Fakultät für Elektrotechnik und Informatik
Technische Universität Berlin

Algorithm Engineering for Hierarchical Tree Clustering

Bachelorarbeit

Tomasz Przedmojski
Matrikel-Nummer 325532

Betreuer	Prof. Dr. Rolf Niedermeier
	Dr. Christian Komusiewicz
	Sepp Hartung
Erstprüfer	Prof. Dr. Rolf Niedermeier
Zweitprüfer	Prof. Dr. Stephan Kreutzer

Datum der Einreichung: 27. Januar 2013

Abstract (Deutsch)
Diese Arbeit setzt sich mit der Anwendung von Datenreduktionsregeln für einen parametrisierten Algorithmus für M-Hierarchical Tree Clustering auseinander. Mittels der Methodik des Algorithm Enginnering wird eine theoretische Datenreduktionsregel implementiert und im Kontext eines Suchbaumalgorithmus ausgewertet. Der im Algorithm Engineering vorgesehene Entwicklungszyklus wird dann noch einmal angewandt und die Regel wird theoretisch und praktisch weiterentwickelt und ausgewertet. In Rahmen der Experimente werden die unterschiedlichen Reihenfolgen der Datenreduktionsregeln untersucht und als eine mögliche Optimierung erkannt. Die Einschränkungen der gewählten Methodik werden diskutiert.

Abstract (English)
This work deals with data reduction rules for a parameterized algorithm for the M-Hierarchical Tree Clustering problem. With help of the Algorithm Engineering framework a theoretic data reduction rule is implemented and evaluated in context of a search tree algorithm. The Algorithm Engineering cycle is then applied for the second time to improve, implement and evaluate the rule. During the experiments step the importance of the order of the rules on the performance is recognized. Finally, the limits of the framework are discussed.

Contents

1 Introduction

Hierarchical clustering is a tool used in biology, social sciences, and statistics [17][1][16][8][12] to obtain a hierarchical representation of the input data. Through recursive partitioning of an input set, a tree structure is formed. The leaves of this tree represent the items from the input and the inner nodes model clusters. The more similar two items are, the smaller is the distance between the leaves representing them. By means of hierarchical clustering the input data can be analyzed and understood at multiple levels of similarity: the further away from root node (which stands for the whole set), the smaller the clusters and more related the items are. The total number of clusters or the height of the tree does not have to be specified upfront, it emerges during the construction of the tree, depending on the input. The NP-hard problem of constructing an optimum hierarchical clustering is called M-HIERARCHICAL TREE CLUSTERING.

Even though (unless P=NP) no NP-hard problem can be solved efficiently, many real-life inputs can be solved in reasonable time with the help of preprocessing techniques, heuristics, and optimizations. This is our main motivation: given an existing solver for M-HIERARCHICAL TREE CLUSTERING, we hope to improve its performance by implementing, optimizing and extending a data reduction rule which has been theoretically developed but judged to be impractical by Guo et al. [13]. As the feasible instances are rather small (less than 200 items), we will focus on the speed of the solver.

We seek to employ the Algorithm Engineering methodology to guide our efforts throughout this work. Algorithm Engineering aims to bridge the gap between theoretical and practical results. This problem is precisely our starting point: promising theoretical development that is too slow to be useful when implemented in a straightforward way.

1.1 Problem Definition

We first define a simpler problem, which is related to M-HIERARCHICAL TREE CLUSTERING, namely CLUSTER EDITING [21][14][5] (also known as CORRELATION CLUSTERING on complete graphs [3][2]). CLUSTER EDITING is a graph editing problem where one seeks to turn an input graph into a disjoint set of cliques. This is achieved by adding or removing edges from the graph. The number of modifications is constrained by an additional parameter k.

Problem definition 1. CLUSTER EDITING *(CE):*
INPUT: *An undirected graph $G = (V, E)$ and an integer $k \geq 0$.*
TASK: *Find a set $S \subseteq \{\{u, v\} : u \in V, v \in V\}$ with $|S| \leq k$, such that $G' = ((E \backslash S) \cup (S \backslash E), V)$ is a vertex-disjoint union of cliques.*

The set S is also called the set of *edge modifications*. Each element $s = \{u, v\}$ of this set either removes an edge from G (when $s \in E$) or adds and edge to G (otherwise). To solve the related optimization problem, one has to find the minimum integer k such that (G, k) is a YES-Instance of CLUSTER EDITING. We will approach this problem using the idea of a conflict.

Definition 1. *A* conflict *or a P_3 in* CLUSTER EDITING *is a triple of vertices (u,v,w), for which it holds that $\{u, v\} \in E$, $\{v, w\} \in E$ and $\{u, w\} \notin E$.*

A graph without any induced P_3 is a vertex-disjoint union of cliques. With this definition one can solve CE by finding a set of modifications of minimum size that resolves all conflicts in G. A conflict is resolved when either the edge $\{u, v\}$ or the edge $\{v, w\}$ is removed, or the edge $\{u, w\}$ is added to G.

Before we come to the definition of M-HIERARCHICAL TREE CLUSTERING we first need to define a *distance function* and an *ultrametric*. Let from now on M be an integer that denotes the maximum number of inner nodes between a leaf and the root node in the clustering tree. Notice the fact that CLUSTER EDITING is equivalent to 1-HTC, since 1-HTC constructs a tree with maximum depth of one: inner nodes of this tree model the clusters constructed through CE. We will take advantage of this observation throughout this work.

Definition 2. *A* distance function *for a node set X is a symmetric function $D : X \times X \rightarrow \{0, ..., M+1\}$ for which it holds that $D(u,v) > 0$ when $u \neq v$ and $D(u,v) = 0$ otherwise.*

We consider a special kind of distance function, a so-called ultrametric. An ultrametric is a distance function which is free of conflicts.

Definition 3. *A distance function U is an* ultrametric *when for all $i,j,k \in X$: $U(i,j) \leq max\{U(i,k), U(j,k)\}$.*

Definition 4. *A* conflict *in M-HTC is a triple (j,l,m) for which it holds that $D(j,l) > D(j,m)$ and $D(j,l) > D(l,m)$; the pair (j,l) is then called the* max distance pair.

A distance function for a CE instance (G,k) would be a binary function and its matrix representation would be the the adjacency matrix of G.

To construct a hierarchical tree of clusters, one has to transform the input distance function into an ultrametric.

Definition 5. *A* modification *to a distance function D is a triple $(u,v,d) \in (X \times X \times \mathbb{N})$ whose application results in another distance function D' with $D'(x,y) = D'(y,x) = d$ if $(x = u \wedge y = v)$ and $D'(x,y) = D'(y,x) = D(x,y)$ otherwise.*

When dealing with sets of modifications in M-HTC we often use the value of such set which is defined as follows:

Definition 6. *The value of a modification set S to a distance function D is equal to $\sum_{(u,v,d)\in S} |D(u,v) - d|$.*

Given the above definitions, we can now formally state the M-HIERARCHICAL TREE CLUSTERING problem.

Problem definition 2. M-HIERARCHICAL TREE CLUSTERING *(M-HTC)*
INPUT: *A distance function D, a node set X, and an integer $k \geq 0$*
TASK: *Set of modifications S whose value is at most k and that turns D into an ultrametric.*

A triple $I = (D, X, k)$ is an input instance for the decision problem as defined above. The matching optimization problem consist of finding the minimum value of k such that (D, X, k) is a YES-instance. This can be computed easily, provided we have a solver for the decision problem.

1.2 Notation

Besides the already introduced definitions, we will use a notation based on standard graph theory notation [22]. Our extensions are summarized in the following table:

Notation	Meaning
$N(v), v \in V$	Set of neighbors of the node v in a graph
$N(K), K \subseteq V$	$(\bigcup_{x \in K} N(x)) \backslash K$
$N^1(K), K \subseteq V$	first neighborhood of K, equivalent to $N(K)$
$N^2(K), K \subseteq V$	$(\bigcup_{L \in N^1(K)} N^1(L)) \backslash N^1(K)$, second neighborhood of K
$D[Y], Y \subseteq X$	a distance function $D[Y] : Y \to \mathbb{N}$ with $D[Y](l, m) := D(l, m)$
$D[a, b \to d]$	a distance function that is a copy of D for which holds $D(a, b) = D(b, a) = d$

1.3 Pseudo Code

For describing algorithms we employ a generic form of pseudo code that bears some resemblance to Pascal code, although the used constructs such as **if** control structures, procedures (**functions**), **while** and **for** loops (iterating over items in collection) can be found in very similar form in many imperative programming languages such as Basic, C, Python or Java. Single statements borrow heavily from our notation and should be easy to comprehend.

Each pseudo code listing consists of one or more procedures. In the latter case procedures are grouped together because they perform a shared task. A procedure definition can be preceded by a requirements section signified by lines starting with a keyword "**Require:**". They express constraints on passed arguments and allow to abstract away error checking and error handling that would normally be done by the programmer. This in turns allows to focus on core ideas of the employed algorithms.

In some cases variables are updated, in the spirit of imperative programming languages, which makes it more difficult for a programmer willing to implement the pseudo code in a programming language with immutable variables (most functional languages fall into this category). We generally tend to avoid mutable variables, nevertheless in some cases we do it for readability.

1.4 Parameterized complexity and FPT problems

Both M-HTC and Cluster Editing are NP-hard problems [17], which means that a polynomial-time algorithm for solving them is impossible unless P=NP.

To better understand the complexity of the underlying problems we must employ parameterized complexity analysis (thoroughly described by Downey and Fellows [10]). Parameterized complexity classifies problems according to their difficulty with respect to multiple parameters, whereas traditional analysis focues only on one parameter: the size of the input.

The set of problems that is of particular interest consists of so-called fixed-parameter tractable (FPT) problems [9]. Problem A is in FPT, when besides the size of the input it takes a parameter k, and there is a solving algorithm running in $f(k) \cdot |x|^{O(1)}$ time, where x is the size of the input. A function f is typically some exponential function, for example 2^k. FPT problems have two very useful features: for given (fixed) parameter k, the problem becomes a polynomial time solvable, and the resulting polynomial function is independent from k. For small values of k many real-world instances of FPT problems can be solved in reasonable time.

1.5 Data reductions

A data reduction is a polynomial-time algorithm that turns an instance I of a problem A into another, equivalent instance I' of the same problem A, which is smaller (and hopefully easier to solve). A data reduction rule is *correct* if the I' a YES-instance if and only if the original instance I is a YES-instance. We use following formal definition in the context of M-HTC:

Definition 7. *A* data reduction *is a polynomial-time algorithm that given $I = (D, X, k)$ outputs $I' = (D', X, k')$, such that I' is a YES-instance of M-HTC if and only if I is a YES-instance.*

Applying data reduction rules to the instance is also known as *kernelization* and the resulting instance is called *problem kernel* [15].

1.6 Methodology

The methodology in this work is based on the algorithm engineering cycle proposed by Peter Sanders [20]. It applies two well-known principles to the domain of algorithm engineering: the scientific method and the waterfall model. Since the former doesn't require any introduction, as it is the basis for attaining human knowledge, we will focus on the latter. The Waterfall model [4] is a framework for developing computer software which consists of an ordered sequence of steps necessary for successful creation and adoption of software. It is one of the most broadly employed software development processes, spawning multiple alternatives or refinements. It should be noted, that Sanders' approach uses multiple iteration in his model, a characteristic common to many models improving upon the waterfall model [18][6].

Sanders made use of both approaches in attempt to bridge the gap between theory and practice, because he noticed three flaws in traditional approaches, when both aspects are treated separately:

- realistic hardware with parallelism, memory organization and hierarchies diverges from simple mathematical models,
- elaborate theoretical ideas are often not directly implementable, and
- real-world data has different performance characteristics than the worst-case analysis predicts.

The cycle (show in Figure 1.1) starts with an *application*, or a group there of, for which an algorithm is developed. Next step in the process deals with formulating a *realistic model* for the problem. After that the cycle is stated with *design*, followed by *analysis*, *implementation* and *experiments*. The design and analysis phases entail the elements known from classical algorithm theory: working with mathematical models of the target machine and underlying problem, formulating an algorithm, proving its validity and performance guarantees. Implementation transports the theoretical knowledge into the realm of software development and confronts simplified models with constraints of hardware and software platforms. Last phase of the cycle is the most crucial for achieving the desired purpose of combining theory and practice: the experiments. Their aim is to assess an algorithm's usefulness on the basis of the performance with real-world data. The conclusions from the experiments should then in turn be used to shape another iteration. Algorithms that

meet the (both: practical and theoretical) requirements should then be placed into an algorithm library, a collection of well-tested procedures which can be reused in other applications.

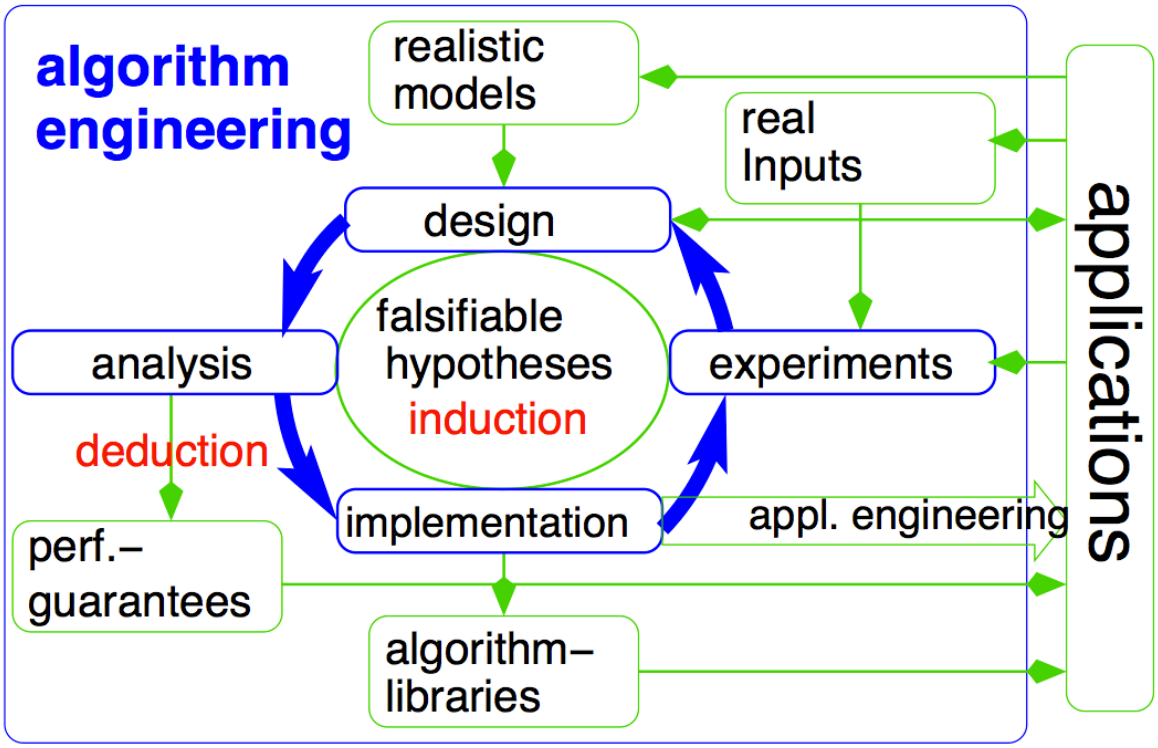

Figure 1.1: Algorithm enginnering as proposed by Sanders [20].

Our goal is to employ the methodology to improve upon the algorithm at hand and document the whole process.

1.7 Experiments

1.7.1 Test data

We base our experiments on two data sets: biological (real world) data used by Guo et al. in their work [13] and randomly generated synthetic data. Because of our chosen framework, which focuses heavily on real-world performance, the first data set should suffice to draw

meaningful conclusions. We are skeptical of this claim and will put it to a test, by comparing the results on biological and synthetic data.

We opt for also using randomized data, because we believe that it establishes lower bound on the performance of the algorithm, should the application change.

For the real-world data we use a data pool of 3964 files with pairwise similarity data of sets of proteins. The pool has been further restricted to those files containing < 70 proteins (i.e. nodes in the input instance), which covers more than 85% of the files.

For the random data we generated instances with different values of n (number of elements) and k and M. We considered the instances with sizes ranging from $n = 80$ to $n = 150$ and three values for M: 3, 4, and 5. For each combination of n and M we iterated over values of k from 1 to n^2 and on each step created random 4 ultrametrics. Each of the random ultrametrics was used to create four input instances by applying k modifications to the ultrametric at random. Hence for each combination of n, M, and k 16 instances in total have been generated. Details on the generation procedure can be found in the work by Guo et al.PAPER.

1.7.2 Test setup

All experiments have been executed on a 3.6 GHz Intel Xeon E5-1620 CPU with Oracle Java Runtime Environment 1.6.0_24. The virtual machine in each test has been started with options -Xms2G -Xmx2G.

During test runs each instance is given a time limit of 300 seconds. If no solution can be found within that time, then the execution is halted and the given instance marked as "unsolvable". For synthetic instances, consecutive values of k are tested for each pair of n (number of elements) and M (maximum distance) from the data set. If more than 10 consecutive instances cannot be solved, then the run is aborted and next pair of n and M is used.

2 Basic Version

In this chapter we first describe the existing M-HTC solver developed by Guo et al. [13], which consists of a search tree-based algorithm and a set of data reduction rules. Then we proceed to describe the critical clique data reduction rule, which has been theoretically developed alongside the solver, but hasn't been put into use because of the performance issues.

By using existing software and theoretical basis we fast forward in our algorithm engineering process to the analysis step. We analyze the weakness of the critical clique rule and implement it using a different approach.

Finally we measure the impact of the critical clique reduction rule and the performance of the solver in a series of experiments.

2.1 An exact algorithm for M-HTC

The idea for the exact algorithm used in the provided M-HTC solver is very simple: abort the program if current value k is less than 0, since the instance is unsolvable. Otherwise find a conflict (j, l, m) where (j, l) is a max distance pair, and recursively branch into three cases:

- Decrease the distance between j and l by 1 and set $k := k - 1$
- Increase the distance between j and m by 1 and set $k := k - 1$
- Increase the distance between l and m by 1 and set $k := k - 1$

The algorithm terminates successfully when there are no conflicts left.

Obtaining a solver for the M-HTC optimization problem is trivial, given the above algorithm. The main procedure is surrounded by a loop that iterates over all possible values of k from 0 in the increments of 1.

For detailed proof of the correctness of the above algorithm and its time complexity see [13].

Claim 1. *The simple search tree solver has the time complexity $O(k \cdot 3^k \cdot n^3)$ and thus M-HTC is fixed-parameter tractable.*

Proof of Claim 1. Since the search tree solver branches into three cases, each time decreasing the value of k by 1 we get a branching vector of $(1, 1, 1)$. Through standard search tree analysis [19] we get an $O(3^k)$ time complexity for the branching part of the solver. In each recursive step, the algorithm searches for conflicts, which can be implemented trivially in $O(n^3)$. The outer loop is executed k times, which leads to $O(k \cdot 3^k \cdot n^3)$. Therefore M-HTC is fixed-parameter tractable. □

The solver that we use employs the following set of data reduction rules, which we give without proof of correctness:

Reduction Rule 1. *If there is a pair $\{i, j\} \subseteq X$ which is the max-distance pair (or not the max-distance pair) in at least $k + 1$ conflicts, then decrease (or increase) the distance $D(i, j)$ by one and decrease the parameter k by one.*

Reduction Rule 2. *Remove all elements which are not a part of a conflict.*

Reduction Rule 4. *For any triple (l, k, m) that is not a conflict and where $D(l, k) < D(l, m) = D(k, m)$ do the following: if (l, k) is marked as a pair for which $D(l, k)$ may not be increased, and (l, m) is marked as a pair for which $D(l, m)$ may not be decreased (or increased), then mark (k, m) as a pair for which $D(k, m)$ may not be decreased (or increased).*

Reduction Rule 5. *An* independent conflict *is a conflict (l, k, m), for which it holds, that there are no conflicts with containing the pairs (l, k), (l, m) or (k, m). Solve all independent conflicts.*

Reduction Rule 4 is given here without details. It uses meta-information about pairs in the distance function: when it is certain that a distance for a pair should not be increased and/or decreased to deduce further meta-information of this type. Empirical observation has shown that it has negligible effect.

These rules are applied consecutively every two steps (i.e. for every second case that the search algorithm branches into). This factor of 2 can be adjusted and is subject to fine-tuning. Branches in the search tree that are recognized by any rule as unsolvable are not pursued any further by the algorithm.

The reduction rule number three, which is missing from the above list, we call *Critical Clique Reduction Rule*. This rule is our main focus in this chapter, therefore we describe it in more detail in the next section.

2.2 Critical Clique Reduction Rule

The Critical Clique data reduction rule is one of the main points of this work. As already mentioned, its theoretical basis has been developed by Guo et al. [13], but a straightforward implementation failed to deliver reasonable performance.

We will briefly describe the idea behind this reduction rule. At its core the rule is very simple, and is an extension of a similar rule for CLUSTER EDITING: nodes with equal neighborhoods should never be separated [14]. Since in M-HTC we do not work with graphs, the idea requires a slightly more complicated approach, which motivates the following two concepts: a *t-threshold graph* and a *critical clique*.

Definition 8. *A* threshold graph $G_t = (V_t, E_t, t)$ *of an M-HTC instance* $I = (D, X)$ *is an undirected graph with following properties:*

1. $V_t = X$
2. $\{u, v\} \in E_t$ *iff* $D(u, v) \leq t$

Consider such t-threshold graph: we observe that some nodes in this graph have equal neighborhoods thus form cliques. Cliques, or *t-clusters* that contain all nodes with equal neighborhood we call *critical cliques*. Resulting graph G_t^C is called *critical clique graph*.

Definition 9. *A* critical clique *graph* $G_t^C = (V, E)$ *is the graph constructed from a threshold graph* $G_t = (V_t, E_t, t)$*, and* G_t^C *satisfies the following conditions:*

1. $\bigcup_{C \in V} C = V_t$ *(every vertex from* G_t *is in at least one clique),*
2. $\forall (C_1, C_2) \in V \times V : C_1 \cap C_2 = \emptyset$ *(pair-wise disjoint sets, i.e., a node is at most in one clique), and*
3. $\forall C \in V : (\forall \{u, v\} \subseteq C : N(u) \cup \{u\} = N(v) \cup \{v\})$ *(every vertex in a clique is adjacent to the same vertex set).*

The Critical Clique rule is based on two ideas. Some critical cliques are large enough, so that they are never split, that is the distance between nodes in a critical clique is never set above the threshold t. Even larger critical cliques "absorb" the elements from their neighborhood by lowering the distance between nodes in the critical clique and a neighbor to the value of t.

The rule consist of two procedures: one that recursively iterates of all possible thresholds and one that applies the reduction for given threshold. Pseudo-code for these rules can be found in Procedure 1 and 2.

Procedure 1 Critical Clique

```
1:  procedure CCRecursive(D, X, t)
2:      CCC(D, X, t)
3:      Construct G_t^C[X] from D
4:      for all isolated cliques K in G_t^C[X] do
5:          if D[K] is an ultrametric then
6:              X := X\K                ▷ Isolated ultrametrics can be safely removed
7:          else
8:              CCRecursive(D,K,t-1)
9:          end if
10:     end for
11:     return X
12: end procedure
```

Procedure 2 Critical Clique Construction

1: **procedure** CCC(D, X, t)
2: Construct $G_k^C[X]$ from D
3: **while** $G_k^C[X]$ contains a nonisolated critical clique K with
- $K \subseteq X$
- $|K| \geq t \cdot |N(K)|$
- $|K| \geq t \cdot |N^2(K)| + |N(K)|$

do
4: **for all** $(u, v) \in (N(K) \times K \backslash N(K))$ **do** ▷ Absorb the neighbors
5: $t := t - |D(u, v) - t + 1|$
6: $D(u, v) := t + 1$
7: **end for**
8: **for all** $u, v \in N(K)$ with $D(u, v) = t + 1$ **do** ▷ Connect the neighbors
9: $t := t - |D(u, v) - t|$
10: $D(u, v) := t$
11: **end for**
12: **end while**
13: **end procedure**

2.3 Implementation

Data structures

While implementing the algorithms presented and used throughout this work we employed several data structures, some of which exhibit interesting characteristics. We represent distance functions as two-dimensional arrays, but, exploiting the symmetry of the source, to save memory we only store a half of the source quadratic matrix (which fully describes the function).

We have chosen to implement graphs using adjacency sets. Since both adjacency lists and adjacency matrices would exhibit their drawbacks (slow checking if two nodes are adjacent for lists, slow iteration over neighbors for matrices) in our application, we went for adjacency sets that combine advantages of both: $O(|N(v)|)$ iteration over the neighborhoods of a vertex and $O(1)$ test whether two vertices are adjacent.

Furthermore the underlying set has been implemented using a very memory and time efficient data structure: sparse integer set. Sparse integer set, as described by Briggs and Torczon [7] uses a counter, called **members** and two arrays, called **sparse** and **dense** to store integers in range $0..n$. The arrays and the counter are filled with zeros on initialization. Procedure 3 describes adding an element to a set:

Procedure 3 Adding an element to a sparse integer set

```
1: procedure AddElementToSet(value, sparse, dense, members)
2:     if ¬ SetContains(value, sparse, dense, members) then
3:         sparse[value] := members
4:         dense[members] := value
5:         members := members + 1
6:         return true
7:     end if
8:     return false
9: end procedure
```

To test whether an element is present in a sparse integer set one has to call Procedure 4:

Procedure 4 Adding an element to a sparse integer set

```
1: procedure SetContains(value, sparse, dense, members)
2:     l := sparse[value]
3:     if l < members then
4:         if dense[l] = value then
5:             return true
6:         end if
7:     end if
8:     return false
9: end procedure
```

To give better understanding of the sparse integer set we simulate adding three elements 5, 1, and 4 to a set. In our example the set is able to hold in range $(0, ..., 6)$.
Figure 2.1 depicts the contents of the two arrays after adding the elements. Notice that

$dense[sparse[value]] = value$ for all values that are in the set. This is the crucial property of the sparse integer set which allows for time characteristics described above.

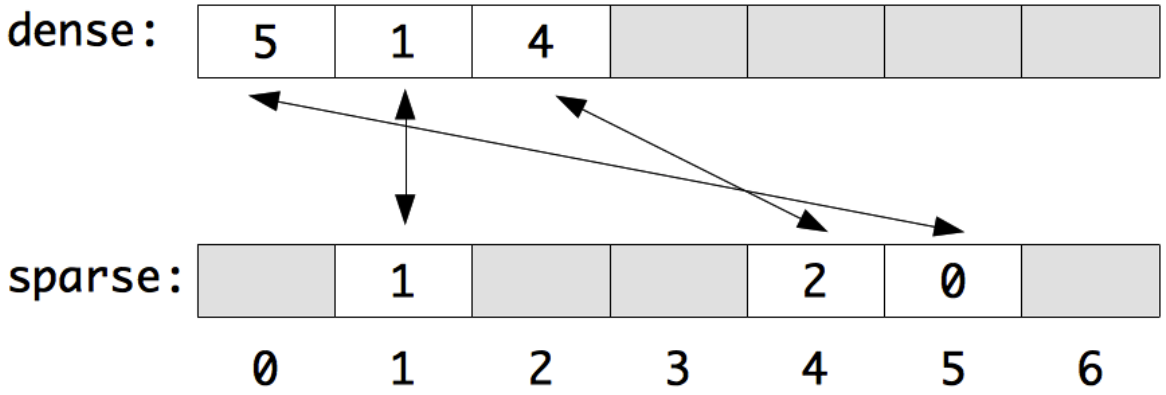

Figure 2.1: A sparse integer set holding three elements: 5,1, and 4.

Iterating over the elements of the set is just a matter of iterating over i first elements of the dense array, where i is equal to the *members* counter of the sparse integer set.

Since in our case the maximum value of an element added to a set is bounded by the number of nodes in graph, which is small, the memory overhead in negligible and we can use sparse integer set to our advantage.

Critical Clique reduction rule

The implementation strictly follows the pseudo code formulation (see Procedure 1) of the rule with one significant change made to the design.

The change is motivated by an obvious inefficiency of the rule as stated: the critical clique graph is reconstructed on every call to the reduction rule. Therefore very similar graphs are constructed in short succession. The redundant computation slowed down the program considerably and led to the exclusion of the rule from the original solver [13].

Since our goal is to make the rule usable and apply it as often as possible to maximize its effect, we aim to eliminate as much redundancy as possible from the computation to improve the running-time efficiency. We solve this problem with a simple trick: we generate a list of threshold graphs and a list of critical clique graphs for every possible threshold (altogether

$2 \cdot M$ graphs). We do this only once on first call to the rule for a given input instance. On subsequent calls we use the same graphs, which that are maintained between the calls.

Whenever the distance function is modified, a callback function is called, that efficiently reconstructs parts of graphs that are inconsistent with the distance function. Therefore every application of critical clique (with exception of the first one) skips the graph construction step.
The amortized cost of maintaining graphs is obviously much smaller than the cost introduced by the inherently redundant computation.

Maintaining threshold graphs. The task of maintaining a list of M threshold graphs is straightforward. This task is expressed with pseudo code in Procedure 5.

Procedure 5 Maintaining threshold graphs

Require: $|k - l| = 1$
Require: $|tgraphs| \geq max(k, l)$
Require: $min(k, l) \geq 0$
1: **procedure** UPDATETHRESHOLDGRAPHS($tgraphs, u, v, k, l$)
2: **if** $k > l$ **then**
3: $graph := tgraphs[l]$
4: $graph := graph \cup \{\{u, v\}\}$
5: **else**
6: $graph := tgraphs[k]$
7: $graph := graph \backslash \{\{u, v\}\}$
8: **end if**
9: **end procedure**

Maintaining clique graphs. The task of maintaining a list of M clique graphs is slightly more complex than the one for threshold graphs. We must take into account that each modification (u, v, d) to the distance function affects the neighborhoods of u and v (for thresholds in range $D[u, v] - |d|$ to $D[u, v] + |d|$), breaking the equivalence between u (respectively v) and other members of the same clique. As a result u (respectively v) may form a new clique, or join an other, existing clique. Details of such changes are embodied in the Procedure 6.

One important sub-procedure is a simple greedy algorithm, that iterates over pairs of cliques in a critical clique graph and joins them is their respective elements have equal neighborhoods. Pseudo-code detailing this process can be found in Procedure 7.

2.4 Experiments

With our experiments we hope to answer couple of questions:

1. How does the relationship between n (respectively k) and the running time look like?
2. Does the rule slow down the solver significantly, as did the original implementation?
3. What is the net effect of the rule, i.e. is the solver able to solve more instances?

We conducted the experiments as described in the first chapter. The results for the biological (real world) data are summarized in the following table. The function of k (and n) which expresses the relationship between time and value of k has been obtained by calculating exponential regression with the **nls** function (which calculates nonlinear least-squares estimates) from the :)standard library bundled with the R language. The solver has been tested with critical clique rule enabled (basic) and without it (vanilla).

Program version	Solved	Total	Function of k	Function of n
Unmodified (vanilla)	2869	3398	1.0304^k	1.118^n
With critical clique (basic)	2868	3398	1.0304^k	1.119^n

The obtained exponential functions have similar base for both k and n, but the plots for the number of nodes (Figure 2.2) are more scattered that the ones for k (Figure 2.3), which form an exponential curve. There are only few "lucky" instances with k value larger than 140 that have been solved, which influence the shape of the curve disproportionately.

We observe that the running time of the program is widely different than the theoretical 3^k. This can be easily explained by the fact that the solver is already heavily tuned and contains several data reduction rules. It aligns well with our initial assumption that many instances can be solved efficiently because of the pre-processing with data reduction rules.

Procedure 6 Maintaining clique graphs

Require: $|k - l| = 1$
Require: $|cgraphs| \geq max(k, l)$
Require: $min(k, l) \geq 0$

procedure UPDATECLIQUEGRAPHS($cgraphs, u, v, k, l$)
  if $k > l$ **then**
    ADDEDGETOCLIQUEGRAPH($cgraphs[l], u, v$)
  else
    REMOVEEDGEFROMCLIQUEGRAPH($cgraphs[k], u, v$)
  end if
end procedure
procedure ADDEDGETOCLIQUEGRAPH($cgraph, u, v$)
  $cliqueU := cgraph[u]$
  $cliqueV := cgraph[v]$
  $cliqueU := cliqueU \backslash \{u\}$
  $cliqueV := cliqueV \backslash \{v\}$
  $cliqueU' := \{u\}$
  $cliqueV' := \{v\}$
  $cgraph := cgraph \cup \{\{cliqueU, cliqueU'\}, \{cliqueV, cliqueV'\}, \{cliqueU, cliqueV\}\}$
    ▷ Remove u and v from their cliques and create single-element cliques for them
  MAXIMIZECLIQUES($cgraph$) ▷ Ensure that all cliques are critical cliques
end procedure
procedure REMOVEEDGEFROMCLIQUEGRAPH($cgraph, u, v$)
  $cliqueU := cgraph[u]$
  $cliqueV := cgraph[v]$
  $cliqueU := cliqueU \backslash \{u\}$
  $cliqueV := cliqueV \backslash \{v\}$
  $cliqueU' := \{u\}$
  $cliqueV' := \{v\}$
  $cgraph := cgraph \cup \{\{cliqueU, cliqueU'\}, \{cliqueV, cliqueV'\}$
    ▷ Remove u and v from their cliques and create single-element cliques for them
  MAXIMIZECLIQUES($cgraph$) ▷ Ensure that all cliques are critical cliques
end procedure

Procedure 7 Maximizing cliques in clique graph

```
1:  procedure MaximizeCliques(cgraph)
2:      for all c is a clique in cgraph do                ▷ Remove empty cliques from graph
3:          if |c| = 0 then
4:              cgraph := (V\{c}, E\{{u, *} : u = c})
5:          end if
6:      end for
7:      for all c is a clique in cgraph do
8:          for all n ∈ N_cgraph[c] do
9:              if N_cgraph[c] ≈ N_cgraph[n] then
10:                 Replace c and n by a clique containing elements from c and n with
    identical neighborhood
11:             end if
12:         end for
13:     end for
14: end procedure
```

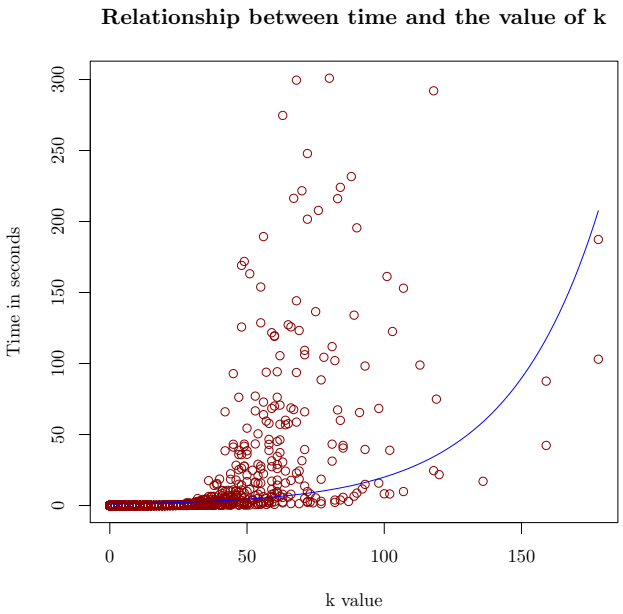

Figure 2.2: Real data: relationship between value of k and time.

The relationship between n and time, depicted in Figure 2.3 is inconclusive. We note that very small instances ($n < 13$) are trivial to solve. Most instances fall into 15..35 range for the number of elements, which is too small to make general statements.

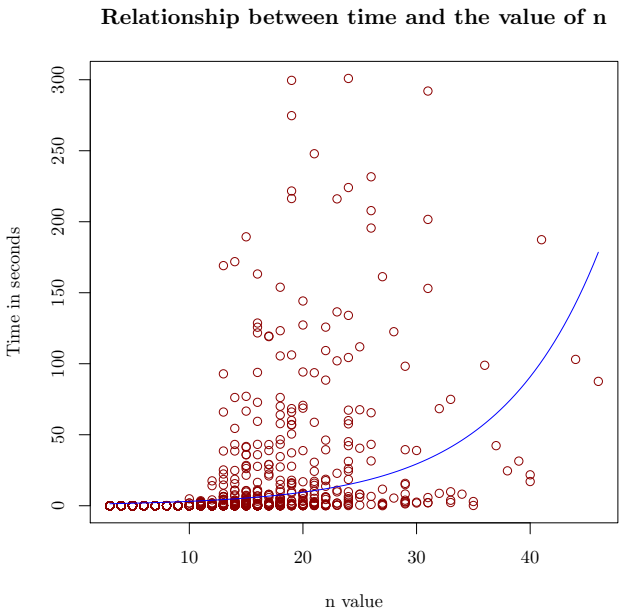

Figure 2.3: Real data: relationship between value of n and time.

As to the effect of the rule on the solver we note that the solver was able to solve one instance less with the reduction rule when compared to the version without. Our implementation mitigated the shortcomings of the naive implementation: very poor performance.

To measure the effects of the rule, we also investigate the effect of the critical clique rule on the number of the recursive steps in the solver: in few cases the critical clique rule lead to a minimal (< 20) increase the number of recursive steps in the branching algorithm. These insignificant cases have excluded for clarity from a diagram depicting the decrease in the number of recursive steps in Figure 2.4.

The changes in the number of recursive steps should lead to shorter running times, as there is a strong correlation between the two (see Figure 2.5). We believe that the cost of the application of the critical clique rule is larger than the gains through reducing the number of recursive steps. We also observed cases of instances solved in very short time, but with very large number of recursive steps. It is probable that their structure doesn't allow for data reduction, yet allows for fast "brute-force" computation.

Figure 2.4: Real data: decrease in the number of recursive steps when employing the Critical Clique reduction rule.

The analysis performed on synthetic data suggests that the basic version (with critical clique reduction rule) performs a little bit better (about 0.6%), which given the nature of the benchmark is insignificant. Summarized results can be found in the table below:

Version	Solved instances	Function of k	Function of n
Vanilla	2368	1.047^k	1.0218^n
Basic	2384	1.044^k	1.0219^n

The figures for synthetic data show clearly that the algorithm primarily depends on the value of k (Figure 2.6). The effect of choosing larger n values is noticeable (Figure 2.7) but not as large as incrementing k.

We conclude that the basic version does not cause the solver to perform significantly worse. This is an important result from the perspective of the previous implementation that has negatively influenced the solver. In the third chapter we will seek to improve the rule, hoping that the reduced number of recursive steps in the branching algorithm can be translated to more solved instances.

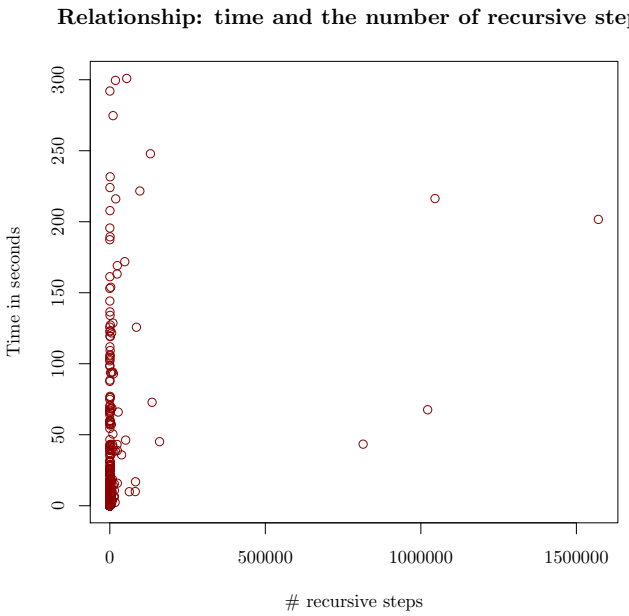

Figure 2.5: Real data: effect of number of recursive steps in the branching algorithm on the running time.

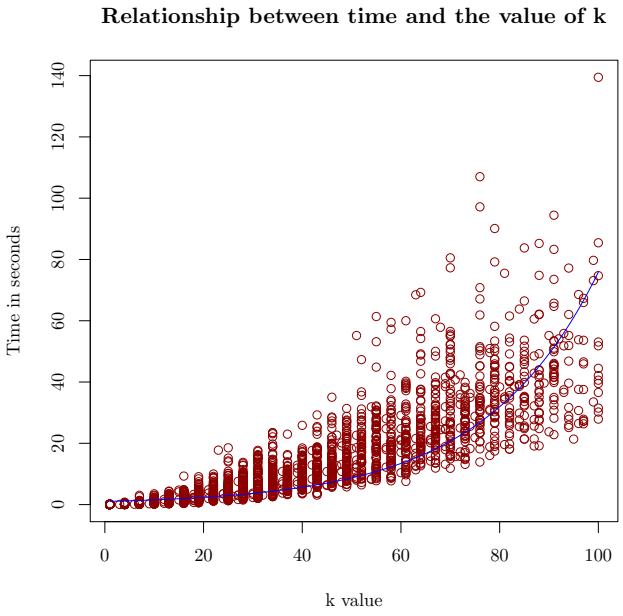

Figure 2.6: Synthetic data: relationship between value of k and time.

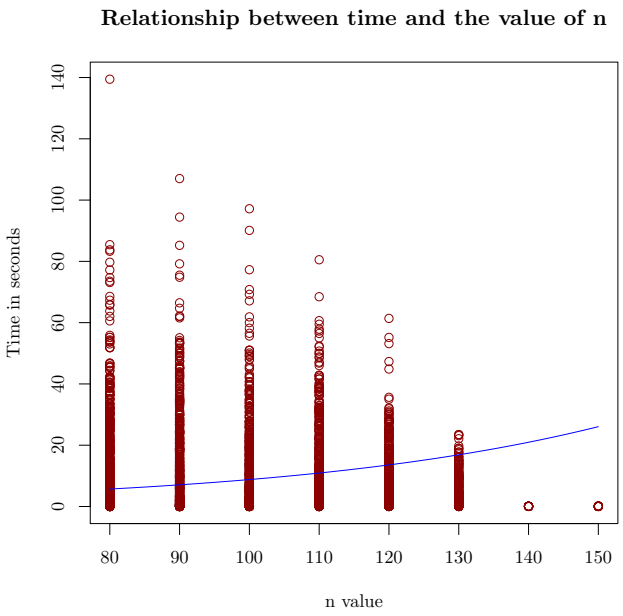

Figure 2.7: Synthetic data: relationship between value of n and time.

3 Extended Version

In this chapter we build on the results from the experiments conducted in the last section of the previous chapter and apply a full algorithm engineering cycle. We aim to theoretically improve our model and design a better version of the critical clique data reduction rule. We also introduce two further rules that take advantage of the structure of critical clique graphs. We do not discuss the implementation, because it is pretty straightforward. We conclude the chapter with further experiments, evaluation of the program and comments on the observed improvements.

3.1 Weaker preconditions for Critical Clique Rule

In this section we analyze and extend the Critical Clique Rule and then prove the correctness of our extensions.

The Critical Clique Rule as described before uses strong bounds on the number of elements in the first and second neighborhoods. We seek to improve the rule by weakening those bounds, i.e. we replace the constant factor t by a calculated value that is possibly smaller (but never larger than t!). For this purpose we employ a *maxDist* function that calculates maximum distance value for the distance function D for any two nodes from two given sets.

Definition 10. *Let* $maxDist_D : \mathcal{P}(V) \times \mathcal{P}(V) \to \mathbb{Z}$ *with* $maxDist_D(A, B) := \max\{D(u, v) : u \in A \wedge v \in B\}$.

We employ this function to weaken the original preconditions for the Critical Clique Rule and thus formulate a new rule, the *Weaker Critical Clique*. Observe that line 3 in Procedure 9 contains the sole modification of the original pseudo-code (Procedure 2 from the second chapter): different bounds on the size of the critical clique.

Reduction Rule 3b (Weaker Critical Clique). *Let $I = (D, X, k)$ be an instance of M-HTC. Apply Procedure 8 to I. Let $I' = (D', X', k')$ be the resulting instance.*

Procedure 8 Weaker Critical Clique

```
1:  procedure WCCRecursive(D, X, t)
2:      WCCC(D, X, t)
3:      Construct G_t^C[X] from D
4:      for all K is an isolated clique in G_t^C[X] do
5:          if D[K] is an ultrametric then
6:              X := X\K
7:          else
8:              WCCRecursive(D,K,t-1)
9:          end if
10:     end for
        return X
11: end procedure
```

Procedure 9 Weaker Critical Clique Construction

```
1:  procedure WCCC(D, X, t)
2:      Construct G_t^C[X] from D
3:      while G_t^C[X] contains a non-isolated critical clique K with
        − K ⊆ X
        − |K| ≥ maxDist_D(K, N(K)) · |N(K)|
        − |K| ≥ maxDist_D(K, N^2(K)) · |N^2(K)| + |N(K)|
        do
4:          for all (u, v) ∈ (N(K) × K\N(K)) do
5:              D(u, v) := t + 1
6:          end for
7:          for all u, v ∈ N(K) with D(u, v) = t + 1 do
8:              D(u, v) := t
9:          end for
10:     end while
11: end procedure
```

Lemma 1. *Weaker Critical Clique Rule (Reduction Rule 3b) is correct.*

To prove Lemma 1, we need to prove three claims. The first claim shows that critical cliques that are large compared to their neighborhood form a cluster in the threshold graph together with a subset of nodes from their first neighborhood.

Claim 2. *Let $I = (D, X, k)$ be an instance of M-HTC, $G_k[X]$ a threshold graph constructed from I, and $K \subseteq X$ a nonisolated critical clique in $G_k[X]$ with $|K| \geq maxDist_D(K, N(K)) \cdot |N(K)|$. Then there exists a closest ultrametric U with a k-Cluster C such that*

(a) $K \subseteq C$ and
(b) $C \subseteq N(K) \cap K$.

Proof of Claim 2. We first prove the *(a)* part of the claim. Our goal is to show that if any K which fulfills the above conditions is split into multiple k-clusters, then those clusters can be joined together giving another closest ultrametric.

Assume towards a contradiction that U' is a closest ultrametric such that $K \not\subseteq C$ for all k-clusters. Consider the set of k-Clusters in U: $C_1, ., ., C_n$. Note that in a nontrivial case $n \geq 2$ and define:

- $K_i := K \cap C_i$,
- $N_i := N(K_i) \cap C_i$, and
- $R_i := C_i \backslash (N_i \cup K_i)$.

We calculate a lower bound on the cost of transforming D into U':

- For splitting K into separate clusters at we "pay" least: $\sum_{i=1}^{n} \sum_{j=i+1}^{n} |K_i \times K_j|$
- For separating K_i from N_j, $j \in \{0, .., n\} \backslash \{i\}$ we "pay" at least: $\sum_{i=1}^{n} \sum_{j=0 \wedge j \neq i}^{n} |K_i \times N_j|$
- For joining K_i with R_i we "pay" at least: $\sum_{i=1}^{n} |K_i \times R_i|$

We now show that any two k-clusters of C can be joined together and the solution will still be optimal. Without loss of generality take two integers i, j with $1 \leq i < j \leq n$ such that $|K_i| \cdot (|R_i| + |N_j| + |K_j|) > |K_i| \cdot (maxDist_D(K, N(K)) \cdot |N_i| + |R_j|)$, which is equivalent to $|R_i| + |N_j| + |K_j| > maxDist_D(K, N(K)) \cdot |N_i| + |R_j|$. Note that the left-hand side of the inequality expresses the cost of splitting K_j from K_i.
Assume towards a contradiction that the opposite is true, that is, $|R_i| + |N_j| + |K_j| \leq maxDist_D(K, N(K)) \cdot |N_i| + |R_j|$.
By summing both sides over all values of i, j we get: $|N(K)| + |K| \leq maxDist_D(K, N(K)) \cdot |N|$, which clearly contradicts $|K| \geq maxDist_D(K, N(K)) \cdot |N(K)|$, proving the assumption that $|R_i| + |N_j| + |K_j| > maxDist_D(K, N(K)) \cdot |N_i| + |R_j|$.

We argue that adding K_i to K_j produces an ultrametric U with smaller distance from D than U', which contradicts the fact that U' is a closest ultrametric. Consider such ultrametic U and the cost of constructing it from U':

- we "pay" at most $|K_i| \cdot ((k - maxDist_D(K, N(K)) + 1)|N_i| + |R_j|)$
- we "save" $|K_i| \cdot (|K_j| + |R_i| + (k - maxDist_D(K, N(K)) + 1)|N_j|) + |K_j| \cdot (|K_i| + |R_j| + (k - maxDist_D(K, N(K)) + 1)|N_i|)$.

It is easy to see that:

$$|K_i| \cdot (|K_j| + |R_i| + (k - maxDist_D(K, N(K)) + 1)|N_j|) \geq$$
$$|K_i| \cdot ((k - maxDist_D(K, N(K)) + 1)|N_i| + |R_j|)$$

because by definition $(k - maxDist_D(K, N(K)) + 1) \leq maxDist_D(K, N(K))$, and as proven above $|R_i| + |N_j| + |K_j| > maxDist_D(K, N(K)) \cdot |N_i| + |R_j|$.

It remains to show that U is an ultrametric. Assume towards a contradiction that U is not an ultrametric. Thus there exists a conflict $Q = \{j, l, m\}$. Since by construction of U only distances between K_i and $X \backslash K_i$ are changed, either $j \in K_i$ or $\{j, l\} \subseteq K_i$. Let $Z = X \backslash K_i \backslash K_j$. We consider only three non-trivial cases:

1. If $j \in K_i$ and $\{m, l\} \subseteq Z$, then it follows that $U(j, m) = U(j, l) = k + 1$ and $U(m, l) \leq k + 1$. This contradicts the fact that Q is a conflict.
2. If $j \in K_j$ and $m \in Z$ and $l \in K_j$, then it follows that $U(j, m) = U(m, l) = k + 1$ and $U(j, l) \leq k$ since K_i and K_j share a cluster. This contradicts the fact that Q is a conflict.
3. The case of $\{j, l\} \subseteq K_i$ and $m \in Z$ is analogous to the first one.

Therefore there are no conflicts in U, which proves our that U is a closest ultrametric to D.

For the *(b)* part of the Claim 2 we consider that each closest ultrametric U has k-cluster C with $K \subseteq C$. Let $A = C \cap N(K)$ and $R = C \backslash (A \cup K)$. We construct an ultrametric U' as follows: for each $u \in K \cup A$ and $v \in R$ we set $U'(u, v) = U'(v, u) = k + 1$. Obviously, U' is an ultrametric, since C is split into two clusters with no conflicts between them. The cost of such construction is equal to splitting A from R and is as most $|R| \cdot (k - maxDist_D(K, N(K)) + 1) \cdot |A|$. In the process we save the cost of joining K with R, which is at least $|R| \cdot |K|$. Since $|K| > maxDist_D(K, N(K))|N|$, we conclude that for every closest ultrametric $C \subseteq N(K) \cup K$.

□

The second claim captures the observation that critical cliques that are large compared to

their neighborhood and their second neighborhood form k-clusters with their first neighborhood.

Claim 3. *Let $I = (D, X, k)$ be an instance of M-HTC, $G_k[X]$ a threshold graph constructed from I, and $K \subseteq X$ a critical clique in $G_k[X]$ with $|K| \geq maxDist_D(K, N(K)) \cdot |N(K)|$ and $|K| \geq maxDist_D(K, N^2(K)) \cdot |N^2(K)| + |N(K)|$. Then $K \cup N(K)$ is a k-cluster in any closest ultrametric U.*

Proof of Claim 3. By Claim 2 we consider a closest ultrametric U' with a k-cluster C with $K \subseteq C \subseteq N(K) \cup K$. The case when $C = N(K) \cup K$ is trivial, therefore we focus on $C \subset N(K) \cup K$.
We construct another ultrametric U by "adding" $B = (N(K) \cup K) \backslash C$ to C:

1. Detach B from $X \backslash (B \cup C)$ by setting $U(u, v) = k + 1$ for all $u \in B$ and $v \in X \backslash (B \cup C)$.
2. Ensure that B forms a k-cluster by setting $U(u, v) = min\{k, U'(u, v)\}$ for all $u, v \in B$.
3. Join B with C by setting $U(u, v) = k$ for all $u \in B$ and $v \in C$.

By definition $B \cup C$ constitute an isolated cluster, and hence U is an ultrametric. To see that U is also a closest ultrametric, consider the cost of the transformation described above: the first operation has the cost with an upper bound of $maxDist_D(K, N^2(K)) \cdot |N^2(K)|$, the cost of the second operation is at most $|N(K)|$. Through the third action we save at least $|K|$. Therefore by the initial condition on the size of K, U is a closest ultrametric. □

Finally, the third claim states that the procedure WCCC, which is called recursively by the reduction rule, is correct.

Claim 4. *Procedure* WCCC *is correct.*

Proof of Claim 4. Let $I = (D, X, k)$ be an instance of M-HTC and $I' = (D', X, k')$ the result of application of WCCC procedure. The procedure iterates over non-isolated critical cliques in the critical clique graph $G_k[X]$ for threshold k. Each critical clique K that fulfills the conditions mentioned in Claim 2 and Claim 3 causes the distance function to be modified in two ways:

1. All cliques from $N(K)$ are joined with K by setting the distance to k.
2. $N(K)$ is detached from $N^2(K)$ by setting the distance to $k + 1$.

By Claim 2 there is an optimal solution such that all neighbors of K must constitute a subset of a k-cluster together with K. Therefore the first type of modification is permitted and the resulting instance is equivalent to I.

By Claim 3 there is an optimal solution such that K forms a k-cluster with its neighbors. Thus the second neighborhood of K must be detached from first neighborhood and therefore the second type of modification is permitted and the resulting instance I' is equivalent to I. □

Proof of Lemma 1. Let $I = (D, X, k)$ be an instance of M-HTC. Let I' be an M-HTC instance which results from application Weaker Critical Clique on I. I' is a YES-instance of M-HTC if and only if I is a YES-instance of M-HTC.

The initial case, when WCCC is applied on X is trivial.

By Claim 4 the procedure WCCC is correct (i.e. all changes applied to the instance produce an equivalent instance) under the assumption that the maximum distance is equal to k. Since the procedure is recursively applied only on isolated critical cliques, this assumption holds true.

Notice that since WCCC is applied only on isolated critical cliques K, there are no external conflicts c with $c \cap K \neq \emptyset$ and $c \cap (X \backslash K) \neq \emptyset$. Given that WCCC is called with $D, K, k-1$, then by its definition the maximum set distance is equal to k which does not introduce any such conflicts c. Therefore the rule is correct.

□

The time complexity of the new rule does not deviate from the original rule. We assume the worst case, that the *maxDist* calculation in the **while** loop of WCCC is implemented with the running time $O(n^2)$. But the loop already contains a $O(n^2)$ calculation as originally stated—joining of the the first neighborhood with the critical clique. Therefore the additional calculation results in a constant factor which can be discarded.

3.2 Lower Bounds on k for Critical Clique

Constructing critical clique graphs gives us an opportunity to formulate two additional reduction rules which aid the Critical Clique. They are not data reductions in a strict

sense, since they do not change the size of the input, but they mark some instances (or branches in the search tree algorithm) as unsolvable and prevent further computation.

Before we introduce two rules, we need to consider the critical clique graph constructed from a M-HTC instance for a given threshold t. It is obvious that the critical clique graph is either a set of disconnected nodes or a graph that can be clustered and therefore can be considered an instance of Cluster Editing. We ignore the former case, since it is trivial, and concentrate on the latter.

The following two lemmas express the relationship between M-HTC and CE in critical clique graphs and are essential to proving correctness of the lower bounds on the value of k.

Lemma 2. *For each P_3 in a Cluster Editing instance constructed from a critical clique graph there exists at least one conflict in the M-HTC instance.*

Proof. Consider a $P_3 = (A, B, C)$ in a critical clique Graph. From the construction of the critical clique Graph it follows that there exists a $P_3' = (a, b, c)$ in the source t-threshold graph $TG_t = (V_{TG_t}, E_{TG_t})$ for which $a \in A$, $b \in B$, $c \in C$ holds. Let u, v with $u \neq v$ and $u, v \in \{a, b, c\}$ and $\{u, v\} \notin E_{TG_t}$. By construction of the t-threshold graph and since (a, b, c) constitutes a P_3' the following holds:

- $D(u, v) > t$,
- For any permutation $\{u', v'\}$ of $\{a, b, c\}$ with $\{u', v'\} \neq \{u, v\}$: $D(u', v') \leq t$

Therefore by definition of a M-HTC conflict, the P_3 (a, b, c) is also an M-HTC conflict. □

Lemma 3. *Each optimal solution to M-HTC must also be a solution to the Cluster Editing Problem in critical clique graph.*

Proof. Proof by contradiction. Assume that an M-HTC solution is smaller than an optimal solution for Cluster Editing Problem in the critical clique graph. Since the M-HTC solution is smaller there is at least one P_3 in Critical Clique Graph that is not covered by it (i.e. no two nodes of P_3 occur in any modification in the M-HTC solution). Since by Lemma 2 for each P_3 there is a conflict in M-HTC, it leads to a contradiction, thus proving the initial claim. □

3.2.1 Number of Non-ultrametrics

The first supplementing rule deals with the number of cliques in Critical Clique Graph that do not induce an ultrametric. We argue that there can be at most k such cliques for the M-HTC instance to be solvable.

Reduction Rule 6. *Let $I = (D, X, k)$ be an instance of M-HTC and $G_k[X]$ critical clique graph for I. Futhermore let X denote the set of critical cliques $C_1...C_n$ that induce an ultrametric, and Y all other critical cliques. If $|X| > k$, then I is a NO instance.*

Lemma 4. *Reduction Rule 6 is correct.*

Proof of Lemma 4. By definition of an induced ultrametric, each clique in X contains at least one internal conflict $Q = \{j, l, m\} \subseteq X$. Therefore there are at least $|X|$ non-overlapping conflicts. This is due to the fact that the cliques in a Critical Clique Graph are disjoint sets of nodes. We note that since there are at most k modifications possible, the instance of CLUSTER EDITING is unsolvable. Thus by Lemma 3 the M-HTC instance is unsolvable. □

3.2.2 Number of Cliques in Connected Components

The second supplementing rule uses the *count of non-isolated cliques* and declares an instance unsolvable if that number is greater than $4k$. Isolated cliques are discarded since they already constitute clusters and are irrelevant to the computation of the solution to CE problem.

Lemma 5. *Let $K_1...K_l$ be clusters resulting from solving the Cluster Editing Problem in a critical clique graph. Denote by Y set of cliques that have been modified, i.e. an incident edge has been added or removed, and by X all others cliques. Then:*

(a) $|Y| \leq 2k$,
(b) $l \leq 2k$, *and*
(c) $|K_i \cap X| \leq 1$ *therefore* $|X| \leq l \leq 2k$.

Proof of Lemma 5. (a) is obvious: since each edge has two endpoints, each modification transfers at most two elements from the set X to set Y.

(b) Assume towards contradiction that a cluster K_i consists only of elements from X (and thus none of Y). Thus, by definition of critical cliques, it must be an isolated cluster. This contradicts the fact that we only consider non-isolated cliques. Therefore each K_i must contain at least one element from Y and as s consequence l has the same upper bound as $|Y|$.

(c) Each cluster K_i contains at most one unmodified clique. This follows directly from the properties of the critical clique graph: two connected cliques X_1, X_2 from X must have the same neighborhoods, otherwise they would need to be modified. Since the neighborhoods of X_1 and X_2 are equal, then by the definition of critical clique graph, they must be one clique. Thus the upper bound for l is also an upper bound for $|X|$. □

Reduction Rule 7. *Let $G := (V, E)$ be the critical clique graph constructed from M-HTC instance $I = (D, X, k)$, let $V_c \subseteq V$ be a set of nodes with at least one neighbor. If $|V_c| > 4k$ then I is a NO-instance.*

Lemma 6. *Reduction Rule 7 is correct.*

Proof of Lemma 6. Consider partitions X, Y of clusters resulting from solving Cluster Editing Problem in critical clique graph as in Lemma 5. Obviously $V_c = X \cup Y$ and since $|X| \leq 2k$ and $|Y| \leq 2k$ it follows that $|V_c| \leq 4k$. Therefore if a node set of non-isolated nodes for CE problem exceeds $4k$ elements, then G is a *NO*-instance of Cluster Editing for given value of k. By Lemma 3 it results that I is a *NO*-instance of M-HTC. □

3.3 Iterative implementation

As mentioned in the description of the Critical Clique rule, the procedure is applied recursively to the threshold graphs. Given the exact nature of the recursion at hand we suspect inefficiencies in this part of the code and therefore try a different, iterative approach.

Another argument in favor of the iterative variant is its simplicity, a feature that is of importance for practical implementations.

The iterative version is very easy. It assumes the existence of the self-maintaining lists of threshold and clique graphs (the maintenance works as described in previous sections).

Then starting with the highest possible threshold it applies the same procedure as in original rule, after that nodes which are not part of an isolated critical clique in current clique graph are removed from all graphs (take note: but not from original instance). This process continues with the next lower threshold value down to the lowest possible value.

The pseudo-code for this version can be found in Procedure 10.

Procedure 10 Iterative implementation of Critical Clique

Require: $|cgraphs| \geq (m-1)$
Require: $|tgraphs| \geq (m-1)$
Require: $m > 0$
procedure CCITERATIVE($m, cgraphs, tgraphs$)
  for $i := (m-1), 1$ **do**
    SIMPLECRITICALCLIQUE($t, cgraphs[i], tgraphs[i]$)
    while $craphs[i]$ contains a non-isolated critical clique K **do**
      for $j := (i-1), 1$ **do**
        Remove all nodes in K from $cgraphs[j]$ and $tgraphs[j]$.
      end for
    end while
  end for
end procedure
procedure SIMPLECRITICALCLIQUE($t, cgraph, tgraph$)
  while $cgraph$ contains a nonisolated critical clique K with
  - $|K| \geq t \cdot |N(K)|$
  - $|K| \geq t \cdot |N^2(K)| + |N(K)|$

  do
    for all $(u, v) \in (N(K) \times K \backslash N(K))$ **do**
      $D(u, v) := t + 1$
    end for
    for all $u, v \in N(K)$ with $D(u, v) = t + 1$ **do**
      $D(u, v) := t$
    end for
  end while
end procedure

We succinctly argue for the correctness without giving a complete proof. It is easy to see that the iterative version of the rule shares the same characteristics with the original rule: instead on recursively branching on isolated critical cliques, it removes non-isolated critical cliques. The original rule ignores them, and the iterative version removes them from the clique and threshold graphs, which achieves essentially the same effect. Since

the isolated critical cliques can be processed independently, it does not matter whether they are processed in a breadth-first or depth-first fashion. Thus the iterative version is correct.

3.4 Experiments

After implementing all of the above theoretical results we proceeded to run the same test suite as with the basic version. We already established that the basic implementation of critical clique reduction rule which maintains clique and threshold graphs does not introduce a significant performance penalty. Our aim was to improve upon this result and make the net result of the application of the rule positive.

For this we compare the extended version (current version) with the vanilla version (program without critical clique rule). The Experiments have been conducted using the same data as in previous chapter. The results for biological data have been summarized in the following table:

Program version	Solved	Total	Function of k	Function of n
Unmodified (vanilla)	2869	3398	1.0304^k	1.118^n
With critical clique (extended)	2869	3398	1.0303^k	1.118^n

Figure 3.1 and Figure 3.2 depict the relationship between the running time and the value of k (resp. the number of nodes). The curve has been calculated using the same method as in the second chapter.

Summarized results for the synthetic data can be found in the following table. Figure 3.6 and Figure 3.7 show the relationship between running time and the value of k (resp. the number of nodes).

Program version	Solved instances	Function of k	Function of n
Unmodified (vanilla)	2368	1.047^k	1.0218^n
With critical clique (extended)	2448	1.041^k	1.0213^n

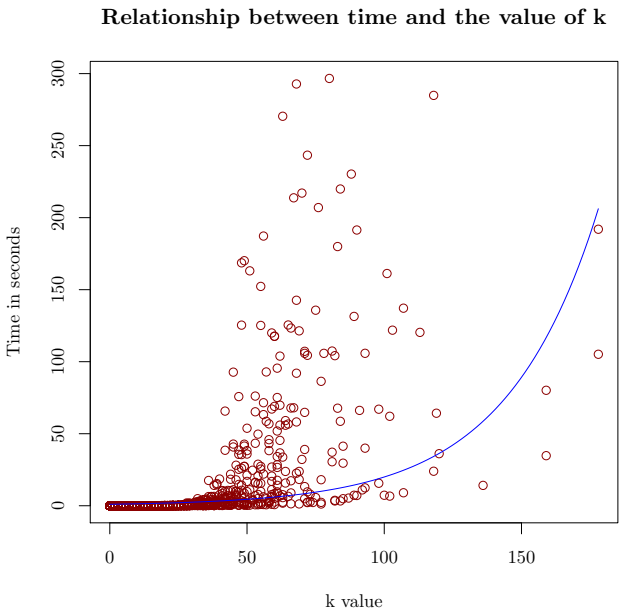

Figure 3.1: Real data: relationship between value of k and time.

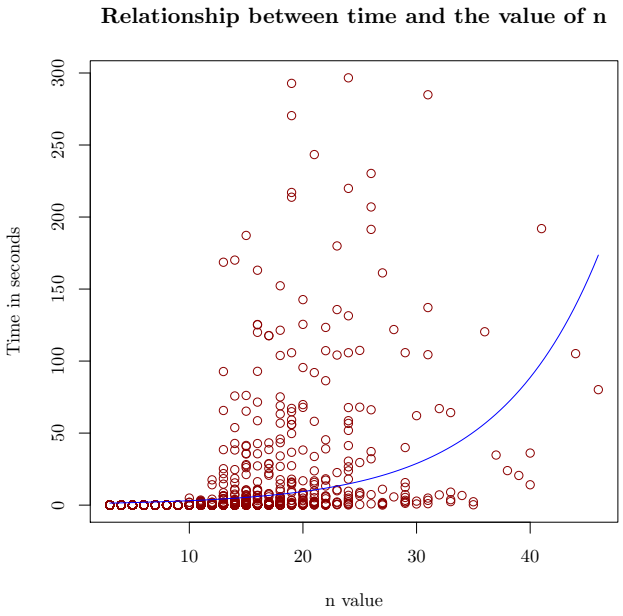

Figure 3.2: Real data: relationship between value of n and time.

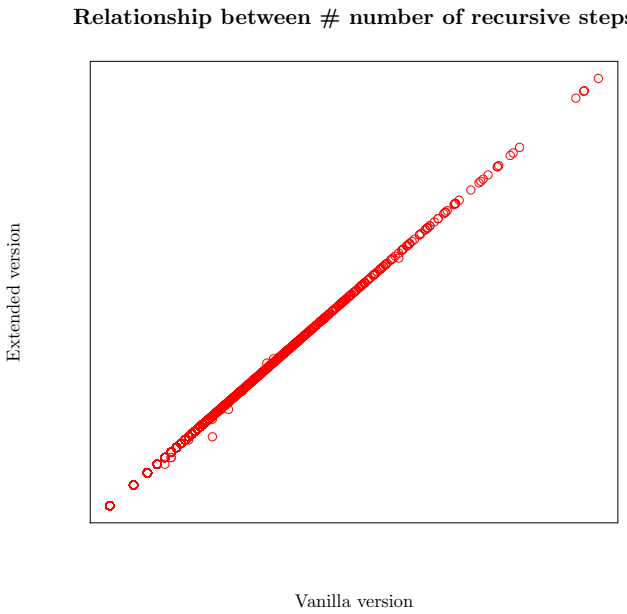

Figure 3.3: Real data: comparing the number of recursive steps needed to solve the instances.

The collected data suggests that even with our improvements to Critical Clique rule, together with two smaller rules does not improve performance with the biological data. Especially the reduction in the number of recursive steps is now almost negligible (Figure 3.4). The ratio between the number of recursive steps in the vanilla version and the extended version of Critical Clique is essentially a flat line, as depicted in Figure 3.3. We observed that this is caused by the lower bound rules on the value of k—which improve the performance of the algorithm but lessen the usefulness of the extended Critical Clique rule.

The results for the synthetic data show 3.3% increase in the number of solved instances. This is an improvement over the basic version of the program of over 2.7%. This effect has been consistent over several runs. The plots showing the relationship between running time and k (resp. the number of nodes) can be seen in Figure 3.6 and Figure 3.7.

The solver does not show any improvement in number of instances for the biological data. We believe that it is due to the fact that solving more than 2869 instances requires much more computation, i.e. the structure of the remaining instances is much more computation-

Figure 3.4: Real data: decrease in the number of recursive steps when employing critical clique reduction rule.

ally intensive and/or resistant to data reduction rules.

Disappointed by the performance with real-world data we analyzed the solver with 20 representative instances (k values in range 30..50) through profiling. The result of this analysis showed that the Reduction Rule 1, which is applied first accounted for over 80% of the running time. We also observed that the rule is powerful enough to solve many instances with very few recursive steps.

Because of these results and recent developments [11] we also conducted an additional, small scale experiment for biological data to measure the effects of the ordering of the rules, and thus possible mutual interactions between them.

We've picked a slightly larger representative set of 50 instances from real world data set and measured the effects on the number of recursive steps and time. We then compared different configurations. The rules are listed by their numbers in order as applied on every second recursive step (where 3basic is the version of critical clique reduction rule from chapter 2 and 3ext is the version from this chapter).

For the third column we calculated mean value of the number of recursive steps for the

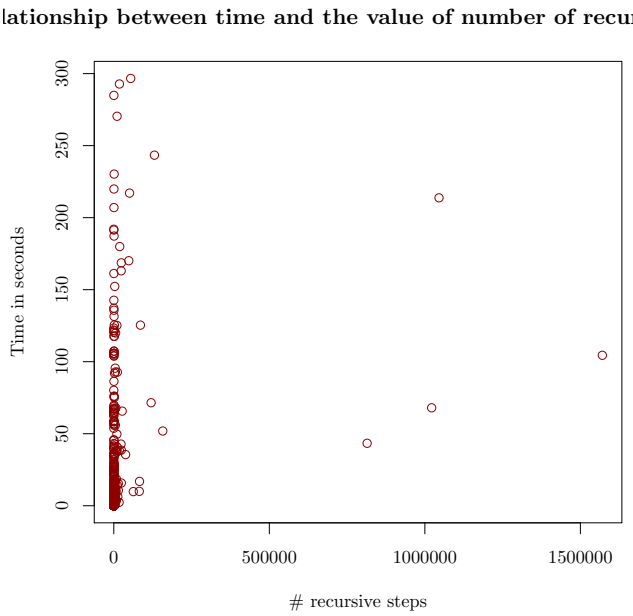

Figure 3.5: Real data: effect of number of recursive steps in the branching algorithm on the running time.

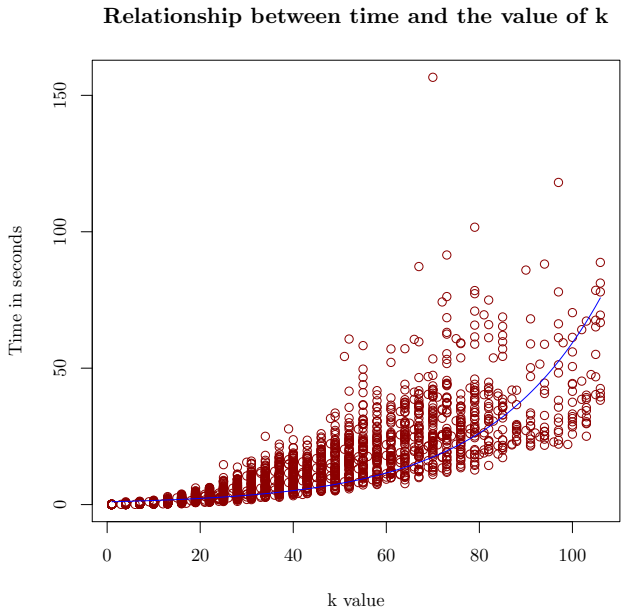

Figure 3.6: Synthetic data: relationship between value of k and time.

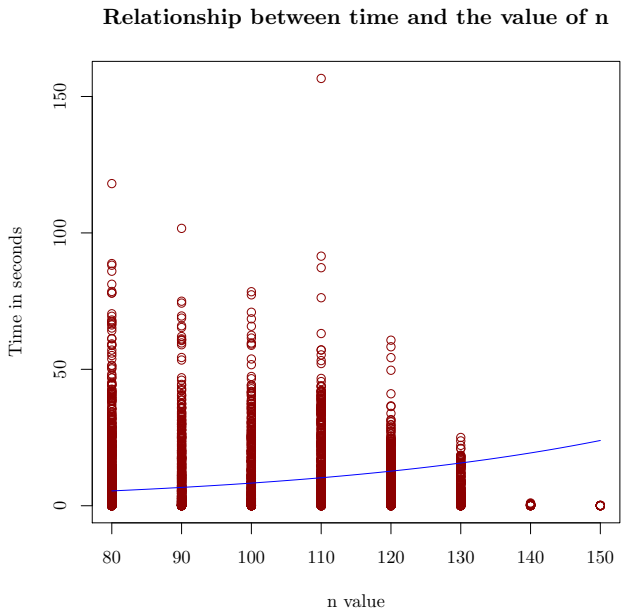

Figure 3.7: Synthetic data: relationship between value of n and time.

two versions and divided them. Values lower than 1 indicate an improvement and values above 1 a decline. For the fourth column we calculated the mean value of the running time for the two versions and divided them. Remarks for the third column do also apply here.

Setup A	Setup B	Recursive steps ratio	Running time ratio
1,2,4,5	1,2,4,5,3basic	0.9729922	0.9647841
1,2,4,5	1,2,4,5,3ext	1.006621	**0.9578519**
1,2,4,5,3ext	3ext,1,2,4,5	1.00079	0.9398267
1,2,4,5,3ext	3ext,2,4,5,1	0.9981938	0.9555669
1,2,4,5	3ext,2,4,5,1	1.0047	0.9173666
1,2,4,5	3ext,1,2,4,5	1.007406	**0.901513**

The first two rows match our previous findings: the basic version reduces the number of recursive steps and (for this subset) reduces the running time when compared with the vanilla version. The extended version minimally increases the number of recursive steps but improves the running time.

We then inspect the effects of moving the extended critical clique to the first position in the list of the applied rules. We note the fact that the values for the same set improve: the solver is on average about 6% faster. After that we apply the most powerful and most time-intensive rule at the end and also notice an improvement in running time, but this time only about 4.5%.

Through application of the extended critical clique rule as the first rule the running time of the solver has been improved by almost 10% on average. Compare this to the initial 4, 2% improvement: it more than doubles the improvement. We conclude that the order of the rules matters in this case, especially since moving Rule 1 to the last position slightly decreases performance.

4 Conclusions and Summary

Throughout this work we employed a methodology based on the Algorithm Engineering Cycle. This proved very beneficial for the overall process: the framework guided our efforts through a challenging process of algorithm development. The initial decision not to rely fully on the methodology has been correct: we gained additional information on the performance of the algorithm by using randomized test data. We would advise to expand test suites when developing/extending algorithms whenever possible.

We also noticed, that the simple 3^k branching algorithm performed really well, but it was because of the multitude of data reductions and fine tuning. While the data reduction rule which we focused on in this work did not bring great improvements, and was largely overshadowed by the Reduction Rule 1, we believe that when pursuing new problems it is the field most worth putting one's effort into.

We also inspected the effects of the order of application of the rules on the solver and found out that by simply changing the order of the rules we were able to double the improvement introduced by the Critical Clique Rule.

We therefore recommend for both practical and theoretical developments in the field of FPT and kernelization to focus more on the mutual interaction of the data reductions rules and benefits thereof. Especially since for practical applications it can be a source of quick-wins.

Bibliography

[1] Nir Ailon and Moses Charikar. Fitting tree metrics: Hierarchical clustering and phylogeny. In *Proc. 46th FOCS*, pages 73–82. IEEE Computer Society, 2005.

[2] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: Ranking and clustering. *Journal of the ACM*, 55(5), 2008.

[3] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1–3):89–113, 2004.

[4] H.D. Benington. Production of large computer programs. *Annals of the History of Computing*, 5(4):350–361, 1983.

[5] Sebastian Böcker, Sebastian Briesemeister, and Gunnar W. Klau. Exact algorithms for cluster editing: Evaluation and experiments. In *Proc. 7th WEA*, volume 5038 of *LNCS*, pages 289–302. Springer, 2008.

[6] B.W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[7] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):59–69, 1993.

[8] Z.Z. Chen, T. Jiang, and G. Lin. Computing phylogenetic roots with bounded degrees and errors. *SIAM Journal on Computing*, 32(4):864–879, 2003.

[9] R.G. Downey and M.R. Fellows. *Parameterized computational feasibility*, volume 92. Citeseer, 1992.

[10] R.G. Downey and M.R. Fellows. *Parameterized Complexity*, volume 3. springer New York, 1999.

[11] H. Ehrig, C. Ermel, F. Hüffner, R. Niedermeier, and O. Runge. Confluence in data reduction: bridging graph transformation and kernelization. *How the World Computes*, pages 193–202, 2012.

[12] V. Filkov and S. Skiena. Heterogeneous data integration with the consensus clustering formalism. In *Data Integration in the Life Sciences*, pages 110–123. Springer, 2004.

[13] J. Guo, S. Hartung, C. Komusiewicz, R. Niedermeier, and J. Uhlmann. Exact algorithms and experiments for hierarchical tree clustering. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI'10)*, pages 457–462, 2010.

[14] Jiong Guo. A more effective linear kernelization for Cluster Editing. *Theoretical Computer Science*, 410(8-10):718–726, 2009.

[15] Jiong Guo and Rolf Niedermeier. Invitation to data reduction and problem kernelization. *ACM SIGACT News*, 38(1):31–45, 2007.

[16] J.A. Hartigan. Statistical theory in clustering. *Journal of Classification*, 2(1):63–76, 1985.

[17] Mirko Křivánek and Jaroslav Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23(3):311–323, 1986.

[18] S. McConnell. *Rapid Development: Taming Wild Software Schedules. 1996*. Redmond, Microsoft Press.

[19] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[20] P. Sanders. Algorithm engineering–an attempt at a definition. *Efficient Algorithms*, pages 321–340, 2009.

[21] Ron Shamir, Roded Sharan, and Dekel Tsur. Cluster graph modification problems. *Discrete Applied Mathematics*, 144(1–2):173–182, 2004.

[22] D.B. West et al. *Introduction to Graph Theory*, volume 2. Prentice hall Upper Saddle River, NJ.:, 2001.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift