universität wien

BACHELORARBEIT

PRACTICAL KERNELIZATION FOR THE EDGE CLIQUE COVER PROBLEM

Verfasser

Jonathan Trummer

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2019

Studienkennzahl lt. Studienblatt:	A 033 521
Fachrichtung:	Informatik - Scientific Computing
Betreuer:	Dipl.-Math. Dipl.-Inform. Dr. Christian Schulz
Co-Betreuer:	Darren Strash, PhD, Hamilton College

Abstract

The edge clique cover problem is the problem of finding a set of complete subgraphs for a given graph, such that every edge of the graph is contained in at least one complete subgraph.

As the edge clique cover problem is NP-hard, computing exact solutions is only feasible for very small graphs. Therefore, heuristics are used to approximate exact solutions as closely as possible. Additionally, reduction rules have been proposed for the edge clique cover problem – which reduce a given graph to a smaller problem kernel, though they have only been combined with exact solvers but not with heuristics.

In this thesis, we combine kernelization and a clique selection heuristic in an attempt to achieve an improvement on the solution quality whilst maintaining a fast running time. We evaluate our algorithm by comparing results of our algorithm with the results of the currently best known heuristic on a wide variety of graphs. Additionally, we evaluate a combination of kernelization and the currently best known heuristic. Our results suggest, that the combined approach can lead to smaller clique covers on many instances and thus better results. Therefore, the combined approach is very promising.

Acknowledgments

I would like to thank my supervisors, Christian and Darren, for their incredible help and guidance throughout my thesis project. Without their ideas and feedback this thesis wouldn't have been possible. I would also like to thank Darren and the Hamilton College team for setting up access to the HPC cluster at Hamilton College. It was a tedious process to get things set up, however, Debby and Steven from Hamilton College were extremely patient and helpful throughout the process.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt und die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht. Weiterhin wurde diese Arbeit keiner anderen Prüfungsbehörde übergeben.

Wien, den 7. Juli 2019

Contents

1 Introduction

An *edge clique cover* is a set $\mathcal{C}$ of complete sub-graphs – called cliques – of a graph such that every edge of the graph is contained in at least one clique in $\mathcal{C}$. Computing a cover of minimum cardinality is known as the *edge clique cover problem*. The edge clique cover problem is *NP-hard* [20, 22], which suggests that a polynomial time exact algorithm is unlikely. The minimum number of cliques covering all edges of a graph is equivalent to the intersection number of a graph [24].

Applications for edge clique covers can be found in diverse fields. When doing compiler optimizations, clique covers are used to help determine which tasks can and cannot be done in parallel. Here, an edge in the complement graph G' connects two operations if they share resources and can therefore not be executed in parallel, a clique C in G' represents a set of operations which can't be executed in parallel and a minimum edge clique cover $\mathcal{C}$ on G' results in an optimal resource assignment [23]. In computational geometry clique covers can be used to store visibility graphs compactly [2]. Further applications can be found in the field of statistics, where multiple-pairwise comparisons are commonly needed [14]. *Letter displays* are used to compactly represent the results of such comparisons, which is equivalent to the edge clique cover problem [14].

1.1 Motivation

As the edge clique cover problem is NP-hard [20, 22], we do not expect an efficient exact algorithm to exist. However, we can apply reductions and kernelization – where we try to reduce a problem to a smaller problem kernel such that a solution on the kernel can be transformed to a solution on the original input in polynomial time.

Reductions and kernelization have been successfully applied to other problems, such as the vertex cover problem [1, 6, 9]. However, the vertex cover problem has a simpler structure than the edge clique cover problem, as each vertex only resides in one of two states: included in the vertex cover or not included in the vertex cover.

Therefore, there are at most 2^n different vertex covers possible, where n denotes the number of vertices. With the clique cover problem, however, every edge can belong to many different cliques, it is not simply a binary problem. Therefore, the number of different possible solutions is dependent on the number of edges, which can be as large as n^2.

Thus, there are more than 2^n possible configurations for the edge clique cover problem.

For this reason, exact solvers are very slow [13] and not feasible for larger problems, as they often employ clique enumeration algorithms. Instead, we wish to define heuristics [4, 7, 15, 19], which approximate the optimal solution as closely as possible whilst computing significantly faster than the exact algorithms. However, heuristics have never been tested alongside the reductions so far.

1.2 Contribution

So far, reduction rules and heuristic approaches for the edge clique cover problem have been investigated separately. In this thesis we build upon the work of Gramm et al. [13, 15], who defined a set of data reduction rules to reduce the edge clique cover problem to a problem kernel. We combine these reduction rules with a heuristic computing a clique likely to be in a small clique cover using local search and a maximal clique heuristic. By selecting a clique likely to be in a small clique cover, we hope that the kernel is modified such that data reduction rules may be applied again. Our results suggest that combining kernelization with a clique selection heuristic can lead to better results on a majority of the tested graphs.

1.3 Structure of Thesis

In Chapter 2 we discuss definitions and notations used throughout this thesis. In Chapter 3 we explore work related to the edge clique cover problem and discuss some of the findings. Chapter 4 gives a detailed description of our algorithm and how the different routines work together. In Chapter 5 we perform experiments on the algorithm defined in this thesis and compare the results against the state of the art heuristic currently available. Additionally, we evaluate a combination of the reduction rules described in Chapter 4 with the currently best known heuristic. Finally, in Chapter 6 we discuss our results and outline some of the future work we wish to do on the topic. Appendix A contains some information about our implementation and Appendix B contains more detailed results of the experiments.

2 Preliminaries

In this chapter we give the definitions used throughout the thesis.

2.1 General Definitions

2.1.1 Graphs

Let $G = (V = \{v_1, v_2, ..., v_n\}, E \subseteq V \times V)$ be an undirected, unweighted, simple graph, where V denotes the set of vertices and E denotes the set of edges, with $n = |V|$ and $m = |E|$. The *neighborhood* of a vertex v is defined as $N(v) = \{u : \{u, v\} \in E\}$ and the *degree* $d(v)$ of the vertex v is defined as $d(v) = |N(v)|$. The *closed neighborhood* $N[v]$ of a vertex v is defined as $N[v] = N(v) \cup \{v\}$. A vertex v is *isolated* if there doesn't exist any vertex $u \in V$ such that $\{u, v\} \in E$.

2.1.2 Cliques and Clique Covers

Given a graph $G = (V, E)$ a *clique* is a set of vertices $C = \{v_1, v_2, ..., v_l\}$ such that $\forall\, v_i \neq v_j \in C : \{v_i, v_j\} \in E$. In words, a clique is a set of vertices, such that every vertex in the clique is adjacent to every other vertex in the clique. An *edge clique cover* of a graph $G = (V, E)$ is a set of cliques $\mathcal{C} = \{C_1, C_2, ...\}$ such that for each edge $\{u, v\} \in E$ there exists at least one clique $C_i \in \mathcal{C}$ such that $\{u, v\} \subseteq C_i$. The uncovered neighborhood $U(v)$ of a vertex v is defined as $U(v) = \{w \mid w \in N(v) \wedge \{w, v\} \notin \mathcal{C}\}$. In words, the *uncovered neighbors* $U(v)$ of a vertex v is a set of vertices $w \in N(v)$ such that the edge $\{w, v\}$ is not contained in any clique C. The *common neighborhood* $N[u, v]$ of two vertices u and v is defined as $N[u, v] = N(u) \cap N(v)$.

2.1.3 Erdős-Rényi Graphs

Erdős-Rényi $G_{n,p}$ graphs [11, 12] are randomly generated graphs. Given a positive integer n and $0 \leq p \leq 1$ a $G_{n,p}$ graph is generated such that the resulting graph consists of n vertices. Vertices are randomly connected by an edge, such that the probability of each edge is equal to p. On average a $G_{n,p}$ graph has $\binom{n}{2}p$ edges.

2.2 Fixed-Parameter Tractability

A problem is *fixed-parameter tractable* if for a given input size n and a parameter k there is an algorithm, which in $\mathcal{O}(f(k)n^{\alpha})$ – with α being a constant independent of n and k – computes a solution to the problem [10, 16]. This means that the complexity of an algorithm is shifted from the input size n to the function $f(k)$. In the case of the edge clique cover problem, given a graph $G = (V, E)$ and a parameter k we wish to find an edge clique cover of G of size k.

2.3 Kernelization and Data Reduction Rules

We wish to reduce a problem to a *problem kernel*. The idea is to take an instance I and apply reduction rules, which reduce the instance I into a smaller instance I', with the size n' of I' being bounded by a function of the parameter k' [10]. To reduce a problem I to a kernel I' – this process is called *kernelization – data reduction rules* are applied. A *data reduction rule* is a mapping ϕ from (n, k) to (n', k'), where ϕ is computable in polynomial time in n and k and $|n'| \leq |n|$ and $k' \leq k$ [17]. A solution on the kernel I' can be transformed to a solution of the original instance I in polynomial time.

3 Related Work

In this chapter we describe all work which is related to the thesis topic at hand.

3.1 Maximal Clique Enumeration

The maximal clique enumeration problem differs from the edge clique cover problem, as with the maximal clique enumeration problem one is interested in finding all cliques which are not properly contained in another clique. However, enumerating all maximal cliques of a graph also results in an edge clique cover for that graph, as the maximal cliques cover all edges of the graph. As our heuristic utilizes a heuristic for finding a maximal clique for the common neighbors of an edge $\{u, v\}$ we would like to give a brief overview of the work done on the maximal clique enumeration problem. Bron and Kerbosch [5] defined two recursive backtracking algorithms to enumerate maximal cliques. Additionally, Tomita et al. [25] proved that the worst-case running time complexity of the algorithm where a *pivot* p is chosen to maximize $|P \setminus N(v)|$ is $\mathcal{O}(3^{n/3})$ for a graph with n vertices .

3.2 Keyword Conflict and NP-completeness

Kellerman [19] defined a method for finding keyword conflicts and a heuristic algorithm for solving a combinatorial problem related to the method. Later, Kou et al. [20] showed not only that the keyword conflict problem is equivalent to edge clique covers, they also described a relationship between edge clique covering and graph coloring problems. Furthermore, Kou et al. [20] improved the Kellerman heuristic by introducing a post-processing step, which ensures that no clique is a subset of the union of subsequent cliques.

Additionally, Kou et al. [20] showed the edge clique cover – and by consequence also the keyword conflict problem – to be NP-hard. Independently, Orlin [22] provided a different proof which came to the same conclusion, namely that the edge clique cover problem is NP-hard.

3.3 Edge Clique Cover

3.3.1 Data Reduction

Though NP-hard problems probably can't be solved exactly efficiently, they can often be reduced to a smaller problem kernel. Problems which can be reduced to a kernel depending on a parameter k are called fixed-parameter tractable problems [16]. Gramm et al. [13, 15] have shown that the edge clique cover problem is reducible to a kernel size of at most 2^k vertices – where k denotes the size of the clique cover – or otherwise does not have a clique cover of size k.

Cygan et al. [8] have shown that the parameterized problem doesn't admit a kernelization with a guarantee of a polynomial output size parameterized by k, where k again denotes the size of the clique cover.

Gramm et al. [13, 15] defined four reduction rules and showed that a graph reduced with respect to Rule 1 and 3 produces a kernel with at most 2^k vertices or else cannot have a solution of size k, where k denotes the size of the clique cover. This thesis builds heavily on these reduction rules.

3.3.2 Exact Algorithms

Gramm et al. [13, 15] proposed a recursive backtracking algorithm for exactly solving the edge clique cover problem. The algorithm chooses an uncovered edge $\{u, v\}$ and enumerates all maximal cliques C that contain $\{u, v\}$. Next, the algorithm branches for every maximal clique in C. This recursively continues until either a solution is found or k cliques have been chosen without finding a clique cover [13, 15].

Unfortunately, the exact algorithm can be very slow. In their paper, Gramm et al. [13] present results for random Erdős-Rényi $G_{n,p}$ graphs. The results show that – depending on the structure of the random graph – even for small graphs with $n = 85$ and $p = 0.15$ the running time varies between 0.01 seconds and 25 minutes.

3.3.3 Heuristics

Gramm et al. [15] additionally proposed an improvement to the Kellerman heuristic [19], which reduces the theoretical running time complexity from $\mathcal{O}(nm^2)$ to $\mathcal{O}(nm)$. Another heuristic for solving the edge clique cover problem is given by Conte, Grossi and Marino [7]. The described heuristic selects uniformly at random an uncovered edge and then adds as many neighbors of the edge as possible to a set, such that the vertices of the set make up a complete subgraph. The proposed algorithm seems to be the state of the art according to their measurements, which compared their algorithm to the improved Kellerman heuristic by Gramm et al. [15]. Conte, Grossi and Marino [7] report their algorithm

to on average find clique covers 5% smaller than using the Gramm et al. [15] heuristic, requiring only about one third of the running time. Conte, Grossi and Marino [7] empirically observed the running time in practice to be close to linear, with the theoretical running time being $\mathcal{O}(\Delta m)$, where Δ denotes the maximum vertex degree.

Behrisch and Taraz [4] provide a different heuristic for covering the edges of a graph with cliques: for k vertices test their common neighborhood for completeness. The largest complete neighborhood is selected as a clique. Behrisch and Taraz [4] either chose k based on prior knowledge of the instances, or – if no prior knowledge existed – set $k \in \{1, 2\}$ and some unspecified values of $k > 2$. This algorithm doesn't guarantee that every edge of the graph is contained in at least one clique and thus doesn't fulfill the edge clique cover definition. Though, as we will see later on, with $k = 2$ this algorithm is equivalent to Rule 2 by Gramm et al. [13, 15].

4 Reduction-based Heuristic

In this chapter we describe the core principles used by our algorithm. We describe the initialization, reductions and post-processing required for the kernelization, as well as the algorithm and post-processing for the heuristic. Our algorithm makes use of the reduction rules defined by Gramm et al. [13, 15] as well as an algorithm for finding a maximal clique within a given common neighborhood $N[u, v]$ of some edge $\{u, v\}$.

4.1 Reduction Rules

We now discuss the reduction rules used by our algorithm. The reduction rules are based on the work of Gramm et al. [13, 15]. For completeness we present the first two reduction rules by Gramm et al., which are the rules we use for our implementation.

In order to make the reduction rules efficient we first perform an initialization step where we construct an auxiliary data structure. In this data structure, for every edge $\{u, v\} \in E$ we store the *common neighborhood* $N[u, v] = N(u) \cap N(v)$ together with a *connectivity number* $c_{u,v}$, which is defined to be

$$c_{u,v} = |\{(x, y) \in E : \forall \{x, y\} \in N[u, v]|.$$

The connectivity number $c_{u,v}$ of an edge $\{u, v\}$ is the number of edges between the vertices of the common neighborhood $N[u, v]$. This lets us compute in constant time, whether the vertices of $N[u, v]$ induce a clique by simply checking if $c_{u,v}$ is equal to the maximum number of possible edges for $N[u, v]$, which is given by $|N[u, v]| * (|N[u, v]| - 1) * 0.5$.

This initialization can be done in $\mathcal{O}(\Delta^2 m)$ time, since for every edge calculating the set of common neighbors $N[u, v]$ takes $\mathcal{O}(\Delta)$ time and calculating $c_{u,v}$ takes $\mathcal{O}(\Delta^2)$ time – where Δ is the maximum degree of G. Doing this for every edge results in $\mathcal{O}(\Delta^2 m)$ time.

Rule 1. *Remove any vertex which is isolated or only connected to covered edges.*

Rule 1 is obviously correct, as every isolated vertex doesn't have any edges which could belong to a clique. Every vertex, which is only connected to covered edges, can be removed as there aren't any further maximal cliques to which this vertex can be added.

Proof. Suppose there is a clique which contains an isolated vertex u. Since $N[u] = \{u\}$ the clique C is $\{u\}$. However, the clique C covers no edges, removing C leads to a smaller clique cover $\mathcal{C}$ and therefore a contradiction.

Similarly, suppose there is a vertex v with $U(v) = \emptyset$. Let $C_1, ..., C_k$ be the cliques that cover the edges incident to v. Then, any other clique C where $v \in C$ can be replaced with $C^* = C \setminus N[v]$ and still be a clique cover of the same cardinality. □

Checking whether Rule 1 applies can be done in $\mathcal{O}(n)$ time. Applying the rule requires an update to the auxiliary data structure created during the initialization. When deleting a vertex w, $N[u, v]$ is updated for all $u, v \in N(w)$ such that $N[u, v] = N[u, v] \setminus \{w\}$. Additionally, $c_{u,v}$ is updated to $c_{u,v} = c_{u,v} - |N[u, v] \cap N(w)|$. These updates can be done in $\mathcal{O}(\Delta)$ time, resulting in an overall running time of $\mathcal{O}(\Delta m)$ for all Rule 1 applications.

Rule 2. *For every uncovered edge contained in exactly one maximal clique, add the edge and its common neighbors $N[u, v]$ to a new clique C and mark the edges $\{w, r\} \in N[u, v] \cup \{u, v\}$ as covered. In other words, if there exists a $N[u, v]$ with $c_{u,v} = |N[u, v]| \times (|N[u, v]| - 1) * 0.5$ add the clique $C = N[u, v] \cup \{u, v\}$ to the clique cover $\mathcal{C}$ and mark the edges of C as covered.*

Every application of Rule 2 introduces exactly one new clique to the clique cover. Exhaustively applying Rule 2 can be done in $\mathcal{O}(m)$ time, as it requires a scan through the auxiliary data structure and performing the $\mathcal{O}(1)$ time calculation as defined in the rule. The algorithm then works as follows: apply Rule 1 and Rule 2, as long as there was a rule application, repeatedly invoke Rule 1 and then Rule 2.

Putting it all together results in a running time of $\mathcal{O}(\Delta^2 m + \Delta m + n) = \mathcal{O}(\Delta^2 m)$. In practice, instead of always scanning through the nodes and the auxiliary data structure to check for possible candidates of the reduction rules we keep a list of candidates for every reduction rule. New candidates for these lists can only occur during initialization and when a reduction rule is applied. During initialization we can in constant time determine candidates for both rules by simply checking if a vertex is isolated (Rule 1) or by comparing the connectivity number to the number of maximal possible edges for $N[u, v]$ (Rule 2). When invoking Rule 1 we can determine candidates for Rule 2 by simply checking the connectivity number of entries affected by the deleted vertex. When invoking Rule 2 we can determine candidates for Rule 1 by looking at the number of uncovered edges for all vertices that are part of the newly formed clique. This is illustrated more detailed in Algorithms 4.2.3, 4.2.4 and 4.2.5

4.2 Our Algorithm

In this section we explain our novel approach. Algorithm 4.2.1 gives a brief overview of the algorithm, in the following subsections we go into more depth of each part of the algorithm.

We start with an empty set of cliques. Next, we apply a pre-processing step to process degree-1 vertices and mark their single edge as a covered clique C. Following the

pre-processing comes the actual reduction. For as long as there are uninitialized edges, initialize the edges of the next vertex and apply the reduction rules until none apply anymore. After all edges are initialized and all reductions are done we do a post-processing step, to ensure correct results. If a given graph should not only be kernelized but solved completely instead, we apply our heuristic on the kernel and add a post-processing step for the heuristic.

Algorithm 4.2.1: ECC: The Reduction-based Heuristic algorithm for the Edge Clique Cover Problem

```
   Input : G = (V, E)
   Output : a clique cover C and a kernel
   // global variables
1  C ⟵ ∅                                   // set of cliques in solution
2  R_1 ⟵ ∅                                 // candidates for Rule 1
3  R_2 ⟵ ∅                                 // candidates for Rule 2
4  PreProcess()
5  while not all edges initialized do
6  |  Initialize()
7  |  repeat
8  |  |  ApplyRuleOne()
9  |  |  ApplyRuleTwo()
10 |  until no rule applied
11 PostProcess()                           // if only the kernelization is required
12 Heuristic()                             // if a complete edge clique cover is required
```

4.2.1 Pre-Processing

The pre-processing step isn't required for running the algorithm, however, it can achieve a small speed-up for some graphs. See Section 5.2 and Table B.2 for an evaluation of the effect of toggling the pre-processing on and off.

The pre-processing iterates over all degree-1 vertices and marks their single edge as a clique. As the single edge of every degree-1 vertex v needs to be a clique, no unnecessary cliques are created here. If the edge were not a clique, the edge wouldn't be covered by any clique at – as the vertices adjacent to the edge do not share any common neighbors – and thus we would not have an edge clique cover. Additionally, as marking an edge as covered can cause other vertices to effectively become degree-1 vertices – in the sense that for a vertex v, $|U(v)| = 1$ – these vertices are also considered, until there are no more degree-1 vertices left.

As mentioned before, the pre-processing step isn't required to ensure a correct result,

Algorithm 4.2.2: PreProcess: pre-process all degree-1 vertices

```
Input : G = (V, E)
Output : a reduced graph, list of covered cliques
1  V ⟵ ∅                                          // set of degree-one vertices
2  for v ∈ V do
3      if |U(v)| = 1 then
4          V ⟵ V ∪ {v}
5  for v ∈ V do
6      C ⟵ {v} ∪ U(v)
7      C ⟵ C ∪ {C}
8      if |U(v)| = 1 then
9          V ⟵ V ∪ {v}
10         V ⟵ V \ {v}
```

however, by eliminating a few vertices before hand we no longer need to consider these vertices in the initialization step, which, as we will see later, is a very costly step.

4.2.2 Initialization

The initialization step is required by the reduction rules, as explained in Section 4.1. In the initialization step we build a lookup table, which, in constant time, lets us determine whether the common neighbors $N[u, v]$ of an edge (u, v) form a clique. This step is very costly, as for every edge (u, v) we need to determine the common neighbors $N[u, v]$, as well as $c_{u,v}$, the number of edges connecting the vertices in $N[u, v]$. As explained in Section 4.1 this step takes $\mathcal{O}(\Delta^2 m)$ time, where Δ denotes the maximum degree of G. However, the constant factor can be reduced significantly. Instead of first finishing the initializations and afterwards doing the reductions, we can initialize the edges of vertices until we find a clique. When a clique is found the initialization is stopped and the reduction rules are applied until none are applicable anymore. Only then do we continue doing the initialization. We call this version of the initialization step *lazy initialization*. With lazy initialization, we reduce the number of edges which need to be initialized. As the initialization only needs to build the common neighborhood $N[u, v]$ and connectivity $c_{u,v}$ for uncovered edges, we can skip edges which are already covered by cliques. This means that if we find a clique C with many vertices early on during the initialization we can skip the initialization for many edges. See Section 5.3 for an experimental evaluation of the running time improvements.

Algorithm 4.2.3: Initialize the lookup data-structure used by the reduction rules

```
Input : G = (V, E)
Output : a lookup data-structure
1  cliqueFound ⟵ False
2  while there are un-initialized vertices in G and cliqueFound is False do
3      for every un-initialized vertex u in G do
4          if u isolated or U(u) = ∅ then
5              R1 ⟵ R1 ∪ {u}
6          for v ∈ U(u) do
7              N[u, v] = N(u) ∩ N(v)
8              c_{u,v} ⟵ number of edges between vertices ∈ N[u, v]
9              if (|N[u, v]| ∗ (|N[u, v]| − 1) ∗ 0.5) = c_{u,v} then
                   // clique found, therefore stop initialization after completing the
                   // initialization for the edges adjacent to the vertex u
10                 R2 ⟵ R2 ∪ {{u, v}}
11                 cliqueFound ⟵ True
12         if cliqueFound is True then
13             return
```

4.2.3 Applying Rule 1

Rule 1 is responsible for deleting vertices which are only adjacent to covered edges. Deleting a vertex requires updates to the lookup table. In particular, the vertex u needs to be removed from the common neighborhoods $N[v, w]$ of all neighbors $v, w \in N(u)$. Additionally, the connectivity $c_{v,w}$ for $N[v, w]$ needs to be adjusted by decrementing the connectivity $c_{v,w}$ by the number of vertices of $N[v, w]$ adjacent to u. Or, more formally, $c_{v,w} = c_{v,w} - |N[v, w] \cap N(u)|$.

As discussed previously, we would like to avoid having to iterate through the lookup table to find candidates for Rule 1 and Rule 2. Instead, we keep a list of candidates. Candidates for Rule 1 can either be generated during initialization or during Rule 2 applications. Candidates for Rule 2 can either be generated during initialization or when entries of the lookup table are altered. An entry for an edge $\{u, v\}$ in the lookup table is a candidate for Rule 2 when $|N[u, v]| * (|N[u, v]| - 1) * 0.5 = c_{u,v}$. Rule 1 is the only place where entries of the lookup table are altered, therefore, after altering an entry in the Rule 1 algorithm, we can simply check if the altered entry is a candidate for Rule 2.

Algorithm 4.2.4: ApplyRuleOne: Rule 1 implementation

```
   Input : Rule One candiates R_1
1  for u ∈ R_1 do
2      for v ∈ N(u) do
3          for w ≠ v ∈ N(u) do
4              N[v, w] ⟵ N[v, w] \ {u}
5              c_{v,w} ⟵ c_{v,w} − |N[v, w] ∩ N(u)|
6              if c_{v,w} = (|N[v, w]| ∗ (|N[v, w]| − 1) ∗ 0.5) then
7                  R_2 ⟵ R_2 ∪ {{v, w}}
8          remove N[u, v] and c_{u,v} from the lookup table
9      V ⟵ V \ {u}
```

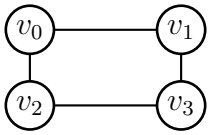
4.2.4 Applying Rule 2

Rule 2 is responsible for finding cliques in the lookup table and marking the edges of these cliques as covered. By marking an edge $\{u, v\}$ as covered we need to remove the entries for the common neighborhood $N[u, v]$ of $\{u, v\}$ and the connectivity $c_{u,v}$ from the lookup table. Failing to do so could result in non-optimal results. If an edge $\{u, v\}$ is contained in a maximal clique, creating a new clique $\{u, v\} \cup N[u, v]$ may result in a non-optimal clique and therefore a non-minimum clique cover. In Section 4.2.3 we have shown that Rule 1 generates candidates for Rule 2. Similarly, Rule 1 candidates are generated during invocations of Rule 2. A vertex is a candidate for Rule 1 when it is either isolated or only adjacent to covered edges. Rule 2 is the only place where edges are marked as covered. Therefore, after marking all edges of a clique as covered, we can simply check for all vertices $v \in C$ if $U(v) = \emptyset$. If there are such vertices, they are candidates for Rule 1

Algorithm 4.2.5: ApplyRuleTwo: Rule 2 implementation

```
1  {u, v} ⟵ extract candidate from R_2
2  C ⟵ {u, v} ∪ N[u, v]
3  for r ∈ C do
4      for s ≠ r ∈ C do
5          mark {r, s} as covered
6          remove N[r, s] and c_{r,s} from lookup table
7      if U(r) = ∅ then
8          R_1 ⟵ R_1 ∪ {r}
9  C ⟵ C ∪ {C}
```

Figure 4.1: A graph consisting of 4 vertices and 4 edges. Applying the reduction rules once leads to a clique being reduced, however it doesn't lead to any vertex being deleted, as $\forall v \in V : d(v) = 2$ and every clique contains exactly one edge. Thus for no vertex v is $U(v) = \emptyset$ after one application of the reduction rules and therefore Rule 1 can't apply.



4.2.5 Post-Processing

Gramm et al. [13, 15] claim that every application of Rule 2 results in at least one new candidate for Rule 1. Which means that after an application of Rule 2 – which reduces a new clique C – there is some $v \in C$ such that $U(v) = \emptyset$. Unfortunately, this is not true, which can be shown very easily.

Consider a square where the corners represent four vertices, as in Figure 4.1. Applying Rule 2 to such a graph once doesn't result in any vertex being adjacent to only covered edges and therefore being deleted. Rule 2 will reduce one of the following edges $\{v_0, v_1\}$, $\{v_1, v_2\}$, $\{v_2, v_3\}$ or $\{v_3, v_0\}$. This leads to $\forall v \in V : |U(v)| \in \{1, 2\}$ and therefore $\nexists v \in V : U(v) = \emptyset$. Note that this post-processing is not required if a heuristic run on the kernel is aware of which edges are covered and which edges are uncovered – as is the case with our heuristic, which utilizes the data structures of the kernelization. However, if another algorithm operates on the output of the kernelization, it is not aware of which edges were covered, yet kept in the kernel. We note that the final kernel contains edges that may be covered by some already computed clique(s).

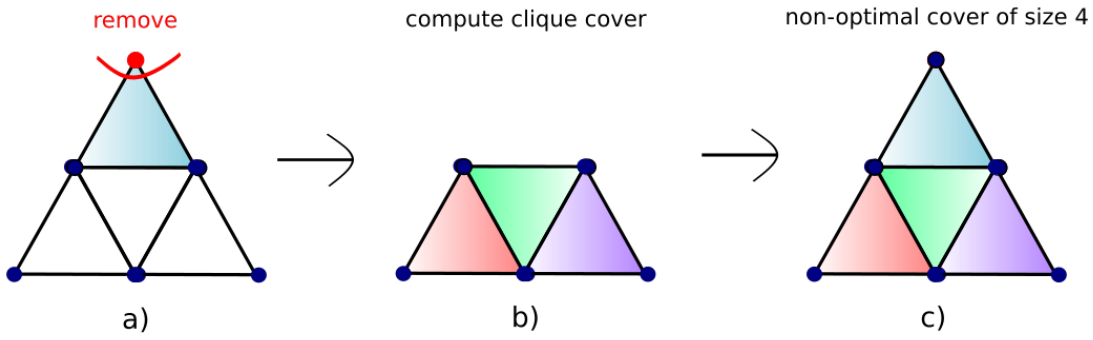
The post-processing step ensures that all cliques with only non-deleted vertices are removed from the clique cover $\mathcal{C}$. This is done by simply iterating through cliques and removing any clique where all vertices are present in the kernel.

In practice, this step can be sped up significantly by making use of the fact that Algorithm 4.2.5 checks for Rule 1 candidates. Whenever Algorithm 4.2.5 for a clique C finds a candidate for Rule 1 we already know that there's at least one vertex within C, which will be deleted, and therefore C will remain in clique cover $\mathcal{C}$. However, when Algorithm 4.2.5 doesn't find any Rule 1 candidates for the vertices of a clique, we need to check C during the post-processing step.

In practice, using this differentiation between cliques that are definitely kept in the solution $\mathcal{C}$ and those which may be removed, speeds up the post-processing step significantly, as the number of cliques which need to be checked is reduced immensely. The theoretical running time complexity is $\mathcal{O}(\Delta m)$, as there are at most m cliques and for every clique we need to check for at most Δ vertices if that vertex is deleted.

We note, that the post-processing is not optimal, as is illustrated in Figure 4.2. Assume

Figure 4.2: Illustration of why the post-processing step is not optimal.



that in step a) the kernelization reduces the highlighted triangle as a clique, as a result, the highlighted vertex is deleted. If we now consider the remaining graph to be the kernel, another algorithm run on the kernel will in step b) reduce the three highlighted cliques. As we can clearly see in step c), this leads to a non-optimal clique cover, as the green triangle is redundant. Therefore, even with the post-processing step in place, running an algorithm on the kernel may skew the results and lead to non-optimal clique covers. As our algorithm operates on the data structures created by the reduction rules, we do not have this problem. However, when the kernel is explicitly stored and fed into another program, this issue will occur.

Algorithm 4.2.6: PostProcessing: remove all cliques with only non-deleted vertices

```
1 for C ∈ 𝒞 do
2     if only non-deleted vertices in C then
3         𝒞 ⟵ 𝒞 \ {C}
```

4.2.6 A New Heuristic

Our newly developed heuristic for solving the edge clique cover problem makes use of the data structures and reduction rules described in Section 4.1. After the reduction rules finish reducing the input graph into a kernel, the heuristic selects a clique in the kernel to cover, so that the reduction rules may be applicable again. This is done by computing a maximal clique covering a single edge and as many of its common neighbors as possible and then updating the data structures. For finding a maximal clique a heuristic is used, which, given a lookup table entry $\{u, v\}$, tries to find a maximal clique within the common neighborhood $N[u, v]$. At the beginning of the edge clique cover heuristic we sort all remaining entries

of the lookup table based on the size of their common neighborhood in descending order. Whenever we need to apply the maximal clique finding heuristic, we take the next entry $\{u, v\}$ of this sorted set and try to find a maximal clique in the common neighborhood $N[u, v]$. As we wish to achieve a small clique cover, we perform a post-processing step, where redundant cliques are removed.

Algorithm 4.2.7: Heuristic

```
1  C_heuristic ⟵ ∅                          // maximal cliques found by the heuristic
2  Q ⟵ {u, v} sorted by |N[u, v]| in descending order
3  while not all edges covered do
4      extract next {u, v} from Q
5      C ⟵ FindMaximalClique({u, v})
6      C_heuristic ⟵ C_heuristic ∪ {C}
       // mark edges of clique as covered, same as in Algorithm 4.2.5
7      for r ∈ C do
8          for s ≠ r ∈ C do
9              mark {r, s} as covered
10             remove N[r, s] and c_{r,s} from lookup table
11         if U(r) = ∅ then
12             R_1 ⟵ R_1 ∪ {r}
       // repeatedly apply the rules, same as in Algorithm 4.2.1
13     repeat
14         ApplyRuleOne()
15         ApplyRuleTwo()
16     until no rule applied
17 HeuristicPostProcessing()
```

The heuristic for finding a maximal clique is given an entry for an edge $\{u, v\}$ from the lookup table. For every vertex $w \in N[u, v]$ we determine $e_w = |N(w) \cap N[u, v]|$. The vertices $w \in N[u, v]$ are sorted by e_w in ascending order. This means that for every vertex $w \in N[u, v]$ we know to how many vertices $r \in N[u, v]$ the vertex w is adjacent to. Next, we repeatedly remove a vertex w from $N[u, v]$ such that $e_w = \min_{\forall r \in N[u,v]} \{e_r\}$ and update e_r for all $r \in N[u, v] \cap N(w)$ by decrementing the values by one. Additionally, the connectivity $c_{u,v}$ is updated to be $c_{u,v} \longleftarrow c_{u,v} - |N(w) \cap N[u, v]|$. We remove the vertex w with minimum e_w, as vertices w with e_w smaller than e_r for $r \neq w \in N[u, v]$ are less likely to be in a maximal clique, due to being adjacent to fewer vertices $r \in N[u, v]$. We repeatedly remove the vertex w with minimum e_w until $|N[u, v]| * (|N[u, v]| - 1) * 0.5 = c_{u,v}$ and thus the vertices $\{u, v\} \cup N[u, v]$ now form a clique C.

One iteration of this algorithm is of complexity $\mathcal{O}(\Delta^2 \log \Delta)$. Calculating all e_w re-

quires $\mathcal{O}(\Delta^2)$ time, sorting the vertices $w \in N[u,v]$ takes $\mathcal{O}(\Delta \log \Delta)$ time. Removing w from $N[u,v]$ and updating $c_{u,v}$ can be done in constant $\mathcal{O}(1)$ time, however, re-sorting the vertices $w \in N[u,v]$ takes $\mathcal{O}(\Delta \log \Delta)$ time and decrementing e_r for $r \neq w \in N[u,v]$ takes $\mathcal{O}(\Delta)$ time. As the while-loop will run at most $\mathcal{O}(\Delta)$ times, this results in an overall running time of $\mathcal{O}(\Delta^2 \log \Delta)$. This algorithm will be run at most m times – once on every edge, therefore the algorithm takes at most $\mathcal{O}(m\Delta^2 \log \Delta)$ time.

Algorithm 4.2.8: FindMaximalClique

```
   Input : an edge {u, v}
   Output : a clique C
 1 for w ∈ N[u, v] do
 2     e_w ⟵ |N(w) ∩ N[u, v]|
 3 sort vertices w ∈ N[u, v] by e_w in ascending order
 4 while c_{u,v} ≤ (|N[u, v]| ∗ (|N[u, v]| − 1) ∗ 0.5) do
 5     w ⟵ next vertex from N[u, v]
 6     c_{u,v} ⟵ c_{u,v} − |e_w|
 7     N[u, v] ⟵ N[u, v] \ {w}
 8     for r ∈ N(w) ∩ N[u, v] do
 9         e_r ⟵ e_r − 1
10     re-sort vertices r ∈ N[u, v] by e_r in ascending order
11 C ⟵ {u, v} ∪ N[u, v]
12 return C
```

The heuristic requires a different post-processing than the kernelization. Furthermore, as the heuristic immediately follows the kernelization and utilizes the data structures built by the kernelization, the post-processing of the kernelization can be omitted. As the heuristic leads to all vertices being deleted, the post-processing of the kernelization doesn't need to clean-up cliques with only non-deleted vertices. Instead, all of those cliques can remain in the solution.

However, there may be many cliques $C \in \mathcal{C}_{heuristic}$ – where $\mathcal{C}_{heuristic}$ denotes the cliques found by the maximal clique heuristic – which turn out to be redundant, as all edges covered by C are additionally covered by other cliques. Therefore, to remove redundant cliques, we perform a slightly modified version of the post-processing step defined by Kou et al. [20]. Their post-processing removes any cliques which are a subset of the union of all subsequent cliques. In our post-processing, we remove any clique from $\mathcal{C}_{heuristic}$ if all of its edges are covered by subsequent maximal cliques $C \in \mathcal{C}_{heuristic}$ or by the cliques $C \in \mathcal{C}_{reductions}$, where $\mathcal{C}_{reductions}$ denotes the set of cliques found by the reduction rules. Formally, a clique $C \in \mathcal{C}_{heuristic}$ is redundant, when $C \subset \bigcup C_i \in \mathcal{C}_{reductions} \cup \mathcal{C}_{heuristic} \setminus \{C\}$. To perform the post-processing we initialize a data structure, where for every edge $\{u,v\}$ we store the number of cliques in which that edge is contained as $n_{u,v}$. This is done for all cliques in

$\mathcal{C}_{reductions}$ as well as the cliques in $\mathcal{C}_{heuristic}$. Next, we iterate over the cliques in $\mathcal{C}_{heuristic}$, and check the counters $n_{u,v}$ for every edge $\{u, v\}$ in a clique C. If and only if at least one of these counters $n_{u,v}$ is set to 1 the clique C is added to the solution $\mathcal{C}$, as it is the only clique, which covers the edge $\{u, v\}$. If for no edge $\{u, v\}$ of a clique C the counter $n_{u,v} = 1$ the clique C is redundant. Therefore, we decrement $n_{u,v}$ by one for every $\{u, v\} \subseteq C$ and discard the clique C.

Claim 1. *This post-processing is optimal in that it removes any redundancy in the cliques $C \in \mathcal{C}_{heuristic}$ such that for all cliques $C \in \mathcal{C}$ – where $\mathcal{C}$ denotes the solution – $C \nsubseteq \bigcup C_i \in \mathcal{C} \setminus \{C\}$.*

Proof. Assume that there's a clique C in the solution $\mathcal{C}$ which is redundant. This implies that for all edges $\{u, v\} \subseteq C$: $n_{u,v} > 1$. By the criteria in Algorithm 4.2.9 at Line 7, the clique C isn't added to the solution $\mathcal{C}$, as it requires an edge $\{u, v\} \subseteq C$ such that $n_{u,v} = 1$. This leads to a contradiction, therefore C is not added to the solution $\mathcal{C}$. □

It is sufficient to only check the cliques $C \in \mathcal{C}_{heuristic}$ for redundancy, as the cliques $C \in \mathcal{C}_{reductions}$ aren't redundant, due to how Rule 2 is defined. As Rule 2 marks a clique C as covered, if there's an edge $\{u, v\} \in C$ that belongs to exactly one maximal clique, there can't be another clique C' which covers $\{u, v\}$ as well.

Algorithm 4.2.9: HeuristicPostProcessing

```
   Input : C_reductions and C_heuristic
   Output : C such that C is an edge clique cover
1  C ⟵ C_reductions
2  N ⟵ ∅
3  for C ∈ C ∪ C_heuristic do
4  |   for {u, v} ⊆ C do
5  |   |   n_{u,v} ⟵ n_{u,v} + 1
6  for C ∈ C_heuristic do
7  |   if ∃{u, v} ⊆ C : n_{u,v} = 1 then
8  |   |   C ⟵ C ∪ {C}
9  |   else
   |   |   // clique is redundant
10 |   |   for {u, v} ⊆ C do
11 |   |   |   n_{u,v} ⟵ n_{u,v} − 1
12 return C
```

5 Experimental Evaluation

In this chapter we benchmark the algorithm defined in Chapter 4 – which we denote by our – against the currently best known heuristic by Conte, Grossi and Marino [7], which we will from now on reference to as cgm. Additionally, we compare cgm against the reduction rules described in Section 4.1 combined with cgm run on the kernel. We refer to this combination of the reduction rules and cgm as kernel+cgm.

We first evaluate the effects of the pre-processing described in Section 4.2.1 on the running time. Next, we evaluate the speed-up we achieve by using lazy initialization (as described in Section 4.2.2). Then, we benchmark kernel+cgm against cgm. And finally, we benchmark our algorithm – our – against cgm.

5.1 Experimental Setup

5.1.1 Environment

All experiments where run on a machine powered by an Intel ® Xeon ® X5650 processor with six cores and 2.67 GHz per core. Although the CPU has multiple cores, the experiments were run on a single core. The machine has 20GB of memory and is running on CentOS 7 with Linux Kernel version 3.10.0-957.1.3.el7.x86_64. cgm was run using openjdk version `1.8.0_191`, our implementation was compiled with g++ 4.8.5 and optimization flag `-O3`.

5.1.2 Methodology

cgm uses a random choice to determine the next edge to be expanded [7]. This randomness can cause fluctuations in the results, given different random seeds. Therefore, we run the algorithms ten times on the graphs to get representative results. When running kernel+cgm, we run cgm five times on the kernel and pick the best result in terms of minimum clique cover size whilst summing up all running times. We run kernel+cgm ten times on the graphs and average the results.

We evaluate the following metrics:

- clique cover size
- maximum clique size

- average clique size
- running time

The clique cover size is the number of cliques such that all edges of the given graph are covered by at least one clique, the maximum clique size is the largest clique found and the average clique size is the average over all cliques in the clique cover.

Though we strive to achieve the smallest clique cover possible, we take the average clique cover number over all runs, in order to guarantee a fair comparison of the heuristics. Secondary – though nonetheless still interesting – parameters are the average and maximum clique size. For the maximum clique size and the average clique size we take the average over all runs. Similarly, the running time is averaged over all runs as well. For the running times we use the running times as reported by the implementations, which don't include the time required for reading in the graph file and writing results. When running cgm on a kernel, we take the result with the minimum clique cover and sum up all running times. For the averages across runs on the same graph we use the arithmetic mean, defined as $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$. We use the geometric mean – defined as $\bar{x} = (\prod_{i=1}^{n} x_i)^{(\frac{1}{n})}$ – over values across multiple data sets (i.e. average clique sizes across all data sets, maximum clique sizes across all data sets, etc.).

5.1.3 Instances

The instances used for evaluation of the algorithms were taken from the 10th DIMACS Implementation Challenge [3] and from the Stanford Large Network Dataset Collection (SNAP) [21]. Additionally, we generated $G_{n,p}$ Erdős-Rényi graphs [11, 12] with $n \in \{100, 1000, 3000\}$ and $p = 0.1$. All graph instances used are listed in Table 5.1.

The instances `astro-ph`, `cond-mat-2003`, `cond-mat-2005`, `citationCiteseer`, `coAuthorsDBLP`, `coPapersCiteseer` and `coPapersDBLP` [3] as well as `ca-HepPh` [21] represent collaborations between authors and papers in scientific research. The graph `as-22july06` [3] represents a snapshot of the structure of the internet at the level of autonomous systems, `as-Skitter` [21] represents the results of many traceroutes from several scattered sources to million destinations and `caidaRouterLevel` [3] represents measurements of the adjacency matrix of the Internet router-level graph.

The instances `eu-2005` and `in-2004` [3] represent small webcrawls of the top-level domains *eu* and *in*, the instance `cnr-2000` represents a webcrawl of the domain *cnr.it* [3]. The instance `amazon0601` [21] represents a co-purchasing graph from Amazon.com Inc. Lastly, the instances `deezer_ro`, `deezer_hr`, `com-youtube`, `soc-pokec-relationships` and `soc-LiveJournal1` [21] represent social networks. We chose these instances, as they provide a range of different graph sizes, ranging from small instances, such as the $G_{n,p}$ graphs and `as-22july06`, to large instances, such as `soc-LiveJournal1`.

Table 5.1: Graph instances used for evaluation, grouped by the source we got the graph from and sorted by n

source	graph	n	m
Erdős-Rényi graphs	$G_{n=100,p=0.1}$	100	488
	$G_{n=1000,p=0.1}$	1 000	49 576
	$G_{n=3000,p=0.1}$	3 000	450 085
DIMACS	astro-ph	16 706	121 251
	as-22july06	22 963	48 436
	cond-mat-2003	31 163	120 029
	cond-mat-2005	40 421	175 691
	caidaRouterLevel	192 244	1 218 132
	citationCiteseer	268 495	2 313 294
	coAuthorsDBLP	299 067	977 676
	cnr-2000	325 557	2 738 969
	coPapersCiteseer	434 102	16 036 720
	coPapersDBLP	540 486	15 245 729
	eu-2005	862 664	16 138 468
	in-2004	1 382 908	13 591 473
SNAP	ca-HepPh	12 006	118 489
	deezer_ro	41 773	125 826
	deezer_hr	54 573	498 202
	amazon0601	403 394	2 443 408
	com-youtube	1 134 890	2 987 624
	soc-pokec-relationships	1 632 803	22 301 964
	as-Skitter	1 696 415	11 095 298
	soc-LiveJournal1	4 846 609	42 851 237

5.2 Evaluation of Pre-Processing

Here, we experimentally evaluate the effects of the pre-processing step from Section 4.2.2 on the running time. We ran the algorithm with the pre-processing turned on and off on all instances listed in Table 5.1 and compared the results. The running time was averaged over five runs.

As we can see in Figure 5.1, turning on the pre-processing has a positive effect on almost all instances, except on the six instances citationCiteseer, coPapersDBLP, as-Skitter, soc-LiveJournal1, soc-pokec-relationships and $G_{n=100,p=0.1}$. If we look at the running time in Table B.2, we can see that for the instances citationCiteseer, coPapersDBLP and $G_{n=100,p=0.1}$ the difference in running time

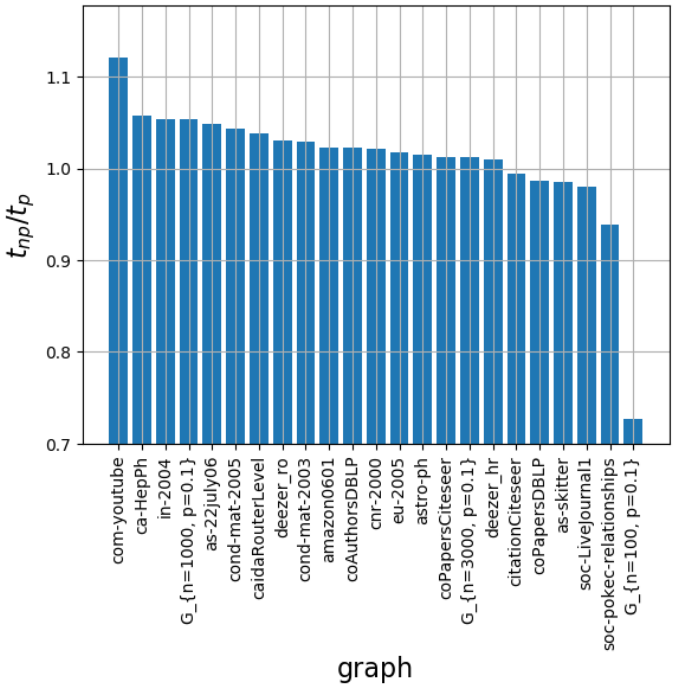
is very small and therefore negligible.

The pre-processing has a negative effect on the running time, when the overhead of checking for degree-1 vertices outweighs the gain made by marking the edges as cliques. In theory, this should be visible in the instances $G_{n,p}$ with $n \in \{100, 1\,000, 3\,000\}$, as the pre-processing is unable to reduce any cliques at all. However, for $G_{n,p}$ graphs with $n \in \{1\,000, 3\,000\}$ the running time is actually slightly reduced, though as we can see in Table B.2, the difference in running time is very small on both instances and may be due to some measurement error or some caching phenomenon.

Overall, the geometric mean of 5.867 seconds over the running times with pre-processing enabled versus 5.906 seconds with the pre-processing disabled suggests, that the pre-processing overall has a slightly positive effect on the running time.

If prior knowledge regarding the number of degree-1 vertices exists, the pre-processing can be toggled based on that knowledge, as a graph with a high ratio of degree-1 vertices is more likely to benefit from the pre-processing, whereas graphs with no degree-1 vertices are more likely to be slightly slowed down by the pre-processing.

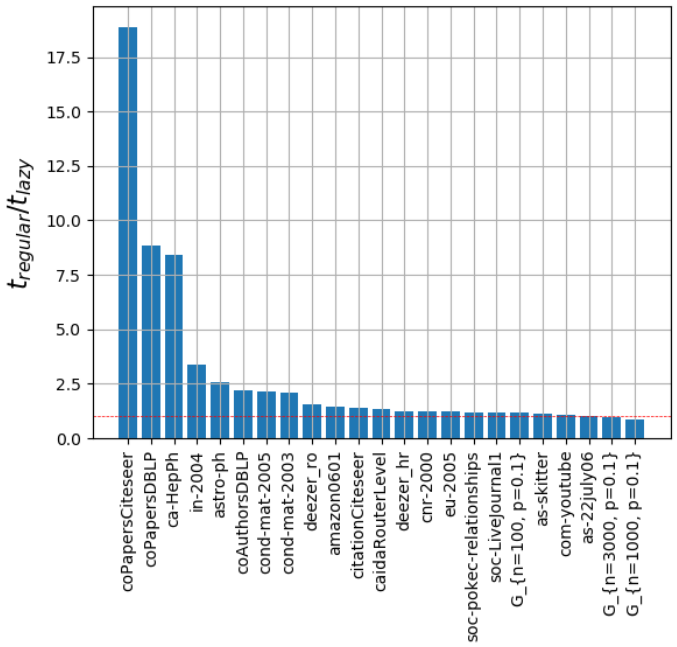
Figure 5.1: Speedup when using preprocessing (t_p) versus without preprocessing (t_{np}). See Table B.2 for more detailed results.



5.3 Evaluation of Lazy Initialization

As described in Section 4.2.3 we perform a lazy initialization (instead of the regular initialization), where we interrupt the initialization to reduce cliques as soon as we find them. In this section we present experimental data on the speedup of the lazy initialization over the regular initialization.

Figure 5.2: Speedup when using lazy initialization (t_{lazy}) instead of regular initialization ($t_{regular}$). Red dashed line marks $y = 1.0$. See Table B.3 for more detailed results.



As we can see in Figure 5.2 the lazy initialization leads to an improvement on almost all instances. Exceptions are `as-22july06`, $G_{n=3000,p=0.1}$ and $G_{n=100,p=0.1}$. However, the difference in those running times is negligible, as can be seen in Table B.3.

The geometric mean of 6.137 seconds over the running times with lazy initialization versus the geometric mean of 11.433 seconds with regular initialization clearly shows the significant advantage lazy initialization has over regular initialization.

The speedup is especially valuable on the instances `eu-2005`, `in-2004` and `soc-LiveJournal1`, as the running time on these instances is fairly slow, which can be seen in Table B.11. The running time for the graph `eu-2005` is reduced from $2\,481$ seconds to $2\,020$ seconds and for the graph `soc-LiveJournal1` the running time is reduced from $1\,926$ seconds to $1\,634$ seconds.

In absolute numbers, the speedup on the instance `in-2004` is largest, with the running time being reduced from $2\,130$ seconds to 632 seconds (about 25 minutes faster).

The speedup occurs, as early on during the initialization many cliques are found and thus eliminate the need to initialize the covered edges. For example, on coPapersCiteseer a clique of size 845 is found early on during the lazy initialization. A clique of size 845 contains $356\,590$ edges, due to the lazy initialization lookup table entries are created only for a small fraction of these edges. When multiple large cliques are reduced during the lazy initialization, the number of edges, for which a lookup table entry needs to be created, is reduced significantly. As the initialization phase is of complexity $\mathcal{O}(\Delta^2 m)$, this has a significant impact on some of these graphs.

On the $G_{n,p}$ graphs $n \in \{1\,000, 3\,000\}$ most of the work is done by the heuristic rather than by the reduction rules, therefore the lazy initialization barely has an advantage over regular initialization. On the $G_{n,p}$ graph with $n = 100$ the reduction rules fully reduce the graph, without the need of the heuristic. However, as the graph is very small, the difference between lazy and regular initialization is very small as well.

Caching may be a factor here as well, as when during initialization a lookup table entry is created which forms a clique, that entry may still reside in cache when the clique is reduced. As access to entries in cache is faster than access to entries in memory, this may lead to an increase in performance. With regular initialization, however, the chance that a lookup table entry is still in cache is greatly reduced.

5.4 Clique Covers

In this section we present an evaluation of the average and maximum clique size as well as clique cover size of our algorithm compared to cgm, as well as reduction rules paired with cgm (kernel+cgm). We ran the implementations on all instances listed in Table 5.1 and averaged the results over ten runs.

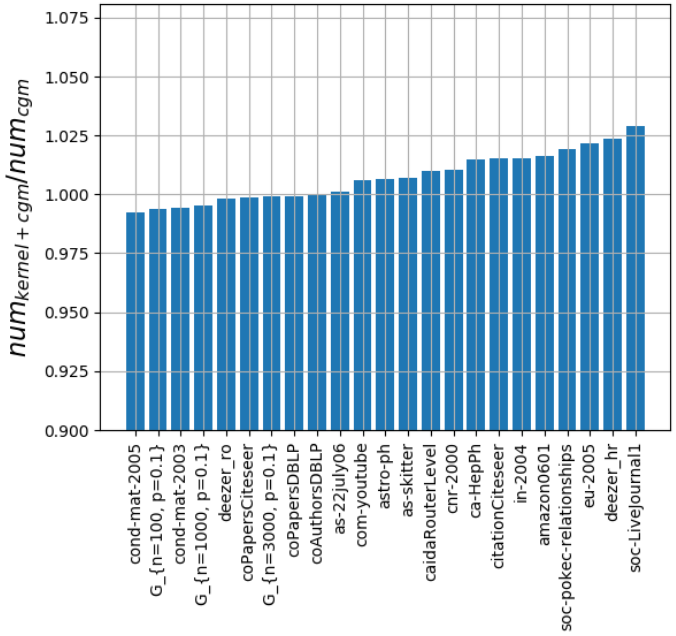
5.4.1 Kernel+cgm Versus cgm

First, we compare the combination of the reduction rules and cgm – denoted as kernel+cgm – against cgm. We define kernel+cgm to be the kernelization followed by five runs of cgm, where we take the best result. Therefore, the running time of kernel+cgm consists of the running time required for the kernelization as well as the running times required by the five runs of cgm on the kernel.

The most important indicator of result quality is the clique cover size. In Figure 5.3 we can see that the clique cover sizes of both approaches are very similar, though kernel+cgm on average results in a slightly larger clique cover. This is additionally confirmed by a geometric mean of $204\,934$ for cgm and a geometric mean of $206\,414$ for kernel+cgm, which again suggests that the clique covers by kernel+cgm are slightly larger.

Looking at Table B.4 we can see that kernel+cgm leads to a smaller clique cover on the instances cond-mat-2005, $G_{n=100,p=0.1}$, cond-mat-2003, $G_{n=1000,p=0.1}$,

Figure 5.3: Comparison of clique cover sizes (cc). A value smaller than 1 indicates that kernel+cgm results in a smaller clique cover. See Table B.4 for more detailed results.



`deezer_ro`, `coPapersCiteseer`, $G_{n=3000,p=0.1}$, `coPapersDBLP` and `coAuthorsDBLP`. cgm reports smaller clique covers on the remaining 14 instances.

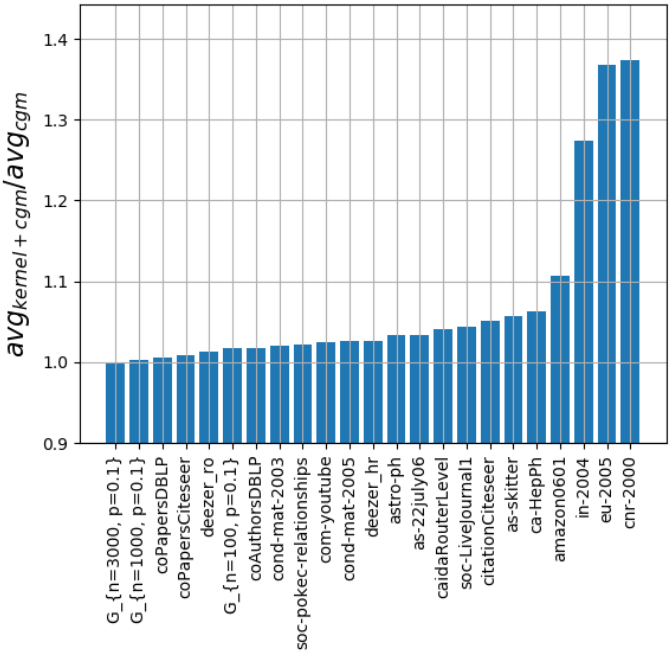
This result indicates, that running kernel+cgm leads to poorer results. Though keep in mind, as we noted in Section 4.2.5, due to the structure of the kernel, the heuristic may find redundant cliques on the kernel and therefore lead to worse clique covers.

Next, we compare the average clique sizes reported by both approaches. If we look at Figure 5.4, we can see that kernel+cgm reports larger average clique sizes on almost all instances, except $G_{n=3000,p=0.1}$, where kernel+cgm and cgm report the same average clique size. The average clique size reported by kernel+cgm is 3.641, whereas the average clique size reported by cgm is 3.416. This suggests, that on average kernel+cgm leads to larger average cliques.

Next, we compare the maximum clique sizes. In Figure 5.5 and Table B.6 we can see that cgm reports slightly larger maximum cliques on the two instances `eu-2005` and `in-2004`, whereas kernel+cgm reports larger maximum clique sizes on the instances `soc-LiveJournal1`, `citationCiteseer` and `deezer_ro`. The geometric mean of the maximum clique size report by kernel+cgm is 40.537, whereas the geometric mean for cgm is 40.120. This suggests that kernel+cgm on average reports slightly larger maximum cliques.

In terms of running times, kernel+cgm is slower on 16 out of 23 instances. This can be seen in Figure 5.6. The geometric mean of 9.671 seconds for kernel+cgm versus 6.129

Figure 5.4: Comparison of average clique sizes. A value smaller than 1 indicates that kernel+cgm results in a smaller average clique size. See Table B.5 for more detailed results.



seconds for cgm further indicates, that kernel+cgm is on average slower than cgm.

On the two instances `eu-2005` and `cnr-2000` kernel+cgm requires over 12 times the running time of cgm. On both instances the majority of the running time is spent on kernelization. More specifically, on the instance `eu-2005` almost 1 100 seconds out of the 1 400 seconds of running time is spent on kernelization, out of those 1 100 seconds the initialization takes 1 040 seconds, the remainder is spent on the reduction rules. On the instance `cnr-2000` the kernelization takes 190 seconds and the heuristic only takes 30 seconds. The running time of the kernelization is split into 155 seconds for the initialization and 35 seconds for the reduction rules.

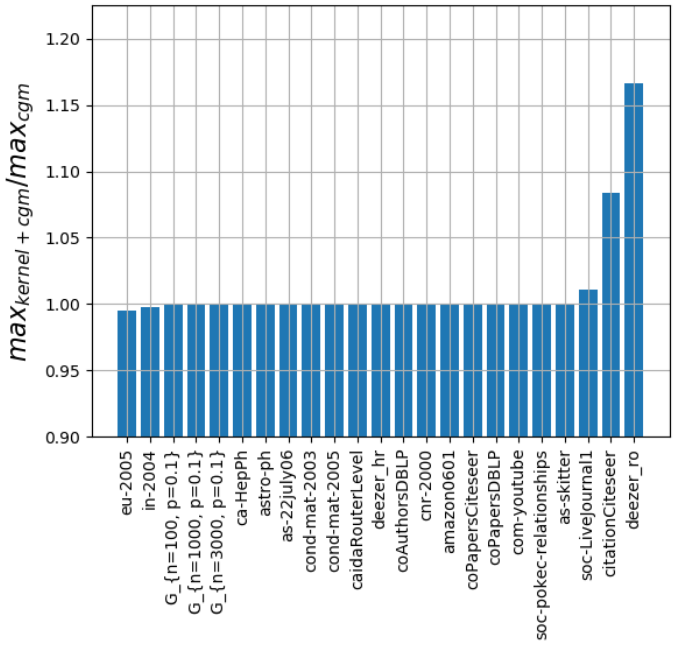
5.4.2 Our Algorithm Versus cgm

We will now present experimental data on a comparison of our versus cgm. For our implementation we use pre-processing and lazy initialization.

The most important indicator of result quality is the size of the clique cover. We can see in Figure 5.7 and Table B.8 that our finds a smaller clique cover on 15 out of 23 instances, whereas cgm finds a smaller clique cover on the remaining eight instances.

The geometric mean of the clique cover sizes for our is 209832, whereas for cgm the geometric mean is 204919, this suggests that although our algorithm finds smaller clique covers on more instances, on average the clique covers of our implementation are larger than

Figure 5.5: Comparison of maximum clique sizes. A value smaller than 1 in indicates that kernel+cgm results in a smaller maximum clique size. See Table B.6 for more detailed results.

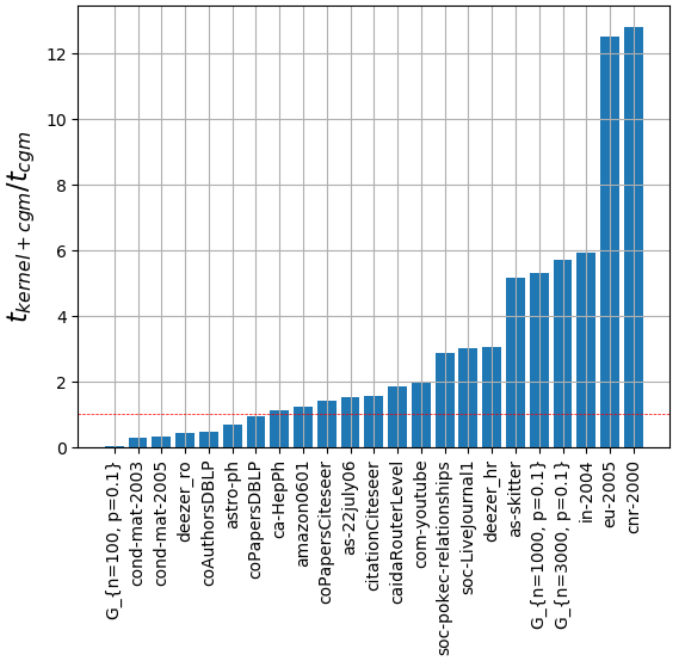


the clique covers by cgm. We can see in Table B.8 that on our best graph – `amazon0601` – our clique cover is 2.5% smaller than cgm, however, on our worst graph – $G_{n=3000,p=0.1}$ – our clique cover is over 60% larger.

If we cross-reference the clique cover sizes with the size of the graph kernels of Table B.1, we notice that the combined approach of kernelization and our heuristic seems to work best on graphs where the kernelization admits a very small graph kernel. This suggests that in our heuristic we may need to further investigate the choice of the lookup table entry which is reduced, such that graphs with larger kernels may be solved with better results. In Table B.1 we also see, that the instance $G_{n=100,p=0.1}$ is solved exactly, as the graph is completely reduced by the reduction rules.

Next, we evaluate the average clique sizes computed by the algorithms. As we can see in Figure 5.8, our results in a larger average clique size in all cases, except for the $G_{n=3000,p=0.1}$ graph. The geometric mean over all average clique sizes for our is 3.887 whereas for cgm it's 3.146, which again shows that our provides larger average clique sizes. However, if we look at Table B.8 we can see that for the six instances $G_{n=1000,p=0.1}$, `as-Skitter`, `soc-LiveJournal1`, `in-2004`, `cnr-2000` and `eu-2005` the clique cover size calculated by our is much larger than the clique cover calculated by cgm. Therefore, the large difference in average clique sizes on these instances may be caused by the difference in clique cover sizes, as the average of a larger data set may increase naturally.

Figure 5.6: Comparison of running times. A value smaller than 1 indicates that kernel+cgm results in a faster running time. Red dashed line marks $y = 1.0$. See Table B.7 for more detailed results.



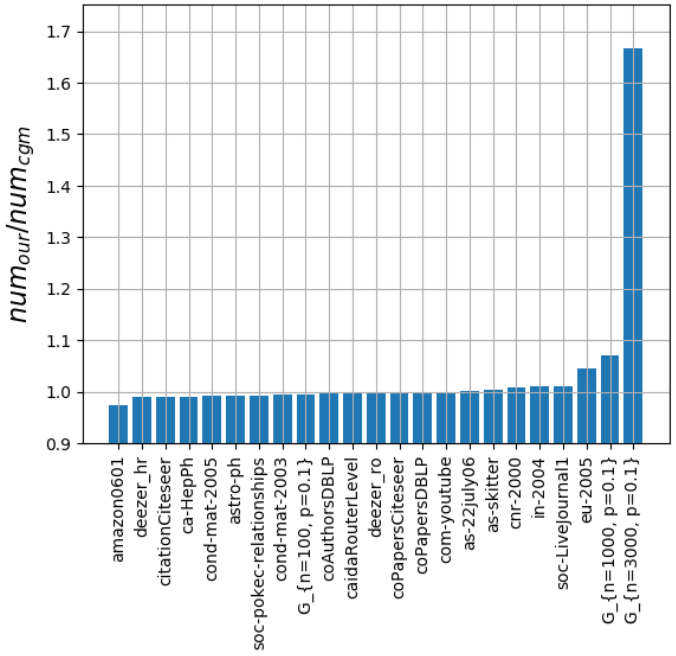
On almost all other instances, where we achieve a larger average clique size, our also achieves a smaller clique cover size. This suggests, that on those instances our is able to cover the graphs with fewer and larger cliques than cgm.

Next, we look at the maximum clique size found by the algorithms. As we can see in Figure 5.9, the maximum clique found by cgm is larger on four instances, whereas on five instances our algorithm finds a larger maximum clique. On the remaining 14 cliques the maximum clique size is identical. The geometric mean over all maximum clique sizes for our is 41.051, whereas for cgm it's 40.408, which suggest that on average our algorithm finds slightly larger maximum cliques.

Finally, we will take a look at the running times required by both implementations. As we can see in Figure 5.10 and Table B.11, on a few instances our implementation outperforms cgm significantly, however, on two instances – `eu-2005` and `cnr-2000` – our implementation takes 18 times longer than cgm. This result is similar to the comparison of kernel+cgm and cgm. On the instance `eu-2005` about 250 seconds are spent on the maximal clique heuristic, 1 695 seconds are spent on initialization and reductions, 665 seconds are spent on the reduction rules and 23 seconds are spent on post-processing. On the instance `cnr-2000` the maximal clique heuristic takes 43 seconds, the remaining 292 seconds are spent on initialization and reductions.

The geometric mean of 6.162 seconds for the running time of our versus 5.950 seconds

Figure 5.7: Comparison of clique cover sizes (cc). A value smaller than 1 indicates that our algorithm results in a smaller clique cover. See Table B.8 for more detailed results.



for cgm confirms that cgm is on average slightly faster.

5.4.3 Instances Solved Over Time

Lastly, we will compare the running times of our, kernel+cgm and cgm by plotting the number of instances solved over time. The result can be seen in Figure 5.11 We can clearly see, that cgm has an advantage over our and kernel+cgm, as it requires the least time to solve all instances. kernel+cgm and our are very similar in terms of running time, though our requires more running time on its slowest instance.

Figure 5.8: Comparison of average clique sizes. A value smaller than 1 indicates that our algorithm results in a smaller average clique size. See Table B.9 for more detailed results.

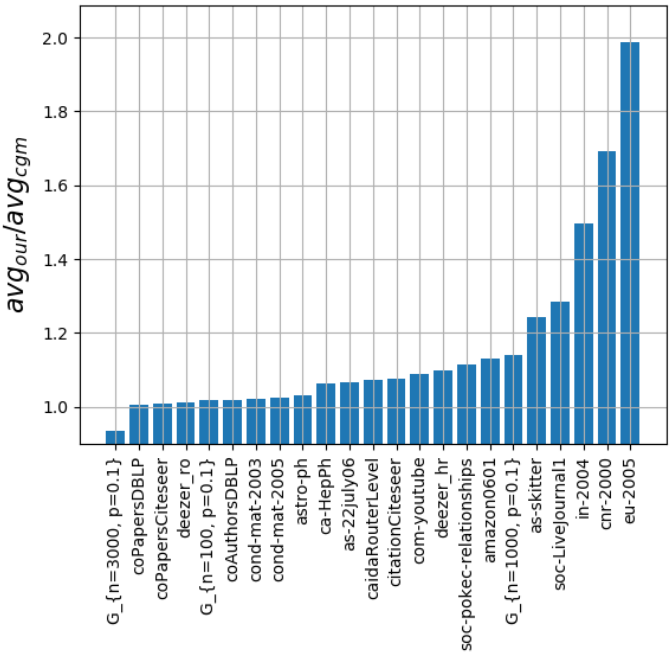


Figure 5.9: Comparison of maximum clique sizes. A value smaller than 1 in indicates that our algorithm results in a smaller maximum clique size. See Table B.10 for more detailed results.

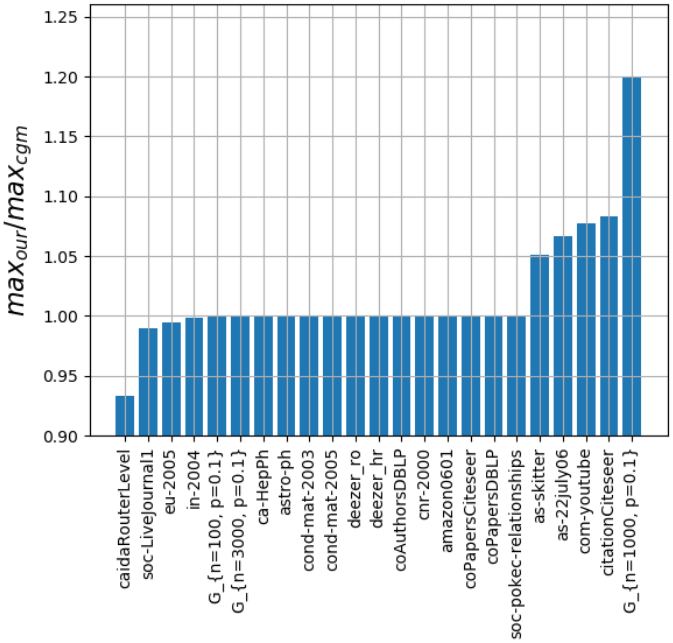


Figure 5.10: Comparison of running times. A value smaller than 1 indicates that our algorithm results in a faster running time. Red dashed line marks $y = 1.0$. See Table B.11 for more detailed results.

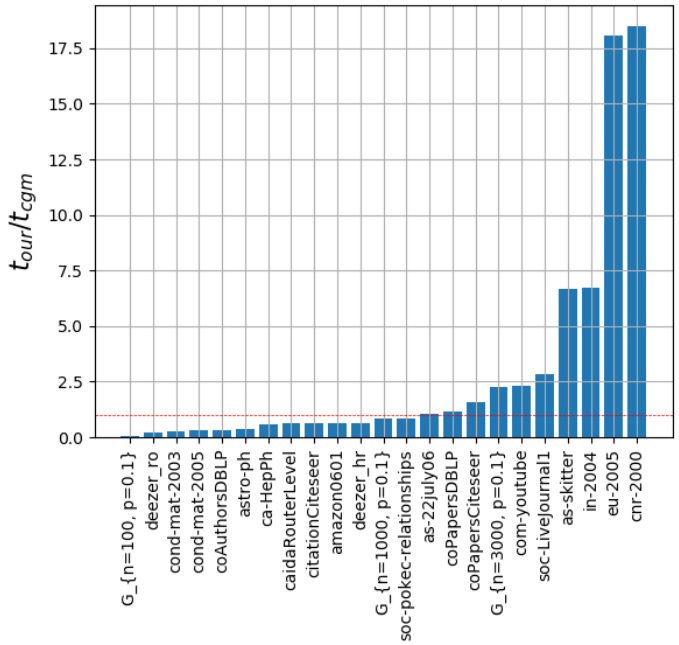
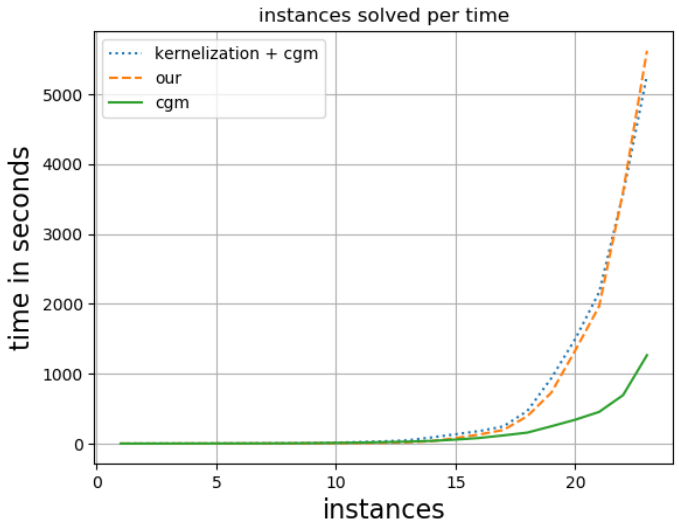


Figure 5.11: Comparison of instances solved over time of our, kernel+cgm and cgm.



6 Discussion

In this thesis we presented a new approach for solving the edge clique cover problem, which combined kernelization with our newly developed heuristic. Additionally, we evaluated a combination of kernelization and the currently best known heuristic. We evaluated the quality of our approach by comparing average clique sizes, maximum clique sizes, clique cover size and running time against the currently best known heuristic. These comparisons were made using a wide range of instances. The approach of combining kernelization with a clique selection heuristic seems very promising. Our newly developed heuristic in combination with the reduction rules beats the currently best known heuristic on 14 out of 23 instances, in terms of minimum clique cover. Combining kernelization with the currently best known heuristic beats the currently best known heuristic on a few graphs as well, here it's nine graphs out of 23. Together, both approaches of kernelization with heuristic outperform current approaches on 16 out of 23 instances. Therefore, this new approach is very promising and can lead to better results.

6.1 Future Work

As the running time of our algorithm is very slow on some instances, future research in speeding up the the kernelization is of significant interest. Furthermore, the choice of the lookup table entry in our heuristic may be improved, as that choice seems to be the driving factor in reducing the clique cover number. Currently, we sort the remaining lookup table entries once based on the size of the common neighborhood, however, it may be interesting to re-sort after a certain number of reduced cliques, to reflect changes in the common neighborhood sizes. Additionally, choosing a different sorting condition may lead to improved results. It could be worth investigating different existing heuristics combined with kernelization, as the combination can lead to an improvement in results.

Bibliography

[1] ABU-KHZAM, F. N., COLLINS, R. L., FELLOWS, M. R., LANGSTON, M. A., SUTERS, W. H., AND SYMONS, C. T. Kernelization algorithms for the vertex cover problem: Theory and experiments. *ALENEX/ANALC 69* (2004).

[2] AGARWAL, P. K., ALON, N., ARONOV, B., AND SURI, S. Can visibility graphs be represented compactly? *Discrete & Computational Geometry 12*, 3 (Sep 1994), 347–365.

[3] BADER, D. A., MEYERHENKE, H., SANDERS, P., SCHULZ, C., KAPPES, A., AND WAGNER, D. *Benchmarking for Graph Clustering and Partitioning*. Springer New York, New York, NY, 2014, pp. 73–82.

[4] BEHRISCH, M., AND TARAZ, A. Efficiently covering complex networks with cliques of similar vertices. *Theoretical Computer Science 355*, 1 (2006), 37 – 47. Complex Networks.

[5] BRON, C., AND KERBOSCH, J. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM 16*, 9 (Sept. 1973), 575–577.

[6] CHLEBÍK, M., AND CHLEBÍKOVÁ, J. Crown reductions for the minimum weighted vertex cover problem. *Discrete Applied Mathematics 156*, 3 (2008), 292–312.

[7] CONTE, A., GROSSI, R., AND MARINO, A. Clique covering of large real-world networks. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (New York, NY, USA, 2016), SAC '16, ACM, pp. 1134–1139.

[8] CYGAN, M., KRATSCH, S., PILIPCZUK, M., PILIPCZUK, M., AND WAHLSTRÖM, M. Clique cover and graph separation: New incompressibility results. *ACM Trans. Comput. Theory 6*, 2 (May 2014), 6:1–6:19.

[9] DEHNE, F., FELLOWS, M., ROSAMOND, F., AND SHAW, P. Greedy localization, iterative compression, and modeled crown reductions: New fpt techniques, an improved algorithm for set splitting, and a novel 2k kernelization for vertex cover. In *Parameterized and Exact Computation* (Berlin, Heidelberg, 2004), R. Downey, M. Fellows, and F. Dehne, Eds., Springer Berlin Heidelberg, pp. 271–280.

[10] DOWNEY, R. G., AND FELLOWS, M. R. *Parameterized complexity*. Springer Science & Business Media, 1999.

[11] ERDŐS, P., AND RÉNI, A. On random graphs i. *Publ. Math. Debrecen 6* (1959), 290–297.

[12] ERDS, P., AND RÉNYI, A. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci 5* (1960), 17–61.

[13] GRAMM, J., GUO, J., HÜFFNER, F., AND NIEDERMEIER, R. Data reduction and exact algorithms for clique cover. *J. Exp. Algorithmics 13* (Feb. 2009), 2:2.2–2:2.15.

[14] GRAMM, J., GUO, J., HÜFFNER, F., NIEDERMEIER, R., PIEPHO, H.-P., AND SCHMID, R. Algorithms for compact letter displays: Comparison and evaluation. *Computational Statistics & Data Analysis 52*, 2 (2007), 725–736.

[15] GRAMM, J., GUO, J., HÜFFNER, F., AND NIEDERMEIER, R. *Data Reduction, Exact, and Heuristic Algorithms for Clique Cover*. pp. 86–94.

[16] GROHE, M. Descriptive and parameterized complexity. In *Computer Science Logic* (Berlin, Heidelberg, 1999), J. Flum and M. Rodriguez-Artalejo, Eds., Springer Berlin Heidelberg, pp. 14–31.

[17] GUO, J., AND NIEDERMEIER, R. Invitation to data reduction and problem kernelization. *SIGACT News 38*, 1 (Mar. 2007), 31–45.

[18] KARYPIS, G., AND KUMAR, V. Metis—a software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing ordering of sparse matrices.

[19] KELLERMAN, E. Determination of keyword conflict. *IBM Technical Disclosure Bulletin 16*, 2 (1973), 544–546.

[20] KOU, L. T., STOCKMEYER, L. J., AND WONG, C. K. Covering edges by cliques with regard to keyword conflicts and intersection graphs. *Commun. ACM 21*, 2 (Feb. 1978), 135–139.

[21] LESKOVEC, J., AND KREVL, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[22] ORLIN, J. Contentment in graph theory: Covering graphs with cliques. *Indagationes Mathematicae (Proceedings) 80*, 5 (1977), 406 – 424.

[23] RAJAGOPALAN, S., VACHHARAJANI, M., AND MALIK, S. Handling irregular ilp within conventional vliw schedulers using artificial resource constraints. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (New York, NY, USA, 2000), CASES '00, ACM, pp. 157–164.

[24] ROBERTS, F. S. Applications of edge coverings by cliques. *Discrete applied mathematics 10*, 1 (1985), 93–109.

[25] TOMITA, E., TANAKA, A., AND TAKAHASHI, H. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science 363*, 1 (2006), 28–42.

A Implementation Details

Our implementation was done in C++11. The program is based on the KaHIP [1] framework.

Graphs must be provided in the METIS format [18]. Our implementation supports undirected, unweighted graphs with no self loops. The following listing shows the helptext of our program, which explains the possible commands.

The first parameter needs to be a path to a graph file in METIS format [18]. The parameter `--output_filename` specifies the location where the kernel will be written to, though note that this is not applicable, when the heuristic is enabled. The kernel will be in the METIS format [18] as well. The parameters `--disable-preprocessing` and `--disable-lazy` can be used to disable the preprocessing – which is described in Section 4.2.1 – and the lazy initialization – described in Section 4.2.2. Lastly, the parameter `--heuristic` enables the heuristic described in Section 4.2.6.

Listing A.1: Helptext of our implementation

```
./deploy/edge_clique_cover --help
Usage: ./deploy/edge_clique_cover FILE options
where options are:
--help                              Print help.
FILE                                Path to graph file.
--output_filename=<string>          Specify the name of the output file
                                    that will contain the kernelized graph.
--disable-preprocessing             disable preprocessing of degree-1 vertices
--disable-lazyinit                  disable lazy initialization
--heuristic                         enables heuristic after kernelization
```

[1] https://github.com/schulzchristian/KaHIP

B Detailed Results

In this section we provide more detailed data of the experimental evaluations.

B.1 Kernel Sizes

Table B.1 lists all instances with the size of the kernel as computed by the kernelization routine.

Table B.1: Graph instances used for evaluation with the size of the computed kernel

	original graph		kernel		kernel / graph	
graph	n	m	n	m	n	m
$G_{n=100,p=0.1}$	100	488	0	0	0.000	0.000
$G_{n=1000,p=0.1}$	1 000	49 576	1 000	49 576	1.000	1.000
$G_{n=3000,p=0.1}$	3 000	450 085	3 000	450 085	1.000	1.000
astro-ph	16 706	121 251	242	3 409	0.014	0.028
as-22july06	22 963	48 436	487	5 814	0.021	0.120
cond-mat-2003	31 163	120 029	36	120	0.001	0.001
cond-mat-2005	40 421	175 691	29	95	0.001	0.001
caidaRouterLevel	192 244	1 218 132	17 949	158 696	0.093	0.130
citationCiteseer	268 495	2 313 294	34 763	277 332	0.129	0.120
coAuthorsDBLP	299 067	977 676	823	6 823	0.003	0.007
cnr-2000	325 557	2 738 969	41 875	928 467	0.129	0.339
coPapersCiteseer	434 102	16 036 720	50	1 203	0.001	0.000
coPapersDBLP	540 486	15 245 729	61	1 347	0.001	0.000
eu-2005	862 664	16 138 468	366 836	9 216 361	0.425	0.571
in-2004	1 382 908	13 591 473	200 772	3 617 400	0.145	0.266
ca-HepPh	12 006	118 489	378	5 907	0.031	0.050
deezer_ro	41 773	125 826	480	1 561	0.011	0.012
deezer_hr	54 573	498 202	21 594	291 232	0.396	0.585
amazon0601	403 394	2 443 408	95 229	571 406	0.236	0.234
com-youtube	1 134 890	2 987 624	41 370	815 540	0.036	0.273
soc-pokec-relationships	1 632 803	22 301 964	572 251	13 878 686	0.350	0.622
as-Skitter	1 696 415	11 095 298	374 438	5 964 615	0.221	0.538
soc-LiveJournal1	4 846 609	42 851 237	1 179 777	26 792 241	0.243	0.625

B.2 Pre-Processing

Table B.2 lists the speed-up when enabling the pre-processing, together with the actual running times.

Table B.2: Speedup when using preprocessing (t_p) versus without preprocessing (t_{np}).

	running time in seconds		
graph	t_{np}	t_p	t_{np}/t_p
$G_{n=100,p=0.1}$	**0.001**	0.001	0.727
$G_{n=1000,p=0.1}$	0.358	**0.340**	1.054
$G_{n=3000,p=0.1}$	16.256	**16.063**	1.012
astro-ph	0.166	**0.163**	1.015
as-22july06	0.307	**0.293**	1.049
cond-mat-2003	0.123	**0.120**	1.029
cond-mat-2005	0.211	**0.202**	1.043
caidaRouterLevel	1.986	**1.914**	1.038
citationCiteseer	**4.063**	4.087	0.994
coAuthorsDBLP	1.376	**1.345**	1.022
cnr-2000	329.429	**322.550**	1.021
coPapersCiteseer	64.231	**63.485**	1.012
coPapersDBLP	**43.272**	43.826	0.987
eu-2005	2 003.436	**1 969.593**	1.017
in-2004	657.689	**623.956**	1.054
ca-HepPh	0.325	**0.307**	1.057
deezer_ro	0.172	**0.167**	1.031
deezer_hr	1.692	**1.675**	1.010
amazon0601	6.737	**6.584**	1.023
com-youtube	58.972	**52.619**	1.121
soc-pokec-relationships	**186.984**	199.159	0.939
as-Skitter	**605.803**	615.150	0.985
soc-LiveJournal1	**1 610.910**	1 643.717	0.980

B.3 Lazy Initialization

Table B.3 lists the speed-up of enabling lazy initialization over regular initialization together with the actual running times.

Table B.3: Speedup when using lazy initialization (t_{lazy}) instead of regular initialization ($t_{regular}$).

	running time in seconds		
graph	$t_{regular}$	t_{lazy}	$t_{regular}/t_{lazy}$
$G_{n=100,p=0.1}$	0.001	**0.001**	1.152
$G_{n=1000,p=0.1}$	**0.347**	0.415	0.836
$G_{n=3000,p=0.1}$	**16.255**	16.424	0.990
astro-ph	0.497	**0.192**	2.591
as-22july06	0.343	**0.346**	0.992
cond-mat-2003	0.288	**0.136**	2.114
cond-mat-2005	0.478	**0.222**	2.149
caidaRouterLevel	2.643	**1.949**	1.356
citationCiteseer	5.602	**4.070**	1.376
coAuthorsDBLP	3.152	**1.436**	2.195
cnr-2000	414.949	**335.937**	1.235
coPapersCiteseer	1 188.033	**62.937**	18.876
coPapersDBLP	381.253	**43.090**	8.848
eu-2005	2 481.053	**2 020.265**	1.228
in-2004	2 130.850	**632.648**	3.368
ca-HepPh	3.050	**0.361**	8.442
deezer_ro	0.279	**0.180**	1.549
deezer_hr	2.164	**1.737**	1.246
amazon0601	9.882	**6.750**	1.464
com-youtube	59.132	**53.893**	1.097
soc-pokec-relationships	238.125	**198.291**	1.201
as-Skitter	686.773	**603.628**	1.138
soc-LiveJournal1	1 926.493	**1 634.968**	1.178

B.4 Clique Cover Measures

B.4.1 Kernel+cgm Versus cgm

Table B.4 lists a comparison of clique cover sizes. Table B.5 lists a comparison of average clique sizes. Table B.6 lists a comparison of maximum clique sizes. Table B.7 lists a comparison of running times.

B.4.2 Our Algorithm Versus cgm

Table B.8 lists a comparison of clique cover sizes. Table B.9 lists a comparison of average clique sizes. Table B.10 lists a comparison of maximum clique sizes. Table B.11 lists a comparison of running times.

Table B.4: Comparison of clique cover sizes (cc). A value smaller than 1 indicates that kernel+cgm results in a smaller clique cover.

	clique cover sizes		
graph	kernel+cgm	cgm	$cc_{kernel+cgm}/cc_{cgm}$
$G_{n=100,p=0.1}$	**332**	334	0.994
$G_{n=1000,p=0.1}$	**15 567**	15 639	0.995
$G_{n=3000,p=0.1}$	**102 909**	103 007	0.999
astro-ph	10 149	**10 083**	1.007
as-22july06	34 006	**33 963**	1.001
cond-mat-2003	**20 429**	20 544	0.994
cond-mat-2005	**27 657**	27 876	0.992
caidaRouterLevel	407 109	**403 160**	1.010
citationCiteseer	703 844	**693 295**	1.015
coAuthorsDBLP	**223 642**	223 679	1.000
cnr-2000	764 663	**756 828**	1.010
coPapersCiteseer	**70 539**	70 638	0.999
coPapersDBLP	**100 476**	100 559	0.999
eu-2005	2 945 228	**2 883 492**	1.021
in-2004	2 433 207	**2 396 153**	1.015
ca-HepPh	10 306	**10 155**	1.015
deezer_ro	**95 287**	95 462	0.998
deezer_hr	261 173	**255 102**	1.024
amazon0601	838 448	**825 107**	1.016
com-youtube	2 216 219	**2 202 778**	1.006
soc-pokec-relationships	12 875 812	**12 631 035**	1.019
as-Skitter	6 074 354	**6 031 503**	1.007
soc-LiveJournal1	18 020 359	**17 510 992**	1.029

Table B.5: Comparison of average clique sizes. A value smaller than 1 indicates that kernel+cgm results in a smaller average clique size.

	average clique size		
graph	kernel+cgm	cgm	$avg_{kernel+cgm}/avg_{cgm}$
$G_{n=100,p=0.1}$	**2.313**	2.275	1.017
$G_{n=1000,p=0.1}$	**3.221**	3.213	1.003
$G_{n=3000,p=0.1}$	**3.745**	3.744	1.000
astro-ph	**4.753**	4.602	1.033
as-22july06	**2.424**	2.346	1.033
cond-mat-2003	**3.836**	3.758	1.021
cond-mat-2005	**4.010**	3.908	1.026
caidaRouterLevel	**2.438**	2.344	1.040
citationCiteseer	**2.561**	2.436	1.051
coAuthorsDBLP	**3.321**	3.265	1.017
cnr-2000	**5.344**	3.890	1.374
coPapersCiteseer	**11.931**	11.838	1.008
coPapersDBLP	**10.212**	10.151	1.006
eu-2005	**6.260**	4.575	1.368
in-2004	**5.604**	4.397	1.275
ca-HepPh	**3.720**	3.500	1.063
deezer_ro	**2.218**	2.189	1.013
deezer_hr	**2.623**	2.556	1.026
amazon0601	**3.352**	3.027	1.107
com-youtube	**2.309**	2.253	1.025
soc-pokec-relationships	**2.525**	2.470	1.022
as-Skitter	**2.801**	2.652	1.056
soc-LiveJournal1	**2.915**	2.795	1.043

Table B.6: Comparison of maximum clique sizes. A value smaller than 1 indicates that kernel+cgm results in a smaller maximum clique size.

	maximum clique size		
graph	kernel+cgm	cgm	$max_{kernel+cgm}/max_{cgm}$
$G_{n=100,p=0.1}$	4	4	1.000
$G_{n=1000,p=0.1}$	5	5	1.000
$G_{n=3000,p=0.1}$	6	6	1.000
astro-ph	57	57	1.000
as-22july06	15	15	1.000
cond-mat-2003	25	25	1.000
cond-mat-2005	30	30	1.000
caidaRouterLevel	15	15	1.000
citationCiteseer	**13**	12	1.083
coAuthorsDBLP	115	115	1.000
cnr-2000	84	84	1.000
coPapersCiteseer	845	845	1.000
coPapersDBLP	337	337	1.000
eu-2005	385	**387**	0.995
in-2004	487	**488**	0.998
ca-HepPh	239	239	1.000
deezer_ro	**7**	6	1.167
deezer_hr	11	11	1.000
amazon0601	11	11	1.000
com-youtube	13	13	1.000
soc-pokec-relationships	29	29	1.000
as-Skitter	57	57	1.000
soc-LiveJournal1	**284**	281	1.011

Table B.7: Comparison of running times. A value smaller than 1 indicates that kernel+cgm results in a faster running time.

	running time in seconds		
graph	kernel+cgm	cgm	$t_{kernel+cgm}/t_{cgm}$
$G_{n=100,p=0.1}$	**0.001**	0.023	0.023
$G_{n=1000,p=0.1}$	2.691	**0.506**	5.320
$G_{n=3000,p=0.1}$	40.651	**7.124**	5.706
astro-ph	**0.443**	0.646	0.685
as-22july06	0.557	**0.366**	1.521
cond-mat-2003	**0.170**	0.594	0.286
cond-mat-2005	**0.249**	0.783	0.317
caidaRouterLevel	6.324	**3.388**	1.866
citationCiteseer	10.256	**6.567**	1.562
coAuthorsDBLP	**1.641**	3.372	0.487
cnr-2000	221.063	**17.269**	12.801
coPapersCiteseer	66.464	**46.457**	1.431
coPapersDBLP	**45.767**	49.062	0.933
eu-2005	1 406.322	**112.526**	12.498
in-2004	561.122	**94.900**	5.913
ca-HepPh	0.621	**0.551**	1.126
deezer_ro	**0.350**	0.837	0.419
deezer_hr	9.107	**2.986**	3.050
amazon0601	14.891	**12.034**	1.237
com-youtube	45.739	**22.684**	2.016
soc-pokec-relationships	668.845	**233.738**	2.862
as-Skitter	467.781	**90.321**	5.179
soc-LiveJournal1	1 702.030	**564.351**	3.016

Table B.8: Comparison of clique cover sizes (cc). A value smaller than 1 indicates that our results in a smaller clique cover.

	clique cover sizes		
graph	our	cgm	cc_{our}/cc_{cgm}
$G_{n=100,p=0.1}$	**332**	334	0.994
$G_{n=1000,p=0.1}$	16 691	**15 609**	1.069
$G_{n=3000,p=0.1}$	171 807	**103 005**	1.668
astro-ph	**9 998**	10 082	0.992
as-22july06	33 975	**33 960**	1.000
cond-mat-2003	**20 418**	20 546	0.994
cond-mat-2005	**27 646**	27 882	0.992
caidaRouterLevel	**401 440**	403 186	0.996
citationCiteseer	**685 931**	693 211	0.989
coAuthorsDBLP	**222 685**	223 678	0.996
cnr-2000	762 226	**756 863**	1.007
coPapersCiteseer	**70 538**	70 642	0.999
coPapersDBLP	**100 472**	100 565	0.999
eu-2005	3 014 772	**2 883 544**	1.046
in-2004	2 419 956	**2 396 168**	1.010
ca-HepPh	**10 054**	10 155	0.990
deezer_ro	**95 139**	95 468	0.997
deezer_hr	**252 266**	255 090	0.989
amazon0601	**804 237**	825 130	0.975
com-youtube	**2 201 402**	2 202 740	0.999
soc-pokec-relationships	**12 542 031**	12 630 901	0.993
as-Skitter	6 049 483	**6 031 413**	1.003
soc-LiveJournal1	17 691 949	**17 511 121**	1.010

Table B.9: Comparison of average clique sizes. A value smaller than 1 indicates that our results in a smaller average clique size.

	average clique size		
graph	our	cgm	avg_{our}/avg_{cgm}
$G_{n=100,p=0.1}$	**2.313**	2.273	1.018
$G_{n=1000,p=0.1}$	**3.670**	3.217	1.141
$G_{n=3000,p=0.1}$	3.505	**3.744**	0.936
astro-ph	**4.748**	4.601	1.032
as-22july06	**2.504**	2.346	1.068
cond-mat-2003	**3.837**	3.757	1.021
cond-mat-2005	**4.010**	3.908	1.026
caidaRouterLevel	**2.515**	2.344	1.073
citationCiteseer	**2.622**	2.436	1.076
coAuthorsDBLP	**3.324**	3.265	1.018
cnr-2000	**6.583**	3.890	1.692
coPapersCiteseer	**11.93**	11.839	1.008
coPapersDBLP	**10.212**	10.151	1.006
eu-2005	**9.088**	4.575	1.986
in-2004	**6.576**	4.397	1.496
ca-HepPh	**3.725**	3.501	1.064
deezer_ro	**2.217**	2.189	1.013
deezer_hr	**2.810**	2.555	1.100
amazon0601	**3.426**	3.027	1.132
com-youtube	**2.457**	2.253	1.091
soc-pokec-relationships	**2.755**	2.470	1.115
as-Skitter	**3.299**	2.652	1.244
soc-LiveJournal1	**3.591**	2.795	1.285

Table B.10: Comparison of maximum clique sizes. A value smaller than 1 indicates that our results in a smaller maximum clique size.

	maximum clique size		
graph	our	cgm	max_{our}/max_{cgm}
$G_{n=100,p=0.1}$	4	4	1.000
$G_{n=1000,p=0.1}$	**6**	5	1.200
$G_{n=3000,p=0.1}$	6	6	1.000
astro-ph	57	57	1.000
as-22july06	**16**	15	1.067
cond-mat-2003	25	25	1.000
cond-mat-2005	30	30	1.000
caidaRouterLevel	14	**15**	0.933
citationCiteseer	**13**	12	1.083
coAuthorsDBLP	115	115	1.000
cnr-2000	84	84	1.000
coPapersCiteseer	845	845	1.000
coPapersDBLP	337	337	1.000
eu-2005	385	**387**	0.995
in-2004	487	**488**	0.998
ca-HepPh	239	239	1.000
deezer_ro	7	7	1.000
deezer_hr	11	11	1.000
amazon0601	11	11	1.000
com-youtube	**14**	13	1.077
soc-pokec-relationships	28	28	1.000
as-Skitter	**62**	59	1.051
soc-LiveJournal1	281	**284**	0.989

Table B.11: Comparison of running times. A value smaller than 1 indicates that our results in a faster running time.

	running time in seconds		
graph	our	cgm	t_{our}/t_{cgm}
$G_{n=100,p=0.1}$	**0.001**	0.024	0.038
$G_{n=1000,p=0.1}$	**0.415**	0.503	0.824
$G_{n=3000,p=0.1}$	16.424	**7.225**	2.273
astro-ph	**0.192**	0.520	0.369
as-22july06	0.346	**0.333**	1.040
cond-mat-2003	**0.136**	0.544	0.250
cond-mat-2005	**0.222**	0.702	0.317
caidaRouterLevel	**1.949**	3.197	0.610
citationCiteseer	**4.070**	6.396	0.636
coAuthorsDBLP	**1.436**	4.396	0.327
cnr-2000	335.937	**18.178**	18.481
coPapersCiteseer	62.937	**40.020**	1.573
coPapersDBLP	43.090	**36.763**	1.172
eu-2005	2020.265	**111.967**	18.043
in-2004	632.648	**93.945**	6.734
ca-HepPh	**0.361**	0.641	0.564
deezer_ro	**0.180**	0.849	0.212
deezer_hr	**1.737**	2.668	0.651
amazon0601	**6.750**	10.595	0.637
com-youtube	53.893	**23.451**	2.298
soc-pokec-relationships	**198.291**	239.586	0.828
as-Skitter	603.628	**90.232**	6.690
soc-LiveJournal1	1634.968	**571.81**	2.859