + m)$ algorithm for modular decomposition (and thus also for finding critical cliques). In order to find Type I modules the algorithm partitions the graph by using the augmented neighborhood (see Definition 3) of the vertices. If there is a partition with more than one vertex after ordering the vertices, then we have found a Type I module (and thus also a critical clique). This ordering we found of the vertices based on their augmented neighbors is also called a *lexicographical ordering.*

A lot of research has been done on how to find modular decompositions in linear sequential time [21, 17, 19]. Parallel modular decomposition however is a less well studied topic. Elias Dahlhaus [7] is one of the few who published a parallel algorithm for modular decomposition. The algorithm is based on a divide an conquer strategy and uses $O(\log^2 n)$ parallel time with $O(n + m)$ processors on a CRCW-PRAM. This algorithm performs $O(\log^2 n \cdot (m + n))$ work. The algorithm is however rather complicated and has a lot of different cases, so it might not be the best algorithm to use in practice.

Finding lexicographical orderings in parallel is a broader studied subject. Lexicographical ordering also has many applications. Sorting strings is an example of finding a lexicographical ordering that is used in many areas. Theoretical work of Hagerup [18] shows that a sequence of strings with s characters in total can be sorted in $O(\log s/\log\log s)$ parallel time using $O(s\log\log s)$ operations. Hagerup's algorithm combines pairs of adjacent characters in order to achieve this complexity. We can use Hagerup's algorithm to find a lexicographical ordering of the vertices based on their augmented neighborhood. Each edge occurs in the augmented neighborhood of two vertices, so s will be equal to $n + 2m$ in our case. We get a complexity of $O(\log(n+m)/\log\log(n+m))$ parallel time with $O((n+m)\log\log(n+m))$ work. This is already an improvement compared to Dahlhaus's [7] algorithm.

We now only have to find a way to extract the critical cliques from the lexicographical ordering in parallel. We know that vertices with the same neighbors will be next to each other in this ordering. Algorithm 2 describes how you can find critical cliques in parallel given a lexicographical ordering of the vertices based on their augmented neighborhood. The algorithm works as follows: for each vertex v_i in the ordering we will compare the augmented neighborhood with the vertex that is placed before v_i in the ordering: v_{i-1}. First we will check whether the degree of v_i and v_{i-1} are the same. If this is not the case, then we set a flag for vertex v_i to 1. For all vertices that do not have a flag that equals 1 after this step, we will do the following: the ith neighbor of v_i will be assigned to check whether it is the same as the ith neighbor of v_{i-1}. If this is not the case, then we set a flag for vertex v_i to 1. Now all vertices with a different augmented neighborhood than the vertex before it in the ordering will have a flag with value 1. All other vertices will have a flag with value 0. Flagging all the vertices will take $O(1)$ parallel time with $O(n+m)$ work. We can now calculate a prefix sum in parallel of these flag values to get the index of the critical clique for each vertex. A prefix sum can be calculated in $O(\log n)$ parallel time and $O(n)$ work.

Algorithm 2: Find Critical Cliques in Parallel

```
Input: (V, A): V are the vertices in the graph, and A contains the augmented
       neighborhood for each vertex.
Output: All critical cliques in the graph.
1  A = ParallelNeighborSort(A);    // Sort the augmented neighborhood for each vertex.
2  O = ParallelVertexSort(V, A);   // Find the lexicographical ordering of the vertices.
3  N;                              // Array filled with n zeros.
4  do in parallel
5      v = O[i];
6      w = O[i - 1];
7      if $d(v) == d(w)$ then
8          do in parallel
9              // Check whether the j-th neighbor is different for v and w.
10             if $A[v][j]$ != $A[w][j]$ then
11                 N[v] = 1;
12             end
13         end
14     else
15         N[v] = 1;
16     end
17 end
18 return CalcPrefixSum(N);
```

Theorem 1. All critical cliques in a graph can be found in $O(\log(n+m))$ parallel time with

$O((n+m)\log\log(n+m))$ work.

Proof. We can use Algorithm 2 to find all the critical cliques. Sorting the augmented neighborhood of each vertex can be done in $O(\log(n+m))$ parallel time and $O(n+m)$ work using a parallel radix sort algorithm. Setting the flags for all the vertices takes $O(1)$ time and $O(n+m)$ work. A prefix sum can be calculated in $O(\log n)$ parallel time and $O(n)$ work. If we then use Hagerup's [18] algorithm to find the lexicographical ordering of the vertices, we get a total complexity of $O(\log(n+m))$ parallel time with $O((n+m)\log\log(n+m))$ work for finding all critical cliques. $\square$

6.1.1 Updating the Critical Cliques

The critical clique graph can change every time we remove or add edges to the graph. Cheng and Meng [6] decided to construct a new critical clique graph after each rule application. It is however also possible to update the critical cliques based on the edits. We designed a method to update the critical cliques in $O(1)$ parallel time and $O(m)$ work. From a theoretical point of view this does not directly improve the complexity of the kernel. In the worst case, we have to apply the rules $O(n)$ times as we will explain in Section 9, so we still use $O(mn)$ work. However, this does allow us to use a theoretically less efficient algorithm to construct the initial critical clique graph. As long as this algorithm uses $O(mn)$ work or better, the total complexity of finding and updating the critical clique graph remains $O(mn)$. From a practical point of view updating the critical clique graph is better than reconstructing it every iteration. We will do significantly less work when only a small portion of the graph is edited after applying a rule.

Rule 4 is the only rule that does not immediately remove the edited critical clique from the graph. Let K be a critical clique that is 4-applicable. We can easily figure out what the critical cliques will be for the vertices in $K \cup N(K)$ after applying Rule 4. These vertices either have the same neighbors as the vertices in K, or they are connected to an additional vertex u. By looking at the degree of each vertex in $N(K)$ after applying Rule 4 we can easily decide which of the two is the case and update their critical cliques accordingly.

Lemma 1 (Lemma 2.2 in [6])**.** *Let K be a critical clique. For any vertex v in $N(K)$, $p_K(v) \geq 1$.*

In order to update the critical cliques for the other vertices in the graph we have to do more work. The editing degree between two connected vertices tell us something about whether they have the same neighbors or not. Let (v, w) be an edge in the graph. By Lemma 1 we know that v and w must be in the same critical clique if $p_v(w) = 0$. We will use this observation to update the critical cliques. If the editing degree of an edge (v, w) becomes zero after applying a rule, then we will merge the two critical cliques of v and w. For now we will assume that the editing degree is already being updated. In Section 7.2 we will explain how we will be doing this exactly.

Consider the situation in Figure 3. Rule 2 is applied to vertex f and g, which resulted in some edits. The dashed edges and vertices are removed from the graph in Figure 3. The vertices a, b and c all have the same neighbors after editing the graph. The editing degree between those vertices thus all became 0. We will choose the critical clique with the smallest identifier to become the new critical clique identifier of a, b and c. In order to do this we can not just simply look at the edges with an editing degree of zero and then change the critical clique of both endpoint to the smallest critical clique identifier. In the example of Figure 3 we can then end up assigning different identifiers to the vertices. Algorithm 3 describes how we will update the critical cliques based on the editing degrees. In the first step we will look at all the edges that have an editing degree of zero and that are not already in the same critical clique. We will set a flag for the vertex with the biggest critical clique identifier. After this step the only vertices without a flag will be those with the smallest critical clique identifiers. In the next step of the algorithm we only consider edges with and editing degree of zero where one of

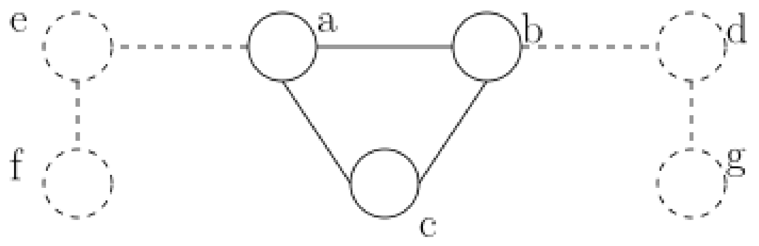

Figure 3: Example of an edited graph where we have to assign new critical clique identifiers to vertex a, b and c. We have to make sure that they will all get the same identifier. Dashed lines denote vertices and edges that are removed from the graph.

the two endpoint does not have a flag. For these edges we can safely change the critical cliques identifiers. It is not hard to see that this will take $O(1)$ parallel time and $O(m)$ work

Algorithm 3: Update Critical Cliques

```
Input: (C, P, E): C contains a critical clique id for each vertex, P contains the editing
       degree values between the critical cliques, and E is the set of edges.
Output: Updated set of critical clique identifiers for each vertex.
do in parallel
    (v, w) = E[i];    // Get the edge.
    int cc1 = C[v];   // Critical clique identifiers.
    int cc2 = C[w];
    if $cc1 \neq cc2$ && $P[cc1, cc2] == 0$ then
        VertFlags[$cc1 < cc2$ ? $w : v$] = 1;
    end
end
do in parallel
    (v, w) = E[i]; // Get the edge.
    int cc1 = C[v];
    int cc2 = C[w];
    if $cc1 \neq cc2$ && $P[cc1, cc2] == 0$ then
        if $!VertFlags[v]$ then
            C[w] = C[v];
        end
        if $!VertFlags[w]$ then
            C[v] = C[w];
        end
    end
end
return $C$;
```

6.2 Practical Implementation for Finding Critical Cliques in Parallel

The algorithms of Hagerup [18] and Dahlhaus [7] both are not the best algorithms you can use in practice. As we already mentioned Dahlhaus's algorithm has a lot of cases, which makes it not very suited to use in practice. Bingmann et al. [2] noticed that Hagerup's [18] algorithm is sub-optimal compared to sequential cases unless $\mathcal{D} = O(s) = O(S)$. Here $\mathcal{D}$ is a lower bound on the execution time of sequential string sorting, s is the number of strings and S is the total length of the strings. For our inputs we can not always guarantee that this is the case. If we use a graph as input that is very close to a complete graph, then $O(S) = O(s^2)$.

Fortunately a substantial amount of practical research has been done on finding lexicographical orderings in parallel. Bingmann et al. [2] for example implemented various parallel sorting algorithms such as multikey quicksort and radix sort as well as a parallel string sample sort algorithm. Davidson and et al. [9] specifically focussed on parallel string sorting algorithms that run on a GPU. Davidson et al. [9] based their approach on a parallel merge sort algorithm. Later Deshpande and Narayanan [11] came up with a faster implementation based on a parallel radix sort algorithm. Their algorithm was also specifically designed for a GPU. According to Deshpande and Narayanan [11] their radix sort approach scales better with large data sets compared to the work of Davidson et al. [9].

We could not find a direct comparison of the algorithms of Bingmann et al. [2] and the algorithm of Deshpande and Narayanan [11]. We decided to use the algorithm of Deshpande and Narayanan [11], because it is specifically designed for the GPU.

Deshpande and Narayanan's [11] algorithm works as follows: for each vertex we load the first two (or more if possible) neighbors in a single integer. We then sort this list of integers using a parallel radix sort algorithm. The next step is to filter out all vertices that are already sorted correctly. After the filter step we load two (or more) new neighbors in a single integer for the vertices that are not filtered out. We repeat these steps until all vertices are sorted. There are two ways we can know that a vertex is in its correct spot: it either has no more neighbors, or it is the only item in its radix sort bucket. Deshpande and Narayanan [11] use a library for the parallel radix sort, so they have to find out themselves whether a vertex is the only item in a bucket. They do this by storing an additional segment-id for the values that they are sorting. All steps in their algorithm that you have to do to update these segment-id are not needed when you implement your own parallel radix sort algorithm.

We added an extra first sorting round to the algorithm where we first sort all the vertices based on their degree before we start looking at the neighbors. This way we hope that we can filter out vertices earlier when they have a lot of common neighbors, but a different degree.

In our case Deshpande and Narayanan's [11] algorithm will use $O(n \log n)$ parallel time with $O(n^2)$ work in the worst case. This happens when the graph is a complete graph. In this case we can never filter out a vertex so we will be sorting n elements n times. However, as the input graph gets less connected, the algorithm will finish more quickly. If we combine this algorithm with Algorithm 3 then we get a total complexity of $O(n + n \log n)$ time and $O(n^2 + mn)$ work for finding and updating the critical cliques.

6.2.1 Parallel Radix Sort

Deshpande and Narayanan [11] used a fast radix sort library to do their radix sorts. We decided to implement our own parallel radix sort algorithm, so that we can do some problem specific optimizations. In this Section we will discuss the radix sort algorithm we implemented. In our implementation we made sure that we can easily see whether a vertex is already sorted correctly. On top of that our radix sort implementation allows us to directly calculate the size of the critical cliques and set the flags of Algorithm 2 in $O(1)$ parallel time with $O(n)$ work. An existing parallel radix sort library will not give us such freedom.

Our radix-sort implementation is based on the ideas of Blelloch [3]. As input we are given an array with values that are encoded with w bits. The radix sort algorithm will iterate w times. In iteration i we will place the values in a bucket based on the i-th bit. We do this by calculating a prefix sum. First we will calculate a prefix sum for all values where the i-th bit is 1. We will place these values first in the bucket. The values of the prefix sum will tell us directly where to place the values in the bucket. Let x be the number of values where the i-th bit is 1. We will also calculate a prefix sum for the values where the i-th bit is 0 and place them in the bucket starting at index x. We can then split the bucket in two: one bucket from index zero to index x and a bucket from index x until the end. We then sort each bucket separately and in parallel in the next iteration.

We will store the values in one array. For each element in this array we store two additional values. One value that tells us at which index the bucket starts for this element, we will call this the min-index. We store another value that tells us where the buckets ends for this element, we will call this the max-index. Whenever we split a bucket in two, these values are updated. When we are done sorting we can use the min- and max-index to find out how big the buckets are. When we are using Deshpande and Narayanan's [11] algorithm to sort the vertices, these sizes will directly tell us how big the critical cliques are. After each iteration of Deshpande and Narayanan's [11] algorithm we can also use the min- and max-index values to determine whether a vertex is already sorted correctly. The position of a vertex inside a bucket of size 1 will namely never change.

We optimized our implementation by using the faster local memory of the GPU. The GPU splits the work into different work-groups. A work-group has a certain number of threads that perform the same calculations. Each work-group will be responsible for calculating the prefix sum of a certain part of the array. Within a work-group we can use the faster local memory of the GPU to calculate the prefix sum of the sub-array. We then have to combine the results of all work-groups to find the total prefix sum. We can recursively do this by again calculating a prefix sum over the values we get from each work-group. When sorting the vertices with Deshpande and Narayanan's algorithm it however turned out that combining the local prefix sum in a quadratic manner was faster in practice. By picking a bigger work size, we make sure that we have to combine less local prefix sum results. It is then faster to loop over the prefix sums of the work groups to combine the results. Graphs we use as input have at most 5000 vertices. We found that using a work-group size of 256 was optimal for our machine. We then have to combine at most $\frac{5000}{256} \approx 19$ results of the work groups for each work group. This of course only happens at the start. Once the buckets get smaller and fit inside a work group, then we do not have to combine anything.

With further optimizations we merged the min- and max-index values in a single integer. The first 16 bits of this integer tell us how far left we have to go in the array to find the min value and the last 16 bits tell us how far right we have to go to find the max value. This safes us 1 memory access for each value we are sorting and thus also results in potentially less cache misses. We also merged the calculation of the two prefix sums by calculating them in the same integer at once. The first 16 bits will contain the sum of the 1 bits prefix sum and the last 16 bits will contain the prefix sum of the 0 bits. This does mean that we can not have an input graph with more than 2^{16} vertices. None of our input graphs come close to this many vertices, so we decided that this would be a fair assumption to make.

One problem with our implementation is that we move all our data in global memory in each round. Memory accesses for placing the values in their new location are random. Random memory accesses patterns are not efficient, because there will be a lot cache misses. Another term for moving the elements in an array like this is *scattering*. Satish et al. [20] addressed this problem of Blelloch's [3] way of doing a radix sort. In order to further improve our radix sort we could have used the ideas of Satish et al [20].

7 Calculating the Editing Degree

The editing degree of the critical cliques can change when we apply the rules. We thus have to keep the editing degree values updated. Cheng and Meng [6] however do not give us much detail on how we can calculate and update the editing degree in $O(mn)$ time. We will therefore first explain how you can calculate and update the editing degree sequentially in Section 7.1. In Section 7.2 we will discuss how you can parallelize these calculations.

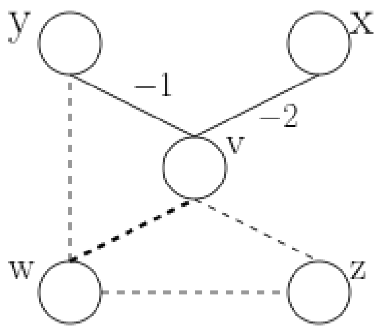

Figure 4: Example of how we will be updating the editing degrees. Dashed lines denote edges that are removed from the graph. We will decrease the editing degree by one for each uncommon neighbor of the vertex that is being removed. x is an uncommon neighbor of w and z, so we decrease its editing degree by 2. y is only an uncommon neighbor of z, so we decrease its editing degree by 1.

7.1 Sequentially Calculating the Editing Degree

The editing degree of vertex v with respect to vertex w, $p_w(v)$, is the same as the number of uncommon neighbors of v and w. We namely have to connect v with all neighbors of w, for this we need $|N(w) \setminus N(v)|$ edits. We also need $|N(v) \setminus N(w)|$ edits to remove the edges that are connected to v, but not to w. For the total editing degree we are thus summing over the amount of uncommon neighbors in the graph. You can calculate the amount of uncommon neighbors if you know the amount of common neighbors of two vertices. Calculating the editing degrees in a graph is thus related to counting triangles in a graph. It is known that triangle counting can not be faster than $O(m^{\frac{2}{3}})$, because there exists a graph that has this many triangles. This means that calculating the editing degrees of the graph is also bounded by $O(m^{\frac{2}{3}})$. The complexity of the kernel will thus not be $O(mn)$ anymore if we recalculate the editing degrees each time we apply a rule. Cheng and Meng [6] do not tell us how we can deal with this problem.

We found that one way to deal with this problem is to update the editing degree after each rule application. We first calculate the editing degrees of the graph in $O(mn)$ time and then we can update the editing degree in $O(n)$ for each edge removal. This will give us a total complexity of $O(mn + kn) = O(mn)$, which fits the complexity of the kernel.

For Rules 1-3 it is easy to update the editing degree. For these three rules we create a disjoint clique and then remove it from the graph. The only edits that will affect the editing degree of a vertex that remains in the graph are thus edge deletions. Updating the editing degree works as follows: for each edge that we remove from the graph, we look at the symmetric difference of the two neighbor lists. This can be done in $O(n)$, because the neighbor-lists of the vertices are sorted. Let v be the vertex that stays in the graph and let w be the vertex that will be removed from the graph. If we find a neighbor of v that is not connected to w, then we will lower the editing degree of that edge by one. Figure 4 illustrates this further. A neighbor of v only has to be updated if it is an uncommon neighbor of v and w. If a neighbor q is a common neighbor of v and w, then (q, w) will also always be removed from the graph. Otherwise w will not be in a disjoint clique. The number of uncommon neighbors between v and q will thus not change, so we do not have to update their editing degree.

In order to calculate the editing degree of a critical clique, we choose a vertex in the critical clique and we sum over the editing degrees with its neighbors.0 If an edge is removed, then its editing degree will not be counted anymore for the vertex. This mean that we can ignore the change in editing degree caused by neighbors connected to w but not to v.

Updating the editing degree after applying Rule 4 is harder. Let K be a 4-applicable critical clique and let u be the vertex that is connected to $(|K| + |N(K)|)/2$ vertices in $N(K)$. For Rule 4 we no not immediately remove $K \cup N(K)$ from the graph, so we also have to update the

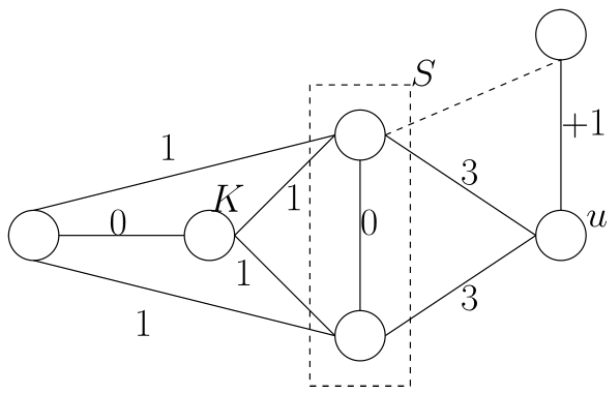

Figure 5: Example of how we will be updating the editing degree for Rule 4. $K \cup N(K)$ just got turned into a clique. S is the set of vertices connected to u. The dashed lines denotes an edge that got removed from the graph and all numbers denote what will happen to the editing degree of that edge. The editing degree of the edges between u and S will be equal to $(|K| + |N(K)| - |S|) + d(u) - |S|$, which is in this case equal to 3.

editing degree of K and $N(K)$ as well. On top of that we now also have to deal with edges that are added to the graph. We will update the editing degrees for K and $N(K)$ as follows: the editing degree of the edges between the vertices in $K \cup N(K)$ all depend on u. Let $S \subseteq N(K)$ be the set for vertices connected to u. We will set the editing degree of the edges that have one endpoint in S and one endpoint not in S to 1. The editing degree of all other edges between the vertices in $K \cup N(K)$ will be 0. For the edges between u and S we know that we need $|K| + |N(K)| - |S|$ edits to connect u with all the neighbors of the vertex. And on top of that we need to remove $d(u) - |S|$ edges that are connected to u, but not to the vertex. The editing degree of the edges between u and S will thus be equal to $(|K| + |N(K)| - |S|) + d(u) - |S|$.

What now remains is the editing degree of the edges between u and $N(u) \setminus S$. The editing degree of these edges will only change if the other endpoint had a common neighbor in $N(K)$. For this we will add an exception in our updating algorithm for updating the editing degrees. Let (v, w) be an edge that is removed from the graph where $v \in N(K)$. If u is a common neighbor of v and w, then we increase the editing degree of (u, w) by one. In all other cases we still only decrement the editing degree for uncommon neighbors only. In Figure 5 we give an example of how all the editing degrees change for Rule 4.

7.2 Calculating the Editing Degree in Parallel

In order to calculate the initial editing degrees we simply loop over all neighbors of an edge and count the number of uncommon neighbors. We can do this because the neighbor lists are sorted. This will take $O(n)$ parallel time and $O(mn)$ work. In order to make it more efficient in our implementation we calculate the editing degree in the critical clique graph. For this you have to count the sizes of the uncommon neighboring critical cliques. So instead of counting 1 for each uncommon neighbor, we count the size of the uncommon neighboring critical clique.

We know that the critical clique graph contains at most $4k_{opt}$ vertices [16]. In the worst case the number of edges in the critical clique graph is $O(k^2)$. This is the case in a complete bipartite graph. All vertices in the complete bipartite graph will be in a different critical clique and all edges will thus also be present in the critical clique graph. Finding the editing degrees can thus be done in $O(k)$ time with $O(k^3)$ work given the critical clique graph. Or if $m < k^2$ then we do it with $O(km)$ work.

The editing degree of an edge is a symmetric property. For an edge (u, v) we have that $p_u(v) = p_v(u)$. We can use this to do only half of the work. We do not have to calculate both editing degrees separately.

The biggest problem when parallelizing the algorithm for updating the editing degrees is that two edits can update the editing degree of the same edge at the same time. This also happens in Figure 4, $p_v(x)$ gets updated by two edits: (w, v) and (v, z). We can circumvent this problem by only considering one edit at a time. Let (u, v) be an edge that gets removed from the graph. Here u will be the vertex that gets removed from the graph and v is the vertex that stays in the graph. For all critical cliques we will check in parallel whether they are connected to v, but not to u. If that is the case then we decrement the editing degree between that critical clique and the critical clique of v. If we can check whether two vertices are connected in $O(1)$ time, then this will result in a total running time of $O(k)$ with $O(k^2)$ work.

In our implementation we instead used atomic decrements to implement this. An atomic decrement makes sure that a value is decremented in one clock cycle. If multiple atomic decrements are called on the same value at the same time, then the decrements will happen after each other. You will always end up with the correct value. Whenever two atomic decrements happen at the same time, your program will slow down a bit. However for us it turned out that this time loss was minimal. In return we get to process multiple edits at the same time. In parallel we loop over the neighbors for each edit in $O(n)$ parallel time and use an atomic decrement if needed. This approach uses $O(kn)$ work. With this approach we also do not need to be able to tell whether two vertices are connected in $O(1)$ time, which makes the implementation easier. We can improve the complexity of this approach by only calculating and updating the editing degrees between the critical cliques and not between every vertex. This will result in a complexity $O(k)$ parallel time and $O(k^2)$ work.

8 Applying the Rules in Parallel

We want to apply as many rules as possible simultaneously. In the sequential version of the kernel we have to construct a critical clique graph or update it each time we apply a rule. Even when there is more than one r-applicable critical cliques in the graph, the sequential algorithm reconstructs the critical clique graph after each rule application. In this section we will look at how we can apply more than one rule at the same time without conflicts.

All rules edit a critical clique and their neighbors. When there is no overlap in the neighbor sets of two (or more) r-applicable critical cliques, then we can safely apply the rules at the same time. More formally we want that $(K_1 \cup N(K_1)) \cap (K_2 \cup N(K_2)) = \emptyset$ for two r-applicable critical cliques K_1 and K_2. If this is the case for all r-applicable critical cliques, then we know that we can apply the rules safely in parallel. In Section 8.1 we will show that this is always the case for Rule 1 and 5. For Rules 2-4 there are one or two exceptions that we can work around.

In Section 8.2 we will discuss how we can apply the edits in parallel. One of the problems there is that it is not straight forward to add edges to the graph in parallel.

8.1 Finding Edits in Parallel

Rules 1-4 all tell us to turn $K \cup N(K)$ into a disjoint clique when a certain condition holds for K. We first have to find all critical cliques for which the condition holds. Then we have to find all the edits that we have to do in order to turn $K \cup N(K)$ into a disjoint clique.

For Rule 1 we have to find all the critical cliques that are bigger than k. We can check in $O(1)$ parallel time and $O(k)$ work whether this is the case for all critical cliques. Let K be a critical clique with $|K| > k$. In order to find the edits we will mark all vertices in K and $N(K)$ with the unique identifier of K. We can then go over all edges in the graph in parallel and set a flag when two endpoints of an edge have a different mark. Now all edges that should be removed from the graph will have a flag. This takes $O(1)$ parallel time and $O(m)$ work. Edges that should be added to the graph can easily be deduced from the fact that we are turning $K \cup N(K)$ into a disjoint clique, so we will just emit the vertices in K to signal this to the user.

Algorithm 4 generalizes this approach for Rules 1-4. We will use the output of Algorithm 4 in Section 8.2 where will we discuss how we can apply these edits.

```
Algorithm 4: Find Edits in Parallel
Input: (C, G, r): G is the graph and C contains a critical clique id for each vertex. r
       denotes the rule for which we want to find the edits.
Output: Flags for all edges that should be removed from the graph and flags for all
        critical cliques that should be turned into disjoint cliques with their
        neighbors.
do in parallel
    CCFlags[i] = CheckRule(i, r); // Check whether a rule applies for critical clique i.
end
do in parallel
    (v, w) = E[i]; // Get the edge.
    if CCFlags[C[v]] then
        VertexMarks[w] = C[v]; // Mark w with the critical clique id of v.
    end
    if CCFlags[C[w]] then
        VertexMarks[v] = C[w]; // Mark v with the critical clique id of w.
    end
end
do in parallel
    (v, w) = E[i]; // Get the edge.
    EdgeFlag[i] = VertexMarks[v] != VertexMarks[w];
end
return (EdgeFlags, CCFlags);
```

8.1.1 Rule 1

Suppose that a critical clique containing vertex v and w is adjacent to two critical cliques: K_1 and K_2. Algorithm 4 will not work when K_1 and K_2 are both r-applicable critical cliques. v and w will both be marked with K_1 and K_2 by two different processors, so we do not know for sure which mark they will get. If v ends up with a mark for K_1 and w ends up with a mark for K_2 then the edge between v and w will be flagged to be removed. In this case our algorithm made a mistake. (v, w) should never be removed from the graph, because it is an edge within a critical clique. In order to avoid this situation we have to show that a vertex v can only be adjacent to a single r-applicable critical clique. Theorem 2 tells us that we can safely apply Rule 1 in parallel with our algorithm.

Theorem 2. *A vertex v can only be connected to at most one critical clique with a size larger than k.*

Proof. Consider a critical clique K_1 with $|K_1| > k$ and a vertex $v \in N(K_1)$. Assume that v is adjacent to another critical clique K_2 with $|K_2| > k$. We have to turn $K_1 \cup N(K_1)$ into a disjoint clique according to Rule 1.

If $K_2 \not\subset N(K_1)$ then we have to remove all edges between v and K_2 from the graph. This will result in more than k edits, because $|K_2| > k$. We will thus never find a solution when $K_2 \not\subset N(K_1)$.

Now consider the other case: $K_2 \subseteq N(K_1)$. By definition of a critical clique, we know that K_1 has at least one different neighboring critical clique compared to K_2. There either is a critical clique that is connected to K_2 and not to K_1. In which case we have to remove

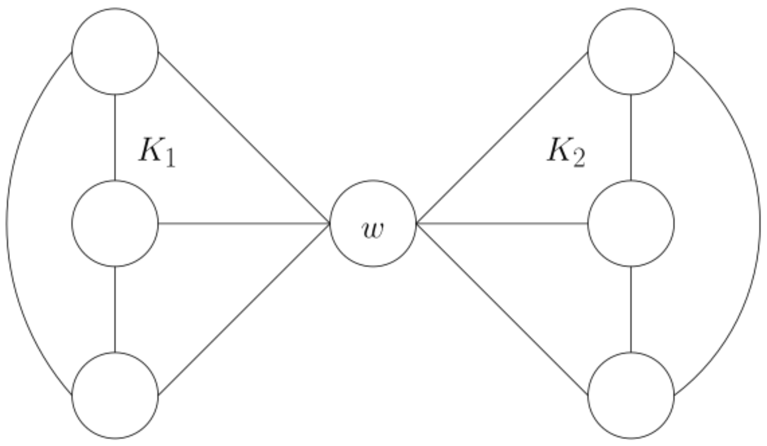

Figure 6: Situation in Theorem 4. K_1 and K_2 are not connected and both have w as a neighbor. $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$ holds for both critical cliques. K_1 and K_2 can not have any other neighbors without violating Theorem 3.

those edges from the graph, resulting in more than k edits. Or there is a critical clique that is connected to K_1, but not to K_2. In which case we have to add more than k edges in order to connect it to K_2. We will thus never find a solution when $K_2 \subseteq N(K_1)$.

Because we need more than k edits in both cases, we can conclude that Theorem 2 holds. □

8.1.2 Rule 2

For Rules 2-4 we need to show that a vertex v can only be adjacent to exactly one critical clique with $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$. However, it turns out that a vertex can be adjacent to more than one critical clique for which this condition holds in two cases:

1. If a vertex is the only neighbor of two critical clique that have the same size, then $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$ holds for both critical cliques (see Figure 6).
2. If a vertex is adjacent to two critical cliques that are also adjacent to each other. It is not always the case here, but it is possible. See Figure 2 for an example. Critical clique 1 and critical clique 2 are connected and they both satisfy $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$.

We will first show that apart from these two exceptions a vertex v can not be adjacent to more than one critical clique with $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$. We will then explain how we can still apply the rules for these two exceptions in parallel.

Cheng and Meng already showed that Theorem 3 holds in their proof of Corollary 2.5 in [6]. We will briefly repeat the proof here, because Corollary 2.5 in [6] also assumes that $K \geq N(K)$, which is not needed for us. We need Theorem 3 to prove Theorem 4. Theorem 3 tells us that the editing degree of a vertex with respect to a critical clique can not be too large.

Theorem 3. *If* $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$*, then for all* $v \in N(K) : p_K(v) \leq |K|$.

Proof. Cheng and Meng [6] already showed this in their proof for Corollary 2.5. We will repeat it here briefly:

$$\begin{aligned}
p_K(v) &= \sum_{u\in N(K)} p_K(u) && - \sum_{u\in N(K)\setminus\{v\}} p_K(u) && \\
&\le \sum_{u\in N(K)} p_K(u) && - (|N(K)|-1) && (\text{Lemma 1}: p_K(u)\ge 1) \\
&\le (|K|+|N(K)|-1) && - (|N(K)|-1) && (|K|+|N(K)| > \sum_{v\in N(K)} p_K(v)) \\
&= |K|
\end{aligned}$$

□

Theorem 4. *A vertex w can be adjacent to at most one critical clique with $|K| + |N(K)| > \sum_{v\in N(K)} p_K(v)$, unless w is the only neighbor of two different critical cliques or the neighboring critical cliques are neighbors of each other.*

Proof. Consider a vertex w and a critical clique K_1 with $w \in N(K_1)$ and $|K_1| + |N(K_1)| > \sum_{v\in N(K_1)} p_{K_1}(v)$. Consider another critical clique K_2 also with $w \in N(K_2)$ and $|K_2| + |N(K_2)| > \sum_{v\in N(K_2)} p_{K_2}(v)$.

Assume that K_1 and K_2 are not adjacent. By Theorem 3, we know that $p_{K_1}(w) \le |K_1|$. We know that $p_{K_1}(w)$ is at least $|K_2|$, because K_1 and K_2 are not adjacent. Therefore we can conclude that $|K_2| \le |K_1|$. Via the same reasoning we know that $p_{K_2}(w) \ge |K_1|$, so $|K_1| \le |K_2|$. When we combine these two inequalities, we get that $|K_1| = |K_2|$.

We will now show that w is the only neighbor of K_1 and K_2. Suppose there is another neighbor v. Without loss of generality we will assume that it is connected to K_1. Consider the following two cases:

- v is connected to w: $p_{K_2}(w)$ will become bigger than $|K_2|$ in this case, which is not allowed according to Theorem 3.
- v is *not* connected to w: $p_{K_1}(w)$ will become bigger than $|K_1|$ in this case. This is also not allowed according to Theorem 3.

From this we can conclude that v can not exist, and thus that w is the only neighbor of K_1 and K_2. □

Let K_1 be a critical clique that is adjacent to all neighbors of another critical clique: K_2. This situation implies that $|K| \ge |N(K)|$ can only hold for one of the two critical cliques:

- If $|K_2| > |K_1|$, then we have that $|K_1| < |N(K_1)|$ (because $K_2 \subset N(K_1)$).
- If $|K_2| < |K_1|$ we have that $|K_2| < |N(K_2)|$ (because $K_1 \subseteq N(K_2)$).
- If $|K_1| = |K_2|$, then we also have that $|K_1| < |N(K_1)|$, because K_1 needs to have at least one neighbor that is different from K_2.

This means that a vertex can only be adjacent to one critical clique with $|K| + |N(K)| > \sum_{v\in N(K)} p_K(v)$ and $K \ge N(K)$ according to Theorem 5. Theorem 4 and 5 thus tell us that we can safely apply Rule 2 in parallel unless a vertex is the only neighbor of two critical cliques of the same size. Figure 7 shows the situation in Theorem 5. We will show that x_1 or x_2 must be zero which means that either K_1 or K_2 is connected to all neighbors of the other.

Theorem 5. *Consider two critical cliques K_1 and K_2 where $|K| + |N(K)| > \sum_{v\in N(K)} p_K(v)$ holds for both critical cliques. If $K_2 \subset N(K_1)$, then one of the two critical cliques is connected to all neighbors of the other.*

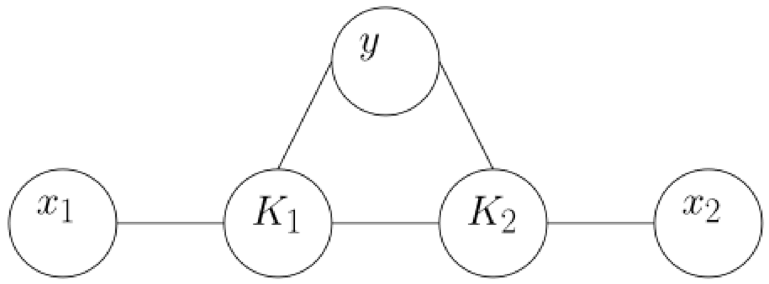

Figure 7: Situation in Theorem 5. K_1 and K_2 are neighbors and $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$ holds for both critical cliques. x_1 is the number of vertices connected to K_1, but not to K_2. x_2 is the number of vertices connected to K_2, but not to K_1. y is the number of vertices connected to both K_1 and K_2.

Proof. Let $x_1 = |N(K_1) \setminus N(K_2)|$, $x_2 = |N(K_2) \setminus N(K_1)|$ and $y = |N(K_1) \cap N(K_2)|$. We can now define the sizes of the critical cliques plus their neighbors as follows for K_1 and K_2:

$$|K_1| + |N(K_1)| = |K_1| + |K_2| + x_1 + y \tag{1}$$
$$|K_2| + |N(K_2)| = |K_2| + |K_1| + x_2 + y \tag{2}$$

For the total editing degree of both critical cliques we can now also deduce a minimal value. For every vertex in $N(K_1) \setminus N(K_2)$ the editing degree of K_1 increases with $2|K_2|$. Additionally for every vertex in $N(K_2) \setminus N(K_1)$ the editing degree of K_1 increases with at least 1. Finally the editing degree increases by at least one for every vertex in $N(K_1) \cap N(K_2)$ (Lemma 1). This yields the following lower bounds for the editing degrees for K_1 and K_2:

$$\sum_{v \in N(K_1)} p_{K_1}(v) \geq y + 2x_1 \cdot |K_2| + x_2 \tag{3}$$
$$\sum_{v \in N(K_2)} p_{K_2}(v) \geq y + 2x_2 \cdot |K_1| + x_1 \tag{4}$$

Since the editing degree should be smaller than $|K| + |N(K)|$ for both critical cliques, we get:

$$y + 2x_1 \cdot |K_2| + x_2 < |K_1| + |K_2| + x_1 + y \tag{5}$$
$$y + 2x_2 \cdot |K_1| + x_1 < |K_2| + |K_1| + x_2 + y \tag{6}$$

Adding these inequalities together gives us the following inequality:

$$2x_1 \cdot |K_2| + 2x_2 \cdot |K_1| + x_2 + x_1 + y < 2|K_2| + 2|K_1| + x_2 + x_1 + y \tag{7}$$
$$2x_1 \cdot |K_2| + 2x_2 \cdot |K_1| < 2|K_2| + 2|K_1| \tag{8}$$

This implies that either $x_2 = 0$ or $x_1 = 0$ because one of the two must be at least 1 in order for K_1 and K_2 to be different critical cliques.

Without loss of generality we will assume that $x_1 \geq 1$ and $x_2 = 0$. It now directly follows from equations 1 and 2 that $|K_1| > |K_2|$, so $|K_2| < |N(K_2)|$. □

The only exception for Rule 2 is thus when a vertex w is the only neighbor of two critical cliques for which Rule 2 applies. Let K_1 and K_2 be such critical cliques. We can work around this situation by making sure that we either remove all edges between K_1 and w or all edges between K_2 and w. Algorithm 4 already does this for us. w will be marked with either K_1 or with K_2. Because w is the only neighbor of K_1 and K_2 there will not be any edges that are faultily removed.

8.1.3 Rule 3 and 4

Theorem 4 tells us that we can safely apply Rules 3 and 4 in parallel under two conditions:

- The input should not contain a (sub)graph that consists of two identical critical cliques (K_1 and K_2) connected by a single vertex w.
- We can not have two neighboring critical cliques with $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$.

We do not have to worry about the first condition, because both Rule 3 and 4 require that $|K| < |N(K)|$ which is not the case in that situation.

In order to work around the second condition we can use Theorem 5. When two neighboring critical cliques both satisfy $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$, then one of then is connected to all neighbors of the other. This means that when there are multiple (two or more) neighboring critical cliques with $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$ then there is exactly one of them that is adjacent to all neighbors of the others. Additionally we know that this will be the largest critical clique.

As an extra step for Algorithm 4 we thus need to make sure that when a critical clique is 3- or 4-applicable and it has a neighboring critical clique that is also 3- or 4-applicable, we only set the flags if it is the largest critical clique. For this we will add an extra step after Line 3 in Algorithm 4. We will look at all edges in the critical clique graph in parallel. If for a critical clique edge (K_1, K_2) both K_1 and K_2 are 3- or 4-applicable then we remove the flag of the smallest critical clique. This can be done in $O(1)$ parallel time with $O(m)$ work.

For Rule 3 it is always the best option to pick the largest critical clique. It will always result in more vertices that we can remove from the graph, because it is adjacent to all neighbors of the other neighboring 3- or 4-applicable critical cliques. For Rule 4 however we do not remove the vertices from the graph. It might therefore be better to apply Rule 3 first on a neighboring 3-applicable critical clique, so that we can remove the vertices earlier. However, Theorem 6 tells us that the largest critical clique is never 4-applicable. This means that by picking the largest critical clique we will always remove the maximum number of vertices from the graph.

Theorem 6. *Let K_1 and K_2 be two adjacent critical cliques that both satisfy $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$. If $|K_1| > |K_2|$ then K_1 can not be 4-applicable.*

Proof. By Theorem 5 we already know that K_1 is adjacent to all neighbors of K_2. Let x denote the number of vertices connected to K_1, but not to K_2 and let y denote the number of vertices connected to both K_1 and K_2. We know that the editing degree of K_2 is at least y, because all common neighbors of K_1 and K_2 must be connected to at least one other critical clique. Otherwise these vertices would be in the same critical clique as K_2. Additionally we need to remove $x|K_1|$ edges for the editing degree of K_2. We thus have that:

$$|K_2| + |K_1| + y > p_{K_2} > x|K_1| + y \tag{9}$$

Because $|K_1| > |K_2|$ this implies that $x = 1$. For the editing degree of K_1 we need to add $2x(K_2 + y)$ edges. If we use the fact that $x = 1$ we have that:

$$|K_1| + |K_2| + x + y > 2x(|K_2| + y) \tag{10}$$
$$|K_1| + |K_2| + 1 + y > 2|K_2| + 2y \tag{11}$$
$$|K_1| + 1 > |K_2| + y \tag{12}$$

Because $|K_2| \geq 1$ we now know that $|K_1| > y$. In combination with the fact that $x = 1$ this means that a vertex u can be connected to at most $|K_1|$ neighbors of K_1. u can thus never be connected to more than $(|K_1| + |N(K_1)|)/2$ neighbors of K_1, so K_1 can never be 4-applicable. □

For Rules 3 and 4 we also have to check whether there is a vertex u that is connected to more than $(|K|+|N(K)|)/2$ neighbors of K. We can find these type of vertices in the following way: first we sort the neighbors of all vertices based on their critical clique mark. If we use the parallel radix sort algorithm we discussed in Section 6.2.1, then we can directly see how many times a mark occurs from the min- and max-values. From there we can easily flag a critical clique if this count exceeds $(|K| + |N(K)|)/2$. This last step can be done in $O(1)$ time with $O(m)$ work. Sorting will cost us $O(\log n)$ time and $O(m)$ work.

An alternative for finding the u vertex is to go over the neighbors of each 3- or 4-applicable critical clique. We will have an array of size k for each critical clique that we will use to count the occurrences of the neighbors. We will look at all critical cliques that are not marked with their own identifier. Lets say that a critical clique c_1 is marked with critical clique c_2. We will go over the neighbors of c_1 and add one to the correct index in the array of c_2. For this we will use an atomic increment. If at any point the count of a vertex exceeds $(|K| + |N(K)|)/2$ for c_2, then we found our u vertex. It turns out that using the atomic increments is a faster method than using the radix sort in practice. However, it has a runtime of $O(n)$ with $O(kn)$ work, which is worse than the radix sort approach.

8.1.4 Rule 5

Rule 5 also tells us to also look at critical cliques that have $|K| < |N(K)|$ and $|K| + |N(K)| > \sum_{v\in N(K)} p_K(v)$. Using the fact that we will only use Rule 5 when the other rules can not be applied, we know that $N_2(K)$ only contains a single vertex. Cheng and Meng already showed this in [6]. We also know that $K \cup N(K)$ forms a complete sub-graph for a 5-applicable critical clique, because otherwise we could have applied Rule 3 or 4. It is then not hard to see that there will never be two adjacent 5-applicable critical cliques. We can thus also apply Rule 5 safely in parallel.

In order to find the edits for Rule 5 we have to select $|K|$ vertices from $N(K)$. We choose to always pick the first $|K|$ vertices from $N(K)$. In parallel we do this by flagging the vertices in $N(K)$ if their location in the adjacency list is smaller than $|K|$. We already know that K and $N(K)$ form a complete sub graph. So we only have to remove the edges between the selected vertices and $N_2(K)$. We will therefore flag all edges that are connected to a selected vertex and a vertex in $N_2(K)$. This all can easily be done in $O(1)$ parallel time with $O(m)$ work.

8.2 Editing the Graph in Parallel

For Rules 1-3 we remove K and $N(K)$ from the graph after we turned it into a disjoint clique. In our internal representation of the graph we thus do not have to add any edges. Just emitting that an edge is added is enough. For Rules 1-3 we can therefore simply remove all edges and vertices from memory that are connected to a r-applicable critical clique.

For Rule 4 however we can not immediately remove K and $N(K)$ from the graph. We thus have to actually add edges to our graph. This imposes some problems. How do we for example know where we place the new edge in memory? We have to avoid conflicts between two processors that want to add an edge to the same vertex. A processor that adds an edge to the end of an adjacency list has to be sure that it is the only one that is placing an edge there. Otherwise it can possibly override the added edge of another processor.

We would also have to allocate memory for edges that we will maybe add to the graph, but how many space should we allocate so that all edits will fit? The best we can do is allocate twice the number of edges that are in the input graph. If we add more than $d(v)$ neighbors to a vertex v, then it would be better to just remove all the edges connected to v. We will therefore never find a solution where we add more than $d(v)$ neighbors to v. From a practical point of view it might not be ideal to allocate twice as much memory as we need. Most of the allocated space will never be used, because we will add at most k edges to the graph and we will be allocating $O(m)$ more memory.

We came up with a way of adding edges to the graph without using additional memory. In fact after applying a rule we will always use less memory than we did before the rule application. We will use the concept of twin vertices in order to do this. Twin vertices are vertices with the same neighbors. In the literature there are two types of twin vertices. True twins and false twins. True twins are connected to each other and false twins are not connected to each other. Twins are also related to modular decomposition (see Section 6). True twins are vertices that you can find in a Type I module. False twins can be found in Type II and Type III modules. We are only interested in true twins. True twins have the same set of neighbors, so we can use the adjacency list of only one of the two vertices to store the connections of both vertices. For all vertices in the graph we will have a pointer to an adjacency list. By letting twin vertices point to the same neighbor list we will use less memory.

When we apply a rule we have to turn $K \cup N(K)$ into a disjoint clique. This means that all vertices in $K \cup N(K)$ will get the same set of neighbors, they all become true twins of each other. We also already know what neighbor set they will all get. Namely the same set as one of the vertices in K. We can thus adjust the pointers of the neighbor-lists of all vertices in $N(K)$ to an adjacency list of a vertex in K. After we do this $K \cup N(K)$ will be a disjoint clique.

For Rule 4 however this is a bit more complicated. We can not remove the connections between $N(K)$ and the vertex u. With Theorem 7 we show that there exists a vertex $w \in N(K)$ which connected to all other vertices in $N(K)$ and also connected to u. This means that we can adjust the pointers of all vertices in $N(K)$ that are connected to u to the adjacency list of w. The pointers of all other vertices in $N(K)$ will still be changed to the adjacency list of a vertex in K.

Theorem 7. *If $|K| < |N(K)|$ and $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$ and there exists a vertex $u \in N_2(K)$ with $|N(u) \cap N(K)| > (|K| + |N(K)|)/2$, then there is at least one vertex in $v \in N(K)$ with $K \cup N(K) \cup \{u\} \subseteq N(v)$.*

Proof. Let $S \subseteq N(K)$ be the set of vertices that is connected to u. Note that $|S| > (|K| + |N(K)|)/2$. If there does not exist a vertex in S that is connected to all vertices in $N(K)$, then the editing degree will be at least 2 for every vertex in S. We need one edit to remove the connection with u and we need at least one edge addition to connect the vertex with all vertices in $N(K)$. This will add up to at least $2\frac{|K|+|N(K)|}{2} = |K| + |N(K)|$ edits. This contradicts the fact that $|K| + |N(K)| > \sum_{v \in N(K)} p_K(v)$. $\square$

All vertices that now share an adjacency list are vertices within the same critical clique. Vertices that are connected to u form a critical clique and the rest of the vertices in $K \cup N(K)$ also form a critical clique. From [16] and [6] we already know that edges within a critical clique will never be removed from the graph. Vertices that share an adjacency list will thus always stay vertices with the same neighbors lists. Which means that we can safely do this.

The old adjacency lists of the vertices that now point another adjacency list can be removed from memory. A 4-applicable critical clique always has at least two neighbors, because Rule 4 requires that $|K| < |N(K)|$. This means that we we will always adjust at least one pointer. Each time after we apply Rule 4 we will thus use less memory.

9 Complexity of the kernel

Our algorithm will not just apply one rule at the time. It will apply as many rules as possible in parallel. This means that we do not always have to update the critical cliques n times. In practice it might even happen that we only have to update the critical cliques a constant

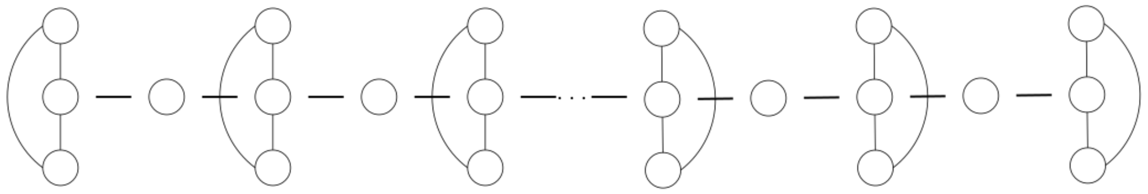

Figure 8: Example of a graph that only allows us to apply one rule at a time. Rule 2 can be applied to only the right (or left) most critical clique. If we remove it from we graph we can again apply Rule 2 only to the right (or left) most critical clique.

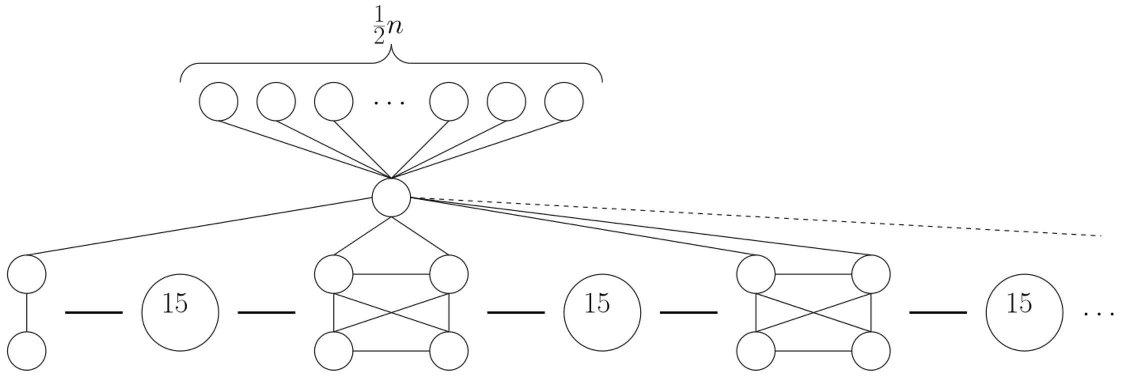

Figure 9: Example of a graph that only allows us to apply one rule at a time. Rule 2 can be applied to only the left most critical clique of size 15. If we remove it from we graph we can again apply Rule 2 only to the left most critical clique. Each time we remove the critical clique and his neighbor we have to update $\frac{1}{2}n$ editing degrees.

number of times. In this case we can thus end up with a total amount of work equal to $O((n+m)\log\log{(n+m)})$ when we use Hagerup's [18] algorithm to find the critical cliques.

There are however still situations where we can only apply one rule each iteration. Consider the graph in Figure 8. We create a chain-like graph of alternating critical cliques of size 3 and 1. In this graph we can only apply Rule 2 on the left most and the right most critical clique. We can make this chain as long as we want. Each time we can thus apply Rule 2 twice and remove 8 vertices from the graph. We will thus iterate $\frac{n}{8}$ times. This shows that our algorithm does not always iterate a constant number of times. However the number edges we have to look at in order to update the critical cliques or editing degrees are not $O(m)$ in this case. Removing an edge can only change the critical clique of an adjacent vertex and its neighbors. In the example of Figure 8 we have to look at at most 8 vertices in order to update the critical cliques. This could still mean that we can get away with a constant amount of work for each rule application.

Figure 9 shows a graph where we also apply the rules $O(n)$ times. Each iteration we can only remove one of the critical cliques of size 15. Unlike the graph in Figure 8 however we also need $O(n)$ work to update the editing degrees of the vertices in the top part of the graph. For this example we could use the kernel of Gramm et al. [15] which we briefly discussed in Section 4 to get rid of the vertices at the top first.

Our parallel algorithm uses $O(n\log n)$ parallel time and $O(mn)$ work in total. This is mainly due to the fact that there are still situations where the rules are applied $O(n)$ times. The most expensive step for applying the rules is finding the u vertex for Rule 3 and 4 which takes $O(\log n)$ parallel time. The total parallel time could definitely be lower if we figure out a way to always apply the rules a constant number of times. Unfortunately we were unable to adjust the kernel to make sure this always happens. If we had figured this out then it would also have been beneficial to look at faster methods for calculating and updating the editing degrees. Right now the total parallel run time for this is $O(k)$.

10 Results

The bio-informatics department of the Friedrich-Schiller University Jena gathered evaluation data from different works that focussed on the cluster editing problem. The data sets can be found on their website: `https://bio.informatik.uni-jena.de/data/#cluster_editing_data`. We also used these data sets to evaluate our implementation. In total these data sets have multiple gigabytes of input graphs. It contains graphs with up to 5000 vertices and 11 million edges. Most of the data is randomly generated.

We decided to only look at graphs with more than 500 vertices. All graphs with less than 500 vertices all take roughly the same amount of time, so they are not very interesting to look at. Most of the graphs contain less than 4 million edges. Only 77 input graphs contain more than 4 million edges. We decided to look at these 77 graphs separately from the rest of the data.

The machine we gathered our results on has the following specifications: CPU: Intel (R) Core (TM) i7-6700HQ @ 2.60 GHz, GPU: GeForce GTX 960 M. We used a lightweight Linux distribution (Manjaro XFCE) as our operating system.

In Figure 10 we show the running time of each part of our algorithm for graphs with less than 4 million edges. You can clearly see two linear trends when we are sorting the neighbors of the vertices. If the neighbors in a graph are already close to being sorted, then the neighbors sort takes a significantly less amount of time. This seems also to be the case for the critical clique graph construction. When the vertices in the graph are already close to being sorted, then the critical clique graph construction finishes faster.

Figure 10 also shows the time our algorithm takes to apply the rules. Again there seem to be two different trends. The time it takes to update of the critical clique graph and the editing degrees are included in the time it takes to apply the rules. If a rule tells us to edit the graph, then the rule applications take significantly more time.

The bottleneck for our algorithm currently is the calculation of the editing degree. In Figure 10 you can clearly see spikes in the runtime for certain graphs. We suspect that the structure of these graphs causes these spikes in runtime. These spikes mostly also matches the spikes in the total runtime, which can be seen in Figure 11.

The structure of the critical clique graph has an impact on the calculations for the editing degree. If the critical clique graph is very similar to the original graph then it will take more time to calculate the editing degrees. In order to see whether this would explain the spikes in runtime, we plotted the edges in the critical clique graph against the runtime in Figure 12 You can see that this does not fully explain the spikes in runtime. There are still spikes visible and our algorithm is even relatively fast for the biggest critical clique graph.

The spikes could also have something to do with how a GPU works. We already briefly spoke about this in Section 6.2. A GPU divides the work into work-groups. Only a limited number of work-groups can be active at the same time and within a work-group the same calculations are always performed at the same time. This means that if you loop over an array in your GPU code, all processors in the work-group have to perform the same amount of iterations. The entire work-group will be done after the last thread is done with all its iterations. A new work-group can only start after that happens. In our case this means that if we are unlucky and all vertices with a high degree end up in a different work-group, we will use considerably more time compared to when all high degree vertices are processed in the same work group. One possible way to help the GPU with this is to sort the critical clique graph based on the degrees before calculating the editing degree. Unfortunately this was not very easy to do with our current implementation.

In Figure 13 we show the total runtime of all input graphs with more than 4 million edges. Here we can also see spikes in the runtime as well. You can clearly see that the calculation of the editing degree takes the most time. The construction of the critical clique graph takes

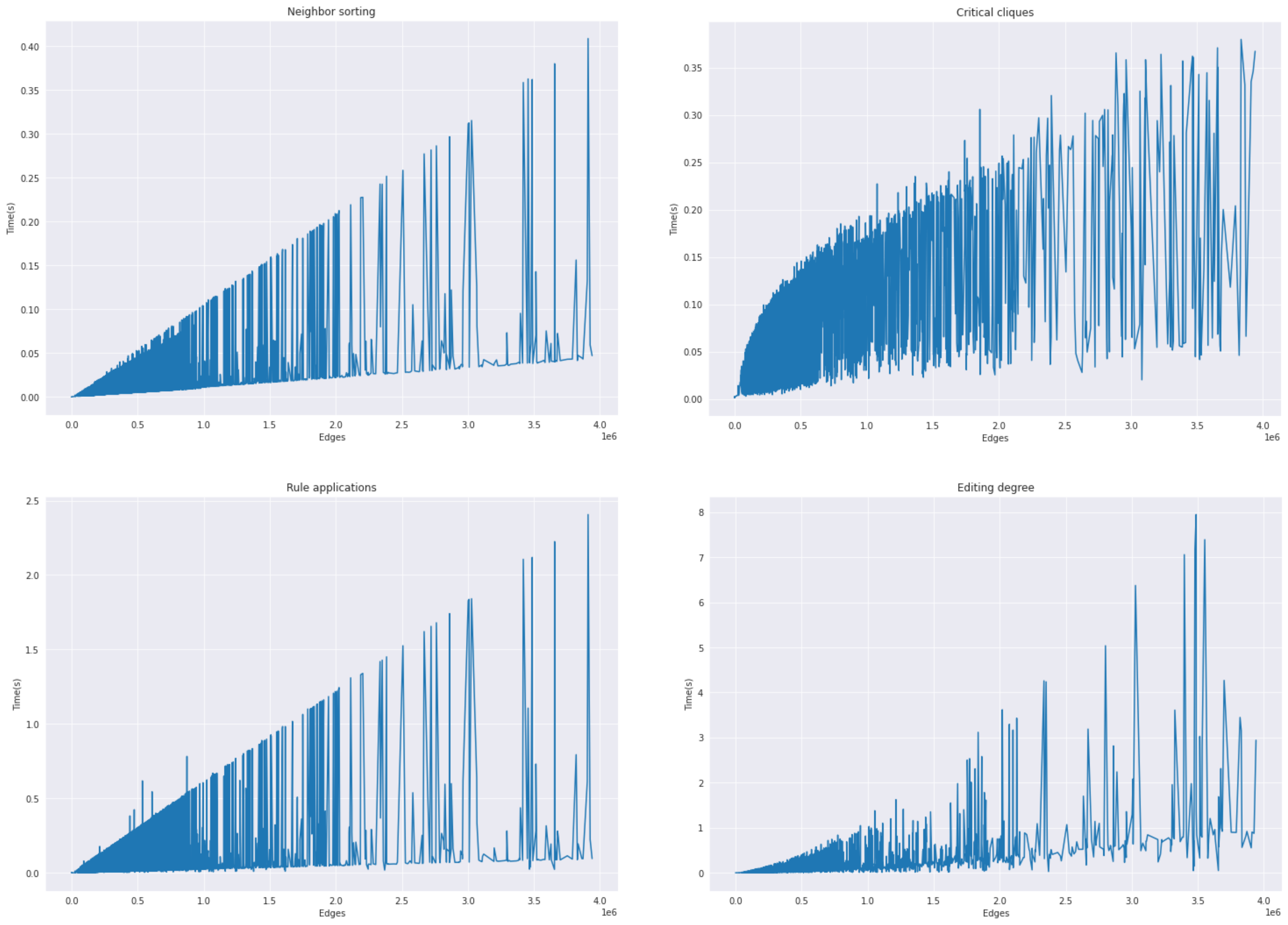

Figure 10: The runtime of different parts of our kernel. Upper left figure shows the runtime of sorting all neighbors in the graph. Upper right figure shows the time it takes to find the critical cliques. The bottom left figure shows the time we spend applying the rules. The time for updating the critical cliques and the editing degree is also included. Finally we show how much time it takes to calculate the editing degree in the bottom right figure.

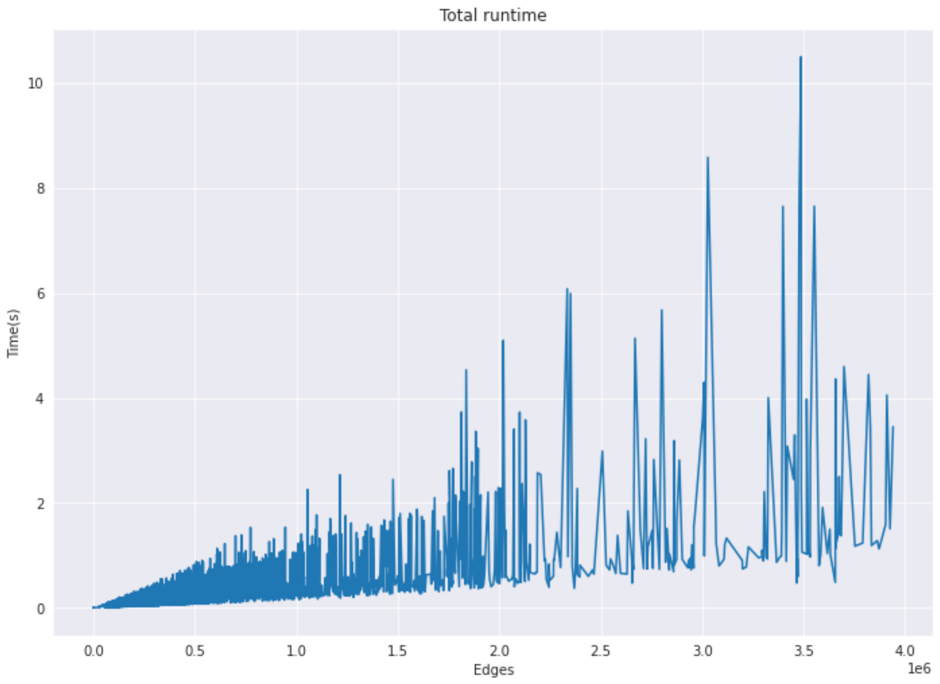

Figure 11: Total runtime of the parallel kernel vs the number of edges in the input graph.

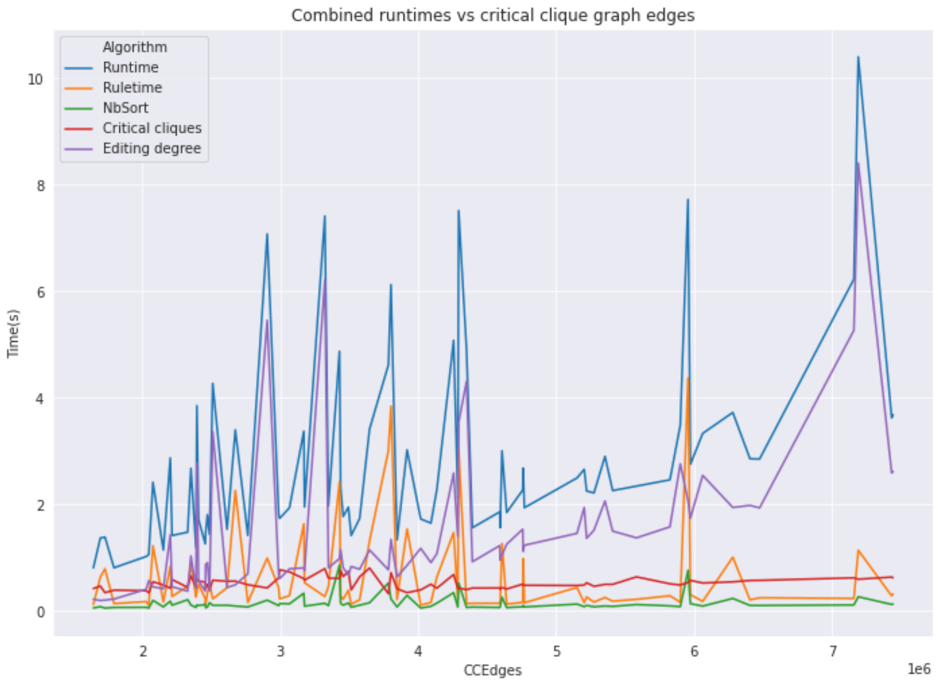

Figure 12: All the runtimes for the different parts of our algorithm for bigger graphs versus the amount of edges in the critical clique graph.

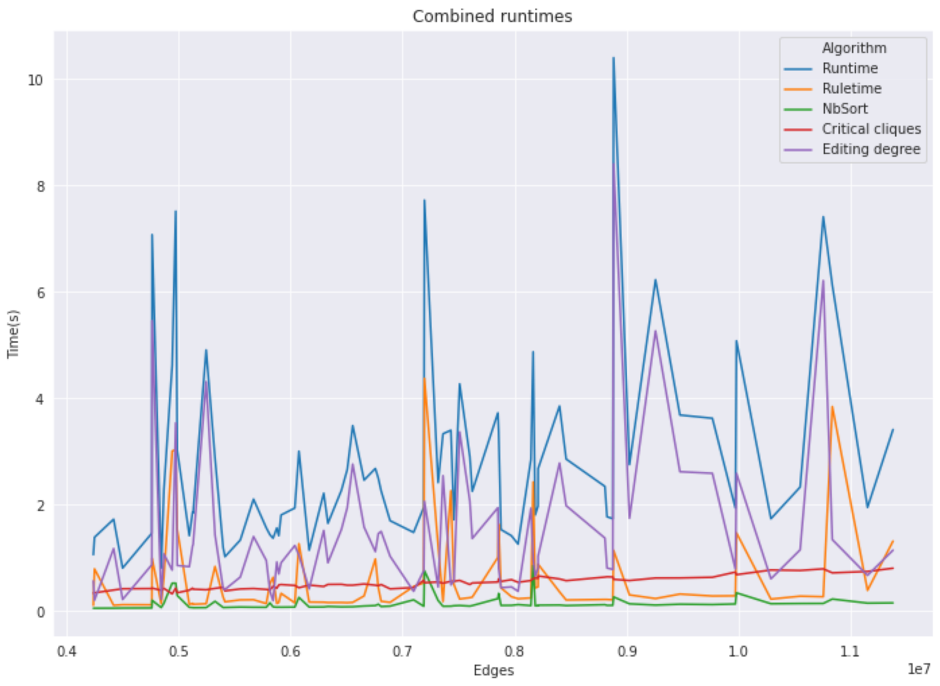

Figure 13: All the runtimes for the different parts of our algorithm for bigger graphs.

relatively less time. Figure 13 also clearly shows that whenever one part of our algorithm uses more time, other parts take more time as well. This makes us believe that it is indeed the structure of that particular graph that makes our algorithm run slower.

In Figure 14 we show the time our algorithm takes for each rule. You can clearly see that Rule 3 and 4 take the most amount of time. Rule 1 and Rule 2 use roughly the same amount of time. This is as expected, because the complexity of their implementations are very similar. Rule 5 takes the least amount of time. This is because Rule 5 is only applied when the others can not be applied anymore. Figure 15 shows how the rules compare to each other for the bigger graphs. Here you can clearly see that Rule 3 and 4 take the most amount of time.

In Figure 16 we show the distribution of how many times the rules are applied. Here we only considered the input graphs that gave us at least one edit. We count a rule application if we check whether a rule applies, not just when we actually find edits. The maximum amount of rule applications is just thirteen, but in most cases the rules are applied less than eight times. Eight rule applications correspond to checking each rule twice. This confirms our suspicion that the rules are applied only a constant number of times in practice. This means that in practice our algorithm should use $O(n)$ time instead of $O(n \log n)$ time.

11 Conclusions

We created a parallel version of Cheng and Meng's [6] 2k kernel for the Cluster Editing problem. We showed that all rules can be applied in parallel without conflicts and that by doing so we always remove the maximum amount of vertices from the graph after a rule application. We also improved parts of the original kernel by coming up with a way to update the critical cliques and the editing degree instead of recalculating them after each rule application. We also implemented an efficient way to find critical cliques in a graph, which could also be used to find type I modules for modular decomposition. For our implementation of our kernel we showed

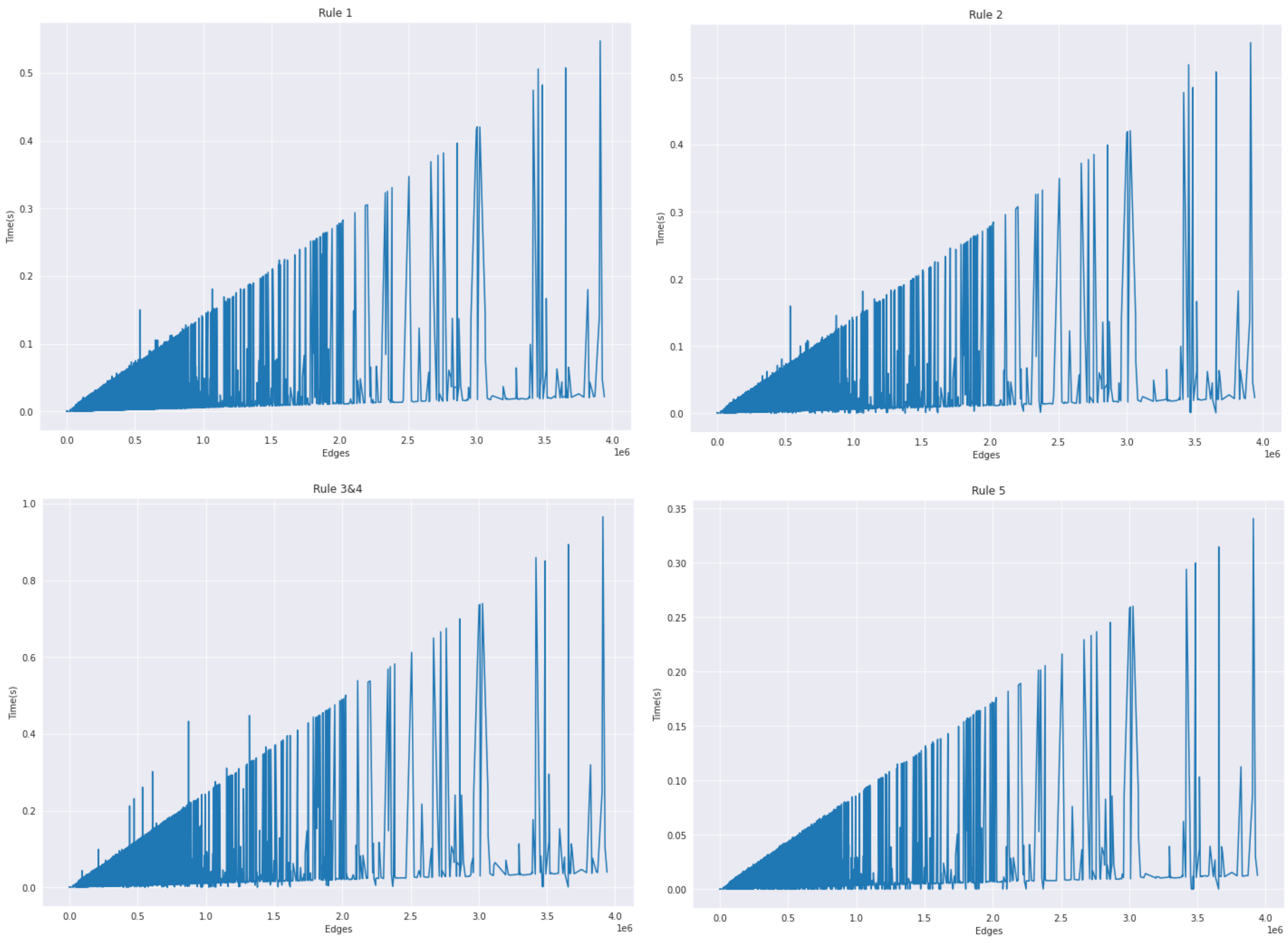

Figure 14: The amount of time we spend per rule. Top left figure shows Rule 1. Top right figure shows Rule 2. Bottom left figure shows Rule 3 and 4. Bottom right figure shows Rule 1.

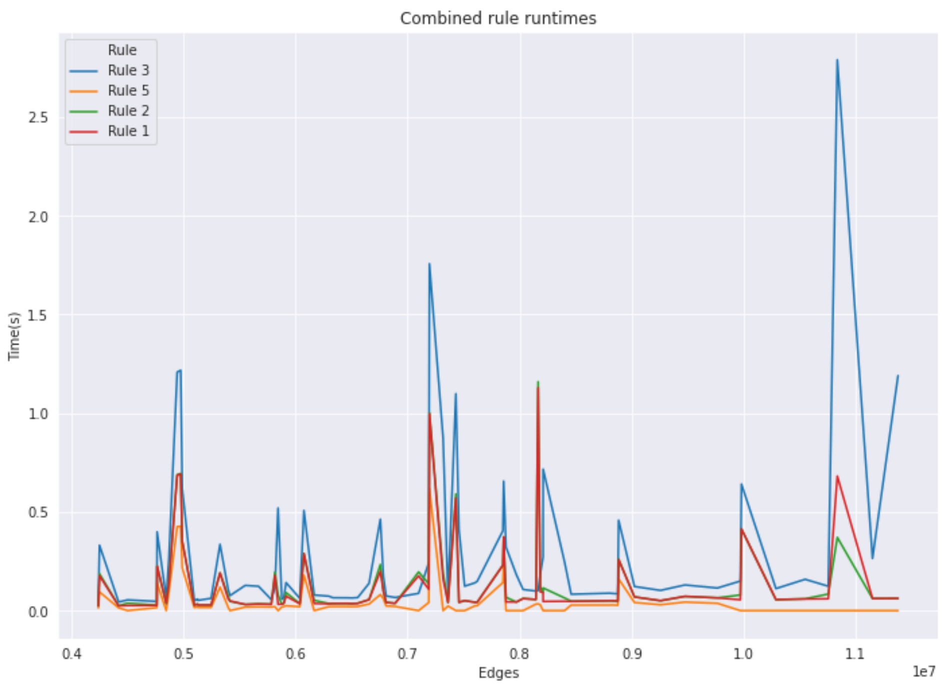

Figure 15: Times spend on applying the rules for bigger graphs.

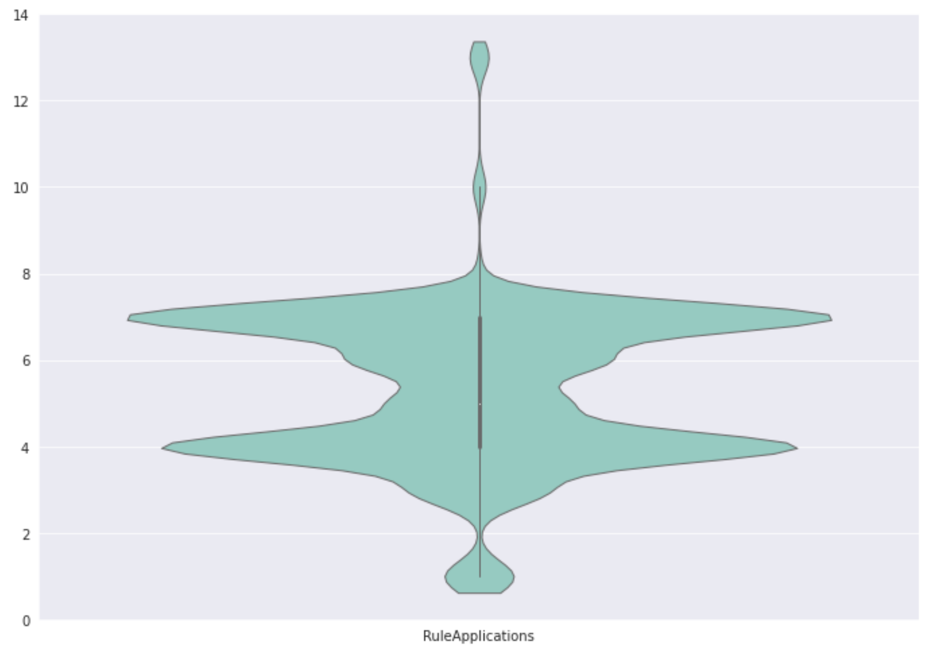

Figure 16: Distribution of the amount of rule applications for graphs where we found at least one edit.

that the rules are applied a constant number of times in practice and that the total runtime scales linearly with the number of edges in the graph. Our implementation also guarantees that after each rule application we use the same amount of memory or less, even when we add edges to the graph.

There are still ways our algorithm could be improved. From a theoretical point of view you could still look further at finding ways to ensure that the rules are only applied a constant number of times. From a practical point of view our implementation could still be improved as well. Mainly the calculations of the editing degree could be done faster. For this you could look at more efficient triangle counting algorithms as we explained in Section 7. Although we tried to optimize our implementation as much as possible, we do believe that there are still a number of ways to improve the speed.

References

[1] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. "Correlation clustering". In: *Machine Learning* 56.1-3 (2004), pp. 89–113. DOI: 10.1023/B:MACH.0000033116.57574.95. URL: https://doi.org/10.1023/B:MACH.0000033116.57574.95.

[2] Timo Bingmann, Andreas Eberle, and Peter Sanders. "Engineering Parallel String Sorting". In: *Algorithmica* 77.1 (2017), pp. 235–286. DOI: 10.1007/s00453-015-0071-1. URL: https://doi.org/10.1007/s00453-015-0071-1.

[3] Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990. ISBN: 0-262-02313-X.

[4] Sebastian Böcker. "A golden ratio parameterized algorithm for Cluster Editing". In: *J. Discrete Algorithms* 16 (2012), pp. 79–89. DOI: 10.1016/j.jda.2012.04.005. URL: https://doi.org/10.1016/j.jda.2012.04.005.

[5] Yixin Cao and Jianer Chen. "Cluster Editing: Kernelization based on edge cuts". In: *Algorithmica* 64.1 (2012), pp. 152–169. DOI: 10.1007/s00453-011-9595-1. URL: https://doi.org/10.1007/s00453-011-9595-1.

[6] Jianer Chen and Jie Meng. "A 2k kernel for the cluster editing problem". In: *J. Comput. Syst. Sci.* 78.1 (2012), pp. 211–220. DOI: 10.1016/j.jcss.2011.04.001. URL: https://doi.org/10.1016/j.jcss.2011.04.001.

[7] Elias Dahlhaus. "Efficient parallel modular decomposition (Extended Abstract)". In: *Graph-Theoretic Concepts in Computer Science, 21st International Workshop, WG '95, Aachen, Germany, June 20-22, 1995, Proceedings.* 1995, pp. 290–302. DOI: 10.1007/3-540-60618-1_83. URL: https://doi.org/10.1007/3-540-60618-1%5C_83.

[8] Peter Damaschke. "Parameterized enumeration, transversals, and imperfect phylogeny reconstruction". In: *Theor. Comput. Sci.* 351.3 (2006), pp. 337–350. DOI: 10.1016/j.tcs.2005.10.004. URL: https://doi.org/10.1016/j.tcs.2005.10.004.

[9] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. "Efficient parallel merge sort for fixed and variable length keys". In: *Proceedings of Innovative Parallel Computing (InPar 12).* 2012. DOI: 10.1109/InPar.2012.6339592.

[10] Frank K. H. A. Dehne, Michael A. Langston, Xuemei Luo, Sylvain Pitre, Peter Shaw, and Yun Zhang. "The Cluster Editing problem: Implementations and experiments". In: *Parameterized and Exact Computation, Second International Workshop, IWPEC 2006, Zürich, Switzerland, September 13-15, 2006, Proceedings.* 2006, pp. 13–24. DOI: 10.1007/11847250_2. URL: https://doi.org/10.1007/11847250%5C_2.

[11] Aditya Deshpande and P. J. Narayanan. “Can GPUs sort strings efficiently?” In: *20th Annual International Conference on High Performance Computing, HiPC 2013, Bengaluru (Bangalore), Karnataka, India, December 18-21, 2013*. 2013, pp. 305–313. DOI: 10.1109/HiPC.2013.6799129. URL: https://doi.org/10.1109/HiPC.2013.6799129.

[12] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[13] Michael R. Fellows, Michael A. Langston, Frances A. Rosamond, and Peter Shaw. “Efficient parameterized preprocessing for Cluster Editing”. In: *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings*. 2007, pp. 312–321. DOI: 10.1007/978-3-540-74240-1_27. URL: https://doi.org/10.1007/978-3-540-74240-1%5C_27.

[14] Michael Fellows, Michael Langston, Frances Rosamond, and Peter Shaw. “Polynomial-time linear kernelization for Cluster Editing”. In: *Fundamentals of Computation Theory, 16th International Symposium, FCT 2007, Budapest, Hungary, August 27-30, 2007, Proceedings*. 2007.

[15] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. “Graph-Modeled data clustering: Fixed-Parameter algorithms for Clique Generation”. In: *Algorithms and Complexity, 5th Italian Conference, CIAC 2003, Rome, Italy, May 28-30, 2003, Proceedings*. 2003, pp. 108–119. DOI: 10.1007/3-540-44849-7_17. URL: https://doi.org/10.1007/3-540-44849-7%5C_17.

[16] Jiong Guo. “A more effective linear kernelization for cluster editing”. In: *Theor. Comput. Sci.* 410.8-10 (2009), pp. 718–726. DOI: 10.1016/j.tcs.2008.10.021. URL: https://doi.org/10.1016/j.tcs.2008.10.021.

[17] Michel Habib, Fabien de Montgolfier, and Christophe Paul. “A simple linear-time modular decomposition algorithm for graphs, using order extension”. In: *Algorithm Theory - SWAT 2004, 9th Scandinavian Workshop on Algorithm Theory, Humlebaek, Denmark, July 8-10, 2004, Proceedings*. 2004, pp. 187–198. DOI: 10.1007/978-3-540-27810-8_17. URL: https://doi.org/10.1007/978-3-540-27810-8%5C_17.

[18] Torben Hagerup. “Optimal parallel string algorithms: sorting, merging and computing the minimum”. In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*. 1994, pp. 382–391. DOI: 10.1145/195058.195202. URL: https://doi.org/10.1145/195058.195202.

[19] Wen-Lian Hsu and Tze-Heng Ma. “Substitution decomposition on chordal graphs and applications”. In: *ISA ’91 Algorithms, 2nd International Symposium on Algorithms, Taipei, Republic of China, December 16-18, 1991, Proceedings*. 1991, pp. 52–60. DOI: 10.1007/3-540-54945-5_49. URL: https://doi.org/10.1007/3-540-54945-5%5C_49.

[20] Nadathur Satish, Mark J. Harris, and Michael Garland. “Designing efficient sorting algorithms for manycore GPUs”. In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. 2009, pp. 1–10. DOI: 10.1109/IPDPS.2009.5161005. URL: https://doi.org/10.1109/IPDPS.2009.5161005.

[21] Marc Tedder, Derek G. Corneil, Michel Habib, and Christophe Paul. “Simpler linear-time modular decomposition via recursive factorizing Permutations”. In: *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*. 2008, pp. 634–645. DOI: 10.1007/978-3-540-70575-8_52. URL: https://doi.org/10.1007/978-3-540-70575-8%5C_52.

[22] Tobias Wittkop, Sita Saunders, Sven Rahmann, Mario Albrecht, John Morris, Sebastian Böcker, Jens Stoye, and Jan Baumbach. “Partitioning biological data with transitivity clustering”. In: *Nature Methods* 7 (June 2010), pp. 419–20. DOI: `10.1038/nmeth0610-419`.

[23] Tobias Wittkop, Anke Truss, Mario Albrecht, Sebastian Böcker, and Jan Baumbach. “Comprehensive cluster analysis with Transitivity Clustering”. In: *Nature protocols* 6 (Feb. 2011), pp. 285–95. DOI: `10.1038/nprot.2010.197`.