On Variants of Clique Problems and their Applications

by

Zhuoli Xiao
B.Sc., University of University of Victoria, 2014

A Project Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

On Variants of Clique Problems and their Applications

by

Zhuoli Xiao
B.Sc., University of University of Victoria, 2014

Supervisory Committee

Dr. Ulrike Stege, Supervisor
(Department of Computer Science)

Dr. Hausi A. Muller, Commitee Member
(Department of Computer Science)

ABSTRACT

Clique-based problems, often called cluster problems, receive more and more attention in computer science as well as many of its application areas. Clique-based problems can be used to model a number of real-world problems and with this comes the motivation to provide algorithmic solutions for them. While one can find quite a lot of literature on clique-based problems in general, only for few specific versions exact algorithmic solutions are discussed in detail. In this project, we survey such clique-based problems from the literature, investigate their properties, their exact algorithms and respective running times, and discuss some of their applications.

In particular, in depth we consider the two NP-complete clique-based problems Edge Clique Partition and Edge Clique Cover. For their natural parameterizations, k-Edge Clique Partition and k-Edge Clique Cover, we present fixed-parameter algorithms based on results in the literature, suggest some pre-processing rules (also called reduction rules) that we have not seen in the literature, as well as discuss some of their current and potential real-world applications.

Contents

ACKNOWLEDGEMENTS

I would like to thank:

Supervisor Ulrike Stege, for mentoring, support, encouragement, and patience.

Parents Xuliang Xiao and Xingduan Guo, for supporting my study and daily life.

Without their help, it would be impossible for me to attain education at the University of Victoria and pursue my master's degree in Computer Science.

Chapter 1

Introduction

Nowadays, with the advances of technology, the role of computer science is gaining in importance in many fields, including the natural sciences, engineering and social sciences. In this report, we survey in depth two (computational) problems and their algorithmic solutions from the world of clique or cluster problems.

A computational problem has the property to be *computationally intractable* if it is not solvable efficiently[1] in the general case. An important complexity class of intractable computational problems is the one of NP-hard (decision) problems. For no NP-hard problem there is a known algorithm that solves the problem for any given instance in polynomial time. Finding a polynomial-time algorithm that solves an NP-hard problem would answer the famous "P=NP?"-question affirmatively. Typical running times of algorithms for NP-hard problems are exponential in the size of the input. Much progress has been achieved tackling NP-hard problems despite their intractability property. Techniques to deal with these problems include fixed-parameter algorithms [8, 9, 19], approximation algorithms[2], and heuristics[3]. While approximation algorithms and heuristics can be fast, they do not guarantee to deliver optimum solutions, an outcome that is not always satisfactory or deemed acceptable.

Parameterized complexity deals with NP-hardness (or classical intractability in gen-

[1] A computational problem is understood to be solvable efficiently if it can be solved by an algorithm in polynomial time, for any input.

[2] An approximation algorithm is an algorithm that returns a solution with a guaranteed quality: the solution obtained is within a multiplicative factor of the optimal one [22].

[3] A heuristic algorithm is an algorithm that reaches a possible solution to the given problem with no optimization guarantee [17].

eral) and includes the design of fixed-parameter algorithms [8, 9, 15, 19]. In parameterized complexity, the problem complexity is studied using a second dimension, the *problem parameter*. Intuitively, parameterized decision problems are distinguished to be either tractable for the specific parameterization considered (such problems are said to be *fixed-parameter tractable* or members of the complexity class FPT, the class of *fixed-parameter-tractable parameterized decision problems*) or *parameterized intractable* (that is they are shown to be *hard* for a parameterized class such as complexity class W[1], a superclass of FPT), and therefore unlikely to be in FPT.

To understand membership in FPT, we consider a decision problem that is associated with one or more parameters. For simplicity, assume the problem parameter is called k, and an instance to our parameterized problem is described as $I =< X, k >$. Then our parameterized problem is in FPT if it can be solved in time $O(n^{O(1)} + f(k))$, where $n = |X|$ is the size of the problem instance and $f(k)$ is a function that depends only on parameter k, that is, function f is independent of n.

Note that, while $f(k)$ can be very large (e.g., exponential in parameter k), if the value of parameter k is small and f does not grow too fast, such a running time can indeed describe a practical algorithm for a large number of instances of substantial size.

Problems that are hard or complete for W[1] typically have running times of the order of $\binom{n}{k}$. A parameterized problem that is hard for W[1] is assumed to be not a member of FPT unless FPT=W[1]. Just like the conjectured answer to the "Is P = NP"-question is "no" by most complexity theorists, FPT = W[1] is assumed to be false also. For further reading on parameterized intractability see, for example, [7, 8, 15].

A rich toolkit of algorithm-design techniques has been developed for solving problems in FPT and designing such algorithms. This toolkit includes the use of polynomial time *reduction rules* and *kernelization*, as well as the *technique of bounded search trees*.

In this report, we concentrate on two parameterized problems discussed in the literature, k-Edge Clique Partition and k-Edge Clique Cover. Both problems benefit from determining (large) complete subgraphs. We investigate both their algorithmic approaches and their applications. Note that complete subgraphs are also called *cliques*.

The most famous clique-based problems is the classic Clique problem. The goal of Clique is to determine, for a given graph, whether a given graph has a clique consisting of at least k vertices. While this classic problem, when parameterized by clique size k is complete for class W[1][4], several of its variants are proven to be in FPT [15].

Our main focus of this MSc project was to discuss algorithmic approaches that deal with clique-based graph problems as a tool to assist solving such problems in other fields. We introduce and study the parameterized decision problems, k-Edge Clique Partition and k-Edge Clique Cover, discuss the complexity of these problems as well as an algorithmic solution for both. Then we discuss applications of these problems in the fields of hardware design, experimental data analysis and (graph-based) data compression.

The remainder of this report is organized as follows. In Chapter 2, we introduce terminology and background knowledge for this project. In Chapter 3, we list a variety of clique-based problems encountered during our literature review. Then for both problems, k-Edge Clique Partition and k-Edge Clique Cover, we discuss FPT algorithms and their running times (Chapter 4), and some of their respective applications (Chapter 5). Finally, we present selected future work in Chapter 6.

[4]A parameterized decision problem is complete for class W[1] if it is (1) hard for class W[1] and (2) a member of W[1] [8].

Chapter 2

Background and Terminology

In this chapter, we introduce terminology required in this document, including basic graph theory. For further resources we refer to [4, 8, 11].

Graph: All *graphs*, typically denoted $G = (V, E)$ with vertex set V and edge set E, are undirected and simple unless otherwise stated.

Open Neighborhood: Given a graph $G = (V, E)$ and a vertex $v \in V$, the *open neighborhood* of v in G is defined as $N(v) = \{w \in V : v \neq w, vw \in E\}$.

Closed Neighborhood: Given a graph $G = (V, E)$ and a vertex $v \in V$, the *closed neighborhood* of v in G is defined as $N[v] = N(v) \cup \{v\}$.

Induced Subgraph: For a graph $G = (V, E)$ with $V' = \{x_1, x_2, ..., x_m\} \subseteq V$, $G(x_1, x_2, ..., x_m) = (V', E')$ denotes the graph that is the by V' *induced subgraph* of G. That is for $G(x_1, x_2, ..., x_m)$, for every pair of vertices $x, y \in V'$, $xy \in E'$ if and only if $xy \in E$. $E(V') = E'$ denotes the set of edges in the by V' induced subgraph.

Complete Graph/Clique: A graph $G = (V, E)$ is *complete* if and only if for every pair of vertices $u, v \in V$ there is an edge $uv \in E$. The vertex set of a complete (sub)graph is also called *clique*.

Note that graphs that contain exactly one vertex and graphs containing no vertices are both cliques.

Maximal Clique [2]: Let $G = (V, E)$ and $C \subseteq V$ be a clique in G. Then C is *maximal* in G if and only if there is no vertex $v \in V$ with $v \notin C$ and $C \cup \{v\}$ is a clique in G.

In other words, a maximal clique cannot be extended. Therefore, no maximal clique is a subset of any other clique in the same graph.

K_n: The complete graph with n vertices is denoted K_n.

K_n-free Graph: A *K_n-free* graph is a graph that does not contain any complete subgraph consisting of n vertices.

Bipartite Clique: Given graph $G = (V, E)$, a *bipartite clique* B in G is a subset of vertices $B \subseteq V$ if the vertices in B can be divided into two disjoint sets V_1, V_2, with:

1. $V_1 \cup V_2 = B$.
2. $V_1 \cap V_2 = \emptyset$.
3. Every vertex in V_1 is adjacent to every vertex in V_2 in graph G.
4. For every pair of vertices $u, v \in B$, if $u, v \in V_1$ or $u, v \in V_2$, then $uv \notin E$.

Edge Clique Partition: Given a graph $G = (V, E)$, an *edge clique partition* of G is a set $CP = \{C_1, C_2, ..., C_l\}$, where l is a positive integer, such that:

1. Each $C_i \in CP$ is a clique in G.
2. For every edge $uv \in E$: $u, v \in C_i$ for exactly one $C_i \in CP$.

When G is a complete graph, based on the sizes of its possible edge-clique partitions it makes sense to distinguish between: *trivial* edge clique partitions and *non-trivial* edge clique partitions for G.

Trivial Edge Clique Partition: A *trivial* edge clique partition of G is denoted as $CP = \{C\}$ where $C = V(G)$. A *trivial* edge clique partition is always of size one.

Non-trivial edge clique partition: A *non-trivial edge clique partition* of G is a clique partition of size at least two, that is an edge clique partition that covers all edges in G with two or more cliques.

Vertex Clique Partition: Given a graph $G = (V, E)$, a *vertex clique partition* of G is a set $VP = \{C_1, C_2, ..., C_l\}$, where l is a positive integer, such that:

1. Each $C_i \in VP$ is a clique in G.
2. For every $v \in V$: $v \in C_i$ for exactly one $C_i \in VP$.

Edge Clique Cover: Given a graph $G = (V, E)$, an *edge clique cover* of G is a set $CC = \{C_1, C_2, ..., C_l\}$, where l is a positive integer, such that:

1. Each $C_i \in CC$ is a clique in G.
2. For every $uv \in E$: $u, v \in C_i$ for some $C_i \in CC$.

Annotated Edge Clique Cover: Given a graph $G = (V, E)$ and a set $A \subseteq E$, an *annotated edge clique cover* for G and A is a set $ACC = \{C_1, C_2, \ldots, C_l\}$, where l is a positive integer, such that

1. each $C_i \in ACC$ is a clique in G and
2. for each edge $uv \in E - A$, $u, v \in C_i$ for some $C_i \in ACC$.

Vertex Clique Cover: Given a graph $G = (V, E)$, a *vertex clique cover* of G is a set $VC = \{C_1, C_2, ..., C_l\}$, where l is a positive integer, such that:

1. Each $C_i \in VC$ is a clique in G.
2. For every $v \in V$: $v \in C_i$ for some $C_i \in VC$.

Note that for every graph $G = (V, E)$, every or G is also an or G. Furthermore, every or G is also a or G.

After having introduced clique properties in graphs, we next introduce some other necessary terminology used in this document.

Dominating Set: Given a graph $G = (V, E)$, a *dominating set* is a subset $V' \subseteq V$ such that for every $v \in V$, there exists a vertex $v' \in V'$ with $v' \in N[v]$.

Polynomial-time Reduction: Given decision problems X and Y, a *polynomial-time reduction* from X to Y is a polynomial-time algorithm A that converts any instance I of X into an instance $I' = A(I)$ of Y, such that I' is a yes-instance for Y if and only if I is a yes-instance for X.

Class NP (class of *non-deterministic polynomial time* problems): NP is the complexity class of all decision problems whose *yes-answers* can be *verified* in polynomial time.

NP-hard problems are the decision problems that are at least as hard as the hardest problems in NP.

NP-hardness: A problem is *NP-hard* if every problem in NP can be reduced to it in polynomial-time.

NP-complete problems are the hardest problems in NP.

NP-Completeness: A decision problem is called *NP-complete* if it is NP-hard and a member of NP.

Parameterized decision problem: A *parameterized* decision problem is denoted $< X, k >$. Here X denotes the decision problem and k denotes its parameter.

Parameterized Complexity Class FPT (Fixed-parameter tractability): A parameterized decision problem k-$\mathcal{P}$ is *fixed-parameter tractable* (or a member of FPT) if there exists an algorithm A that solves every instance $I =< X, k >$ of k-$\mathcal{P}$ in running time $O(|X|^{O(1)} + f(k))$ or $O(|X|^{O(1)} \cdot f(k))$. A is called a *fixed-parameter algorithm.*

Reduction Rules: Given a parameterized decision problem k-$\mathcal{P}$, a *reduction rule* for k-$\mathcal{P}$ is a polynomial-time mapping from a given parameterized instance I to a new parameterized instance I', where I' is a yes-instance for k-$\mathcal{P}$ if and only if I is a yes-instance for k-$\mathcal{P}$.

We illustrate reduction rules using the example of Vertex Cover, a classic NP-complete problem [7]. Its most famous parameterized variant is described as follows.

k-Vertex Cover
Instance: A graph $G = (V, E)$, a positive integer k
Parameter: k
Question: Does there exist a set $V' \subseteq V$ with $|V'| \leq k$ such that every edge in G is *covered* by V' (i.e., for each edge $uv \in E$, at least one of u and v is in V')? V' is also called a *vertex cover* (of size at most k) for G.

Reduction Rule 1. *Let $G = (V, E)$ be a graph that contains a vertex $v \in V$ of degree 0. Then G has a vertex cover of size k if and only if $G - v$, that is G with v removed, has a vertex cover of size k also.*

Proof. This rule is correct since including v into the vertex cover would not benefit as v is not incident to any edges. Identifying and removing such a vertex v can be done in polynomial time. □

Reduction Rule 2. *Let $G = (V, E)$ be a graph that contains vertices $v \in V$ with degree larger than k. Then G has a vertex cover of size k if and only if $G - v$ has a vertex cover of size $k - 1$.*

Proof. For correctness of this rule observe that not including v into the vertex cover would force all of $N(v)$ be in the vertex cover, which is impossible since $|N(v)| > k$ and therefore the resulting vertex cover would be too large. Identifying and removing such a vertex v, and adjusting parameter k can be done in polynomial time. □

Reduction Rule 3. *Let $G = (V, E)$ be a graph that contains two adjacent vertices $u, v \in V$ with $N(v) \subseteq N[u]$. Then G has a vertex cover of size k if and only if $G - u$ has a vertex cover of size $k - 1$.*

Proof. To see that this rule is correct note the following: Assume that vertex u is not part of a k-vertex cover for G. Then all the neighbors of u must be part of the solution. Then v and its neighbors in $N(v) - u$ must be part of the k-vertex cover for G. Swapping v and u would result in a k-vertex cover that is no worse. Rule 3 can be executed in polynomial time since testing containment of neighborhoods for each adjacent pair of vertices can be done in linear time each.

□

For completeness we also mention the following rule. Its soundness proof can be found in [7].

Reduction Rule 4. *Let $G = (V, E)$ be a graph that contains a vertex x with neighbors $N(x) = \{a, b\}$ and none of the above rules applies to G. Then update G as follows: Delete x, add edge ab, and add all possible edges between the vertices in $\{a, b\}$ and $N(a) \cup N(b)$.*

Note that if none of the above reduction rules applies to G, no vertex of degree 0, 1 or 2 is left in G, that is every vertex in G has degree at least three and at most k.

Kernelization [18]: Given a parameterized decision problem, a *kernelization* is a polynomial-time reduction that maps a given instance $I =< G, k >$ to a new instance $I' =< G', k' >$, such that:

1. $k' \leq k$.
2. $|G'| \leq g(k)$ for some computable function $g(k)$.
3. I' is a yes-instance if and only if I is a yes-instance.

I' is called *(problem) kernel.* If $g(k)$ is polynomial in k then we call I' a *polynomial (sized) kernel.*

We point out that simply applying reduction rules 1–3 for the Vertex Cover problem until no such rule applies any longer, shrinks the graph in polynomial time to a polynomial kernel. Removing all degree-0 vertices using Rule 1 can be done in time $O(|V|)$. The other two rules need, all together, be run at most k times, since otherwise the smallest vertex cover for G would be larger than k, which in turn would result in $< G, k >$ being a no-instance for k-Vertex Cover. After reducing the graph as described, every vertex is of degree at least 2 and at most k[1]. Thus, if G has a vertex cover of size k then the graph can have at most $k(k + 1)$ many vertices. Therefore, if the reduced graph has more vertices than $k(k+1)$ many, the instance is a no-instance.

Bounded Search Tree [5]: The *technique of bounded search trees* is an algo-rithm-design technique that solves any instance $< X, k >$ to parameterized decision problem

[1]When the instance is reduced using all four reduction rules described above, every vertex in the graph is of degree at least three and at most four.

k-$\mathcal{P}$ by using a systematic enumeration of candidate solutions in fixed-parameter-tractable time. The resulting search tree is bounded in size in a function $g(k)$ that depends on parameter k only. The root of the (bounded) search tree corresponds to the original problem $< X, k >$.

Typically, when applying the technique of bounded search trees, the problem is solved recursively by branching according to a constant number of different cases per search-tree node. However, note that the number of branches can be a function of the problem parameter k. The size $g(k)$ of the bounded search tree can be described as a function that depends on parameter k and its largest *branching number*, defined below.

Branching Vector and Branching Number: When applying the technique of bounded search trees for a parameterized decision problem k-$\mathcal{P}$, whenever branching is applied to an instance $I =< X, k >$ of k-$\mathcal{P}$, the instances resulting from the branching are represented as the children of I in the bounded search tree. If a branching rule applied to I creates h children and reduces the parameter of their respective instances to $k - d_1, k - d_2, ...$ and $k - d_h$, then this branching has *branching vector* $(d_1, d_2, ..., d_h)$.

Branching vector $(d_1, d_2, ..., d_h)$ corresponds to the recursion $T_k = T_{k-d_1} + T_{k-d_2} + \cdots T_{k-d_h}$ and its corresponding characteristic polynomial is $z^d = z^{k-d_1} + z^{k-d_2} + ... + z^{k-d_h}$, where $d = max\{d_1, d_2, ..., d_h\}$. If $\alpha \in \mathcal{R}$ is a root of maximum absolute value solving this characteristic polynomial, we call α the *branching number* of branching vector $(d_1, d_2, ..., d_h)$. Furthermore, $t_k = O(\alpha^k)$ is the size of this search tree described by this branching.

As an example, consider the following branching taking from the bounded-search-tree algorithm for k-Vertex Cover described in, e.g., [7]:

Branching Rule 1. *Let $G = (V, E)$ be a reduced graph for k-*Vertex Cover *for parameter k. Then for any vertex v in G branch as follows. For branch 1, include u into the vertex cover, remove u from G and reduce k by 1. For branch 2, include $N(u)$ into the vertex cover, remove all vertices in $N(u)$ from G and reduce k by $|N(u)|$.*

Proof. Soundness of this branching rule follows from the following observation: vertex u either is part of a k-vertex cover (in case a solution exists), or it is not. In the latter case, to cover all vertices incident to u without using v every neighbor of u must be included into the vertex cover. □

Consider an algorithm solving k-Vertex Cover via first reducing the instance according to reduction rules 1–4 and then branching as described in Branching Rule 1. For each created instance we again apply the four reduction rules as long as possible, before we continue to branch then. In this case, since $|N(u)| \geq 3$, the branching vector for this search tree is $(1, 3)$. Thus the search is of size at most $O(1.4656^k)$.

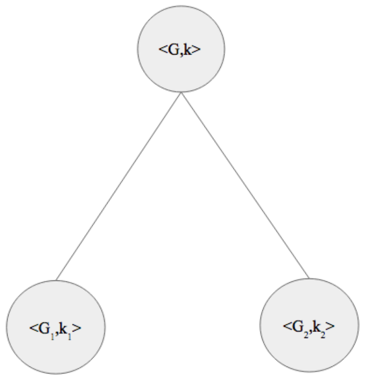


Figure 2.1: Illustration of Branching Rule 1. $G_1 = G - \{u\}$, $k_1 = k - 1$ and $G_2 = G - N(u)$, $k_2 = k - |N(u)|$ with $|N(u)| \geq 3$.

Chapter 3

Problem Definitions

In this chapter, we list the definitions of graph-based parameterized decision problems that we discuss in this report. Many of their classic versions can be found in [11].

Among those problems, k-Clique, k-Vertex Clique Partition, k-Compression Clique Cover, k-Clique Editing, k-Clique Deletion, and k-Dominating Clique are in the class of FPT according to reference [8]. Both k-Edge Clique Partition and k-Edge Clique Cover are shown to be NP-complete [20].

k-Clique [8]

Input:	A graph $G = (V, E)$, a positive integer k
Parameter:	k
Question:	Does there exist a clique C for G of size at least k?

The non-parameterized version, Clique, is NP-complete [11]. k-Clique is in FPT in [8].

k-Edge Clique Partition [18]

Input:	A graph $G = (V, E)$, a positive integer k
Parameter:	k
Question:	Does there exist an edge clique partition CP for G of size at most k?

The non-parameterized version, Edge Clique Partition, is NP-Complete [20], and an FPT algorithm to solve k-Edge Clique Partition is given in [18].

k-Vertex Clique Partition [8]

Input: A graph $G = (V, E)$, a positive integer k

Parameter: k

Question: Does there exist a vertex clique partition VP for G of size at most k?

The non-parameterized version, Vertex Clique Partition, is NP-Complete [6]. Note that this problem is also called Clique Partition in the literature.

k-Edge Clique Cover [14]

Input: A graph $G = (V, E)$, a positive integer k

Parameter: k

Question: Does there exist an edge clique cover CC for G of size at most k?

The non-parameterized version, Edge Clique Cover, is NP-Complete [20], and an FPT algorithm to solve its annotated version is given in [14].

k-Annotated Edge Clique Cover [14]

Input: A graph $G = (V, E)$, a set $A \subseteq E$, a positive integer k

Parameter: k

Question: Does there exist an annotated edge clique cover ACC for G and A of size at most k?

k-Annotated Edge Clique Cover is in the same complexity class as k-Edge Clique Cover. An FPT algorithm is given in [14].

k-Vertex Clique Cover [8]

Input: A graph $G = (V, E)$, a positive integer k

Parameter: k

Question: Does there exist a vertex clique cover VC of size at most k?

The non-parameterized version, Vertex Clique Cover, is NP-Complete. See [16], page 95.

k-Compression Clique Cover [8]

Input: A graph $G = (V, E)$, an edge clique cover CC for G of size $k + 1$ for G and a positive integer k

Parameter: k

Question: Does there exist an edge clique cover CC' for G of size at most k?

k-Compression Clique Cover is shown in FPT [8], page 616.

k-Cluster Editing [8]

Input: A graph $G = (V, E)$, a positive integer k

Parameter: k

Question: Is it possible to edit graph G into a vertex clique partition using at most k operations, consisting of edge additions and edge deletions?

k-Cluster Editing is shown to be in FPT [8].

k-Cluster Deletion [8]

Input: A graph $G = (V, E)$, a positive integer k

Parameter: k

Question: Is it possible to use at most k edge deletions to edit graph G into a a vertex clique partition?

k-Cluster Deletion is shown to be in FPT [8].

k-Dominating Clique [8]

Input: A graph $G = (V, E)$, a positive integer k

Parameter: k

Question: Does there exist a *dominating clique* $DC \subseteq V$ of size at least k such that DC is both a dominating set and a clique in G?

k-Dominating Clique is shown to be in FPT [8].

Chapter 4

Exact Algorithms for two Clique-based Problems

As mentioned in the previous chapter, k-CLIQUE, k-EDGE CLIQUE PARTITION and k-EDGE CLIQUE COVER are all NP-complete and as such there are no known algorithms to solve them in polynomial time. Their natural parameterizations, however, do not share the same complexity: while k-CLIQUE, the classic CLIQUE problem parameterized by the clique size k, is W[1]-complete, both k-EDGE CLIQUE PARTITION and k-EDGE CLIQUE COVER are in FPT.

In this chapter we consider fixed-parameter algorithms for both of these problems, using techniques from the FPT toolkit including kernelization and the technique of bounded-search-trees. We assume that the vertices and edges of input graphs are stored in adjacency lists where we assume that the indices of the array storing the adjacency lists corresponds to the names of the vertices in G (see Figure 4.1,). Thus, in the list starting at index u, there exists a node with value v if and only if $uv \in E$. Note that therefore, it takes time $O(1)$ to find a particular vertex and time $O(n)$ to search for a particular edge.

We use hash tables to determine the intersection of the neighbourhoods of two vertices u and v (described below). Note that this can be done in running time $O(n)$:

1. Initialize a hash table ht, an empty set R of vertices.
2. Push all vertices $N[u]$ onto ht.

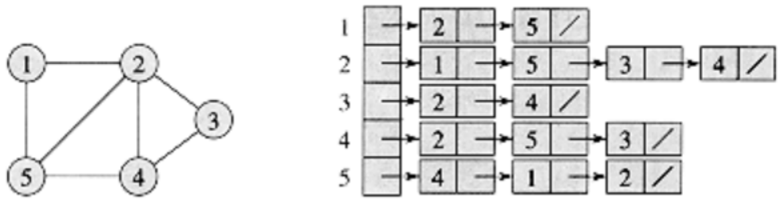


Figure 4.1: A graph $G = (V, E)$ and its corresponding adjacency list.

3. For every $w \in N[v]$, if w in ht, append w to R.
4. Return $R = N[u] \cap N[v]$.

4.1 Algorithms for k-Edge Clique Partition

When given an input to k-Edge Clique Partition, in a solution CP of size k for G, both of endpoints of each edge of the input graph are included in exactly one of the cliques in CP. This property constrains the times of repetition of vertices in a solution for an instance of k-Edge Clique Partition.

We describe a fixed-parameter algorithm originally suggested by Mujuni and Rosamond [18]. The algorithm solves k-Edge Clique Partition using a number of polynomial-time reduction rules and a bounded search-tree algorithm. The polynomial-time reduction rules reduce a given an instance $I =< G, k >$ with $|V| = n$ to a polynomial problem kernel that contains at most k^2 vertices, while the bounded search-tree algorithm leads to a final answer to the question whether or not instance I is a yes-instance. We also calculate an upper bound running time of the presented FPT algorithm. We note that the running time in [18] for k-Edge Clique Partition is said to be $O(n^3k + 2^{k^2}n)$. We could not verify this and instead present a running time analysis resulting in $O(kn^3 + (2^{k^2-2})^k(2^{k^2-2})k^4)$.

4.1.1 Reduction Rules and a k^2-Kernel for k-Edge Clique Partition

Following are four reduction rules for k-Edge Clique Partition [18]. We also prove their correctness, which is not provided with details in [18].

Reduction Rule 1. *Let $I =< G, k >$ be an instance of k-*Edge Clique Partition*. If there exists a vertex $v \in V$ with $deg(v) = 0$, then remove v from G.*

Proof. Given an instance of k-Edge Clique Partition, denoted as $I =< G, k >$, let v be a vertex in G with $deg(v) = 0$. Furthermore, let $I' =< G - v, k >$ be the instance obtained by removing v from I. We prove that I is a yes-instance if and only if I' is a yes-instance.

First, we show that if I is a yes-instance then I' is a yes-instance. Assume CP is an edge clique partition of size k for I. Since v is a singleton in G, $v \notin C_i$ for any $C_i \in CP$. Therefore, CP covers every edge in $G - v$ and is a solution of size k for instance I'.

We next prove that if I' is a yes-instance then I is a yes-instance. Assume I' is a yes-instance and CP' is a solution of size k for I'. Since $E' = E$, CP' is also a solution for I. □

For the following reduction rule we assume that Rule 1 is not applicable to instance I.

Reduction Rule 2. *Let $I =< G, k >$ be an instance of k-*Edge Clique Partition*. If there is a vertex $v \in V$ with $deg(v) = 1$, then remove v from G and decrease k by one.*

Proof. Given an instance of k-Edge Clique Partition $I =< G, k >$, let v be a vertex in G with $deg(v) = 1$. Furthermore, let $I' =< G - v, k - 1 >$ be the instance obtained by removing v from I. We prove that I is a yes-instance if and only if I' is a yes-instance.

First we show that if I is a yes-instance then I' is a yes-instance. Let $N(v) = \{u\}$. Let CP be an edge clique partition for I, where CP is of size k. There is a clique $C_i \in CP$ such that $C_i = \{u, v\}$, since vertices u, v have no common neighbours and edge uv

cannot be covered by a clique C' such that $C_i \subset C'$. Therefore $CP' = CP - \{C_i\}$ is an edge clique partition for I' of size at $k-1$, since $G' = G - v$.

We next prove that if I' is a yes-instance then I is a yes-instance. Let CP' be a solution of size at most $k-1$ for I'. Then $CP = CP' \cup \{C\}$, where $C = \{u, v\}$, is a solution for I, since uv is the only additional edge to be covered in I and uv can be covered by clique C. □

For the following reduction rule we assume that neither Rule 1 nor Rule 2 can be applied to instance I.

Reduction Rule 3. *Let $I =< G, k >$ be an instance of k-*Edge Clique Partition*. If there exists an edge $uv \in E$ with $N(u) \cap N(v) = \emptyset$, then remove edge uv from G and decrease k by one.*

Proof. Given $I =< G, k >$, $G = (V, E)$, with $uv \in E$ with $N(u) \cap N(v) = \emptyset$, let $I' =< G - uv, k - 1 >$ be the instance obtained by removing uv from I. We prove that I is a yes-instance if and only if I' is a yes-instance.

First we show that if I is a yes-instance then I' is a yes-instance. Assume CP is a solution for instance I of size at most k. Because $N(u) \cap N(v) = \emptyset$, there is a $C_i \in CP$ with $C_i = \{u, v\}$. Then $CP' = CP - \{C_i\}$ is a solution of size at most $k-1$ for I'.

We next prove if I' is a yes-instance then I is a yes-instance. Let CP' be a solution of size at most $k-1$ for I'. Then $CP = CP' \cup \{C\}$, where $C = \{u, v\}$, is a solution of size at most k for I since uv is the only additional edge to be covered in I and uv can be covered by clique C. □

In order to prove the correctness of the next reduction rule, Rule 4 described below, we consider the following lemma. For this, recall that a non-trivial edge clique partition of a complete graph is of size at least two (Section 2).

Lemma 1. *Let $G = (V, E)$ be a complete graph with $n > 2$ vertices. Then the smallest non-trivial edge clique partition of G is of size at least n.*

Proof. We prove this lemma by induction on n, the number of vertices in the graph.

Hypothesis: let G be a complete graph with $n > 2$ vertices. Any non-trivial edge clique partition for G is of size at least n.

Base Case: Let $n = 3$, assume vertices in G are v_1, v_2, v_3. The smallest non-trivial edge clique partition for G is $\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_1, v_3\}\}$.

Induction Step: Assume G^* is a complete graph of size n and G' is a complete graph of size $n + 1$, such that $G^* = G' - v$. We want to prove if non-trivial edge clique partition CP^* for G^* is of size at least n, then CP' for G' is of size at least $n+1$. We will prove this by contradiction.

We first show when given a CP' of size h for G', we can build a CP^* of size equal to or smaller than h for G^*.

For any clique $C' \in CP'$, removing any vertex v from C' still yields a clique, namely $C^* = C' - v$. Furthermore, removing v from G' and from all cliques in partition CP' does not create any edges that are not partitioned by CP^*. Therefore $|CP^*| \leq |CP'| = h$.

Note that we define the smallest clique to be of size two. Therefore, to remove vertex v from C' such that $v \in C'$ and $|C'| = 2$, we remove C from CP' instead.

Now we will show the contradiction. Let us assume that CP^* for G^* is of size at least n and that CP' can be of size $h < n + 1$. We consider two cases: $h < n$ and $h = n$.

Case 1: $h < n$. By removing v from every clique in CP' we obtain a non-trivial edge clique partition CP^* for G of size smaller than n, which contradicts the hypothesis that any non-trivial edge clique partition for G^* is of size at least n.

Case 2: $h = n$. Given statements A and B, if statement A implies B, but statement B implies not A, then using the logic computation, we will get that A implies not A, which is a contradiction.

Here our statement A is that there exists a CP' of size n for G' and our statement B is that there exists a CP^* of size n for G^* by removing v from every clique of CP'.

We have proved that A implies B already. Now we only need to prove B implies not A.

Let C^* be a clique in CP^* such that $C^* \cup \{v\}$ is in CP', if we can prove that we need at least $n-1$ cliques to cover out-going edges from C^* in G, then CP' is of size at least $n+1$, since we need to also include C^* and at least one additional clique in CP' to partition edges which are not incident to C^*.

Note that since $G^* = G' - v$, so C^* is a clique in G' as well. Every cliques in clique partition is of size at least two, so let $|C^*| = j$ such that $j \geq 2$.

There are $n-j$ vertices in $V' - C^* - v$, so the total number of out-coming edges (those not connected to v) from C^* are $(n-j) * j$. Assume $v_1^*, v_2^* \in C_i^*$. There are $n-j$ edges connecting v_1^* and other $n-j$ vertices in $V' - C^* - v$. Denote them with a set E_1. And there are also $n-j$ edges connecting v_2^* and other $n-j$ vertices in $V' - C^* - v$. Denote them with a set E_2. Note that $E_1 \cap E_2 = \emptyset$.

For every pair of edges $e_1 \in E_1$ and $e_2 \in E_2$, e_1 and e_2 is not covered by the same clique, otherwise pair of vertices v_1^* and v_2^* are in two different cliques of CP^*.

Let's consider two different vertices $f, g \notin C^i*$ and following edges:

1. e_{1f} connects vertices v_1^* and f.
2. e_{1g} connects vertices v_1^* and g.
3. e_{2f} connects vertices v_2^* and f.
4. e_{2g} connects vertices v_2^* and g.

If we use one clique $C'_0 = v_1^*, f, g$ to cover edges e_{1f} and e_{1g}, then it takes two cliques to cover edges e_{2f} and e_{2g}, since f, g are both in C'_0 and cannot appear together in any clique which is not C'_0.

For this reason, in a minimum solution CP' it takes exactly one clique to cover out-going edges from vertex v_1 and it takes one additional clique to cover each of the remaining out-going edge from C^*.

Therefore to cover all the out-going edges from C_i^*, we need at least $f(j) = 1 + (j - 1)(n - j) = 1 + \frac{n^2-1}{4} - (j - \frac{(n+1)^2)}{2})$ cliques. $f(j)$ takes its minimum value $n - 1$ when $j = 2$. Then we need to include C^* and at least one additional clique to cover edges not incident to vertices of C^*, so the size of CP' is at least $n + 1$, which is a contradiction to our assumption.

□

For the following reduction rule we assume that none of Rules 1–3 can be applied.

Reduction Rule 4. *Let* $I =< G, k >$ *be an instance of* k-Edge Clique Partition. *If there exists a vertex* $v \in V$ *with* $|N[v]| > k$ *where* $N[v]$ *induces a clique, then remove* $E(N[v])$ *from* G*, and reduce* k *by one.*

Proof. Assume v is a vertex in G such that $|N[v]| > k$ and vertices in $N[v]$ induces a clique. Furthermore, let $I' =< G - E(N[v]), k - 1 >$ be the instance obtained from applying Rule 4 to on I. We prove that I is a yes-instance if and only if I' is a yes-instance.

First we show that if I is a yes-instance then I' is a yes-instance. Assume CP is a solution of size at most k for I. Further assume C is a clique in G where $|C| > k$. Then the subgraph induced by C, denoted as $G(C)$, is a complete graph. According to Lemma 1, if we want to do a non-trivial partition on $G(C)$, we need at least $k+1$ cliques. Therefore the only way to cover all the edges in $E(C)$ is to include C as a member in CP.

Therefore, if there is a vertex $v \in V$ with $|N[v]| > k$ where vertices in $N[v]$ induces a clique, then $C_i = N[v]$ is a member of CP. Then $CP - \{C_i\}$ partitions all edges in $E - E(N[v])$, and $CP' = CP - \{C_i\}$ is a solution of size at most $k - 1$ for I'.

We next prove that if I' is a yes-instance then I is a yes-instance. Given instance $I' =< G - E(N[v]), k - 1 >$, let CP' be a solution of size at most $k - 1$ for I'. Then $CP = CP' \cup \{N[v]\}$ is a solution of size at most k for $I =< G, k >$ since clique $N[v]$ partitions all the edges in $E(N[v])$. □

We now prove the running time of above reduction rules, and then prove the size of problem kernel when none of these rules is applicable.

Lemma 2. *In time $O(kn^3)$, one can use Rules 1–4 to generate a reduced instance where none of these rules applies.*

Proof. First, we only consider the application of Rule 1. Given a singleton in G, it takes constant time to remove it from G. The removal of singletons does not decrease the degree of any vertex in G. Therefore to determine all singletons in a given graph G (and their removal) can be done in time $O(n)$.

Then we consider the running time of applying Rule 1 and Rule 2 together using below algorithm:

1. Initialize an empty queue Q.
2. Inspect $deg(v)$ for all $v \in V$. If $deg(v) = 0$, remove v from G. If $deg(v) = 1$, add v to Q.
3. Dequeue Q's first vertex v', remove v' from G and reduce k by one. Assume u' is the only neighbor of v'. Update $deg(u')$ after the removal of v'. If $deg(u') = 0$, remove u' from Q and G. If $deg(u') = 1$, add u' to the rear of Q. Stop when Q is empty or k decreases to zero.

In above algorithm, we look up the degree of a vertex at most $2n$ times in total. The first n times happen when we look up the degree of every vertex and en-queue the degree-one vertices into Q. Then when removing a vertex v' from a vertex in Q, we look up the degree of its neighbor u', where u' is not in queue Q. Note that there are at most n vertices in Q. Therefore, we need to inspect at most n vertices in step 3. Overall, all look ups of degrees of vertices can be done in linear time.

After applying Rule 1 and Rule 2 in time $O(n)$, assuming that none of the Rules 1 and 2 can be applied any longer, there are neither singletons nor degree-one vertices in I.

We now consider the application of Rule 3.

To apply Rule 3, we need to determine an edge uv with $N(u) \cap N(v) = \emptyset$. For each edge $uv \in E$, we obtain $N(u)$ and $N(v)$ directly from the adjacency list of the graph, which takes time $O(n)$. Determining the intersection of $N(u)$ and $N(v)$, which can be done in linear time, e.g. by using a hash table. Thus, it takes time $O(n)$ to determine whether a given edge uv can be removed using Rule 3. The removal of edge uv itself takes time $O(1)$. Inspecting all m edges in E for applicability of Rule 3, and applying the rule, takes time $O(m) * O(n) = O(mn)$.

Note that we only need to inspect all $uv \in E$ once to determine and remove all uv with $N(u) \cap N(v) = \emptyset$ from G, because the application of Rule 3 does neither remove any vertex from G nor decreases $|N(u') \cap N(v')|$ for any edge $u'v' \in E$ such that $u'v' \neq uv$. Therefore, each application of Rule 3 takes time $O(mn)$.

Note that nor singleton or degree-one vertex can be a common neighbour of vertices u and v when $uv \in E$. Therefore, after applying Rule 3, and when Rule 3 is not applicable, we require an extra $O(n)$ time to re-apply Rule 1 and Rule 2. Only now we are guaranteed that none of Rules 1,2 and 3 applies to the reduced graph.

Therefore the overall running time to apply Rules 1–3 is $O(n^3)$.

Now let us assume that none of Rules 1–3 applies. We consider Rule 4.

It takes time $O(n)$ to find a vertex v with $|N[v]| > k$ from G by inspecting the degree of each $v \in V$. To determine if the vertices in $N[v]$ induce a clique takes time $O(n^2)$ (by checking whether there is an edge between every pair of vertices in $N[v]$). Since there are n vertices in G, it takes time $O(n) * O(n^2) = O(n^3)$ to determine a vertex that satisfies the preconditions of Rule 4 in I. There are at most n^2 edges in $E(N[v])$. Therefore the removal of the edges in $E(N[v])$ takes time $O(n^2)$.

Figure 4.2 illustrates that Rule 4 might be applicable many times: If we assume that we inspect vertex v_2 before v_4 when applying Rule 4, the first time we inspect vertex v_2, we note that $|N[v_2]| = k$, that is Rule 4 is not applicable at this time. Then we inspect v_4, which meets all the requirements to apply Rule 4. When applying Rule 4, we remove $E(N[v_4])$ and reduce k by one. As it is shown in Figure 4.3, now $E(N[v_4])$ is removed and k is reduced to two. Therefore at this time, Rule 4 can be applied on v_2 since $|N[v_2]| > k$ and vertices in $N[v_2]$ induce a clique.

Thus, we need to inspect all $v \in V$ again whenever Rule 4 is applied. Thus, the overall running time to detect and apply all Rule 4 cases is $O(n^3 k)$.

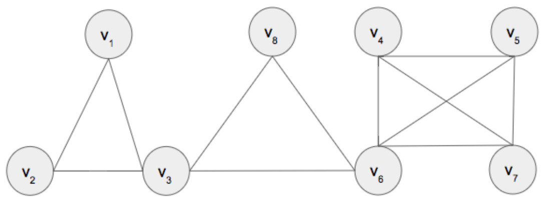


Figure 4.2: Given an instance $I =< G, k >$ with $k = 3$, Rule 4 is applicable on v_4 but not v_2.

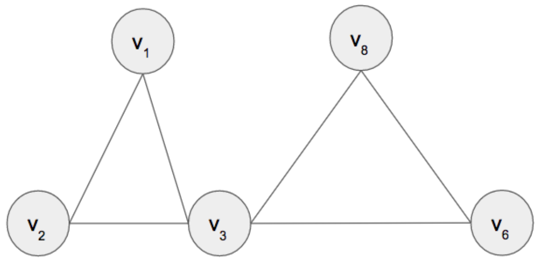


Figure 4.3: After the removal of v_4 and $E(N[v_4])$ in I, k is reduced by one. Rule 4 now can be applied on v_2.

Rule 4 has no influence on the applicability of Rule 3 because no vertex is removed in Rule 4. Rule 1 and 2 have no influence on the applicability of Rule 3 neither, which is proved before. Therefore when Rule 4 is not applicable, we keep applying Rule 1,2,4 until none of them is applicable.

Every time we apply Rule 2 and 4, k is reduced by one. Therefore we can apply them at most k times in total. Among all four reduction rules, the time complexity of Rule 4 is the highest. In the worst case, we apply Rule 4 k times, which takes time $O(kn^3)$.

$\square$

We call an instance $I =< G, k >$ for k-Edge Clique Partition *reduced* if none of

Rules 1–Rule 4 is applicable.

In general, we call an algorithm that transforms any parameterized instance I into a parameterized instance I' a *kernelization* if the size of instance I' is bounded by a function that only relies on parameter k. We also call I' a *kernelized* instance.

Lemma 3. *Let $I =< G, k >$ be a reduced instance of k-*Edge Clique Partition*. Then I is of size at most k^2 [18].*

Proof. Assume there are more than k^2 vertices in instance $I =< G, k >$ where Rule 1 to Rule 4 do not apply. Because Rule 1 is not applicable, there is no singleton in G. That is, every vertex is incident to some edge $e \in E$. Because Rule 4 is not applicable, all the cliques in G are of size at most k. By including k cliques into CP, we can only put at most k^2 vertices in the partition. Furthermore, if there are more than k^2 vertices of degree ≥ 2, there is at least one vertex v that is not in any clique $C \in CP$. Therefore v's incident edge e_0 is not induced by a clique in partition CP. Thus, I is a no-instance.

□

4.1.2 A brute force algorithm and a bounded search tree algorithm for Edge Clique Partition

When none of the above four reduction rules can be applied, using a brute force algorithm to search for the solution of the reduced instance, can looks as follows: Try all combinations for possible edge clique partitions of size k and test their validity. For each such partition, we arrange at most n^2 pairs of vertices (which correspond to edges in the graph) into a set $H = \{C_1^h, C_2^h, \ldots, C_k^h\}$. If each subset $C_i^h \in H$ is a clique and for every edge xy $x, y \in C_i^h$ for some i, $1 \leq i \leq k$, then H is an edge clique partition for the reduced instance I.

There are at most $S_{(n^2,k)}$ ways to construct such set H, where $S_{(n^2,k)}$ is the number of ways to put "n^2 balls into k identical bins". To determine whether each C_i is a clique can be done in time $O(n^2)$. To verify whether each H is an edge clique partition, we verify that each $C_i \in H$ is a clique. This takes time $O(kn^2)$. Therefore it takes time

$O(S_{(n^2,k)}kn^2)$ to obtain the solution for the reduced instance. Since in the reduced instance $n < k^2$, the total running time is $S_{(k^4,k)}O(k^5)$.

To improve the running time, we provide a bounded search-tree algorithm below to determine our final answer for the obtained kernelized instance of size at most k^2.

Algorithm 1 Bounded Search-Tree Algorithm for a kernelized instance of k-Edge Clique Partition

Input: A graph $G = (V, E)$, an initially empty set CP to store cliques, an integer k.
Question: Is there an edge clique partition of size at most k for G?

```
1:  function $C_{Partition}(G, CP, k)$
2:      if $E = \emptyset$ then return TRUE
3:      else if $k = 0$ then return FALSE
4:      else
5:          choose $uv \in E$ such that $|N(u) \cap N(v)|$ is minimum
6:          find a set $S$ of all cliques in $N(u) \cap N(v)$
7:          for each $K \in S$ do
8:              $K' = K \cup \{u, v\}$
9:              $CP' = CP \cup \{K'\}$
10:             $k' = k - 1$
11:             $G' := G - E(K')$
12:             if $C_{Partition}(G', CP', k')$ then return TRUE
13:     return FALSE
```

Lemma 4. *Algorithm 1 solves the sized k^2 kernelized instance of k-*Edge Clique Partition *in the running time of $O((2^{k^2-2})^k 2^{k^2-2} k^4)$.*

Proof. Assume $I =< G, k >$ is the kernelized instance of k-Edge Clique Partition. According to Lemma 3 there are at most k^2 vertices in graph G. We apply Algorithm 1 to search for the solution to $I =< G, k >$, where $G =< V, E >$ and $|V|$ is bounded by k^2 since none of Rules 1 to 4 applies.

We first calculate the maximum height of tree. In Line 13 of Algorithm 1, we create one branch for each possible clique (to be included into the solution set) that partitions edge $uv \in E$, where u,v has minimum number of common neighbours over other edges. The maximum height of the tree T is k, because whenever we branch on any subinstance of I, k will be reduced by one for including a clique in solution CP.

Now we calculate the number of nodes in the tree. We first branch on the given instance $I =< G, k >$, and then branch recursively on I's subinstances until we find a solution or no more branches can be created. For each subinstance, we search G and find the edge uv such that $|N(u) \cap N(v)|$ is minimum. The size of $|N(u) \cap N(v)|$ is at least one since none of Rule 1 to Rule 3 applies. There are at most $k^2 - 2$ common neighbours for u and v. Therefore there are at most 2^{k^2-2} different cliques in $N(u) \cap N(v)$. We branch on this subinstance by including each possible $K \cup \{u, v\}$ in CP (see Lines 8 and 9), where K is a member of S. Therefore, there are at most 2^{k^2-2} branches for each subinstance in T.

Now we calculate the running time of processing each node (subinstance) in the tree. When processing a subinstance of I, we first check all the edges in E to determine $uv \in E$ such that $|N(u) \cap N(v)|$ is minimum (see Line 5). The running time for this operation is $O(k^6)$. Then we generate set S which stores all the cliques in $N(u) \cap N(v)$ (see Line 6). Since there are at most $k^2 - 2$ vertices in $N(u) \cap N(v)$, we obtain 2^{k^2-2} sets of vertices that have the potential to be cliques. To verify whether or not each of those sets is a clique, it takes time $O(k^4)$. Therefore, the time complexity of processing a subinstance to generate S is $2^{k^2-2}k^4$.

The height of T is k and each instance in T has at most 2^{k^2-2} branches. There are at most $(2^{k^2-2})^k$ nodes in the bounded search tree T. The time of processing a node (subinstance) in T is $2^{k^2-2}k^4$. Therefore, the upper bound for Algorithm 1 for a kernelized graph is $O((2^{k^2-2})^k 2^{k^2-2}k^4)$.

However, in [18], the running time of bounded search tree is calculated as follows.

Because of to Line 12, between two levels of the tree, k is reduced by one. Therefore, the height of the tree is at most k. Because in a reduced instance, there are at most k^2 vertices, the number of children at each node of the search tree is bounded by k^2. It takes linear time in the number of vertices to process each search tree node before branching. Therefore the total running time of Algorithm 1 is $O(2^{k^{2^k}}n)$. $\square$

The running time for the fpt algorithm that first kernelizes the graph as discussed and then uses the bounded search tree algorithm described here is composed of time $O(n^3k)$ to apply Reduction Rules 1 4 and time $O(2^{k^{2k}}n)$ for the bounded search tree algorithm. Therefore the total running time in our FPT algorithm for k-edge clique partition is $O(n^3k + 2^{k^{2k}}n)$.

4.1.3 New Lemmas and Reduction Rules for k-Edge Clique Partition

In addition to the reduction rules in [18], we developed a list of new lemmas and reduction rules to further pre-process a given instance I for k-Edge Clique Partition before applying a bounded search-tree algorithm.

Below are the new lemmas and reduction rules. Each is followed by the proof for their correctness.

Reduction Rule 5. *Let $I =< G, k >$ be an instance of k-*Edge Clique Partition*. Suppose there are three vertices $u, v, w \in V$, such that $uv, uw, vw \in E$, and $deg(u) = 2, deg(v) = 2$ and $deg(w) \geq 2$. Then $I =< G, k >$ is a yes-instance if and only if $I' =< G - u - v, k - 1 >$ is a yes-instance.*

Proof. We want to prove that I is a yes-instance if and only if I' is a yes-instance.

We first prove if I is a yes-instance then I' is a yes-instance. Let CP be a minimum solution of size at most k for I. In graph G, there is no clique C of size larger than three such that $u, v, w \in C$. Therefore, to cover edges uv, uw and vw, we need to either include clique $C_{(u,v,w)}$ or all cliques $C_{(u,w)}, C_{(v,w)}, C(u, v)$ in CP. However, including $C_{(u,w)}, C_{(v,w)}, C(u, v)$ in CP provides a larger edge clique partition. Therefore, $C_{(u,v,w)} \in CP$ and $CP' = CP - \{C_{(u,v,w)}\}$ covers all edges in graph $G' = G - u - v$. Therefore I' is a yes-instance of size at most $k - 1$.

Next we prove if I' is a yes-instance, then I is a yes-instance. Assume CP' is a solution of size at most $k - 1$ for I'. We know clique $C_{(u,v,w)}$ covers all edges incident to u and v, which are uv, uw and vw. Since CP' covers all edges in graph $G - u - v$,

$CP = CP' \cup \{C_{(u,v,w)}\}$ covers all edges in G. Therefore I is a yes-instance.

□

Reduction Rule 6. *Let $< G, k >$ be an instance of k-*Edge Clique Partition*. Suppose $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two subgraphs of G, where G_1 is a clique. If $V_1 \cap V_2 = v$, $V_1 \cup V_2 = V$, $E_1 \cap E_2 = \emptyset$ and $E_1 \cup E_2 = E$. $I =< G, k >$ is a yes-instance if and only if $I' =< G_2, k-1 >$ is a yes-instance.*

Proof. We want to prove that I is a yes-instance if and only if I' is a yes-instance.

We first prove if I is a yes-instance then I' is a yes-instance. Let CP be an minimum solution of size k for I. If we don't include $C_1 = V_1$ in CP, according to Lemma 1, we need at least $|V_1|$ cliques to partition all edges in G_1. And none of those $|V_1|$ cliques contains any vertex from $V_2 - \{v\}$. This contradicts that CP is minimum. Then $CP' = CP - \{C_1\}$ covers all edges in graph $G - G_1$, which is our G_2. Therefore CP' is a solution of size at most $k-1$ for $I' =< G_2, k-1 >$.

Then we prove if I' is a yes-instance then I is a yes-instance. Clique $C_1 = V_1$ covers all edges in G_1 since G_1 is a complete graph. Therefore $CP = CP' \cup \{C_1\}$ is a solution of size at most k for I.

□

Reduction Rule 7. *Let $I =< G, k >$ be an instance of* Edge Clique Partition*. Suppose $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are two subgraphs of G. Assume $V_1 \cap V_2 = v$, $V_1 \cup V_2 = V$, $E_1 \cap E_2 = \emptyset$ and $E_1 \cup E_2 = E$. Further assume the size of minimum edge clique partition for subgraph G_1 is k_1. $I = (G, k)$ is a yes-instance if and only if $I' =< G_2, k - k_1 >$ is a yes-instance.*

Proof. We want to prove that I is a yes-instance if and only if I' is a yes-instance.

We first prove if I is a yes-instance then I' is a yes-instance. Let CP be a minimum edge clique partition of size k for instance I and asuume CP_1 is a minimum edge clique partition for G_1. Since the only intersection of G_1 and G_2 is vertex v and CP_1 covers all edges in G_1, then $CP_2 = CP - CP_1$ covers all edges in $G - G_1$, which is our G_2. Therefore $CP_2 = CP - CP_1$ is an edge clique partition of size $k - k_1$ for G_2.

Now we prove that if I' is a yes-instance then I is a yes-instance. Let CP_2 be an edge clique partition of size at $k - k_1$ for I'. Since CP_2 covers all edges of G_2 and

CP_1 covers all edges of G_1, $CP_1 \cup CP_2$ covers all edges in $G_1 \cup G_2$, which is our G. Therefore $CP = CP_1 \cup CP_2$ is an edge clique partition of size k for I.

□

If we are given a graph G that has a small subgraph G_1 such that G_1 only intersects with the rest of G at vertex v, we can compute an edge clique partition for G_1 then remove G_1 from G and reduce k, and then use a bounded search tree algorithm on $G - G_1$. Doing this can reduce the running time compared to directly applying a bounded search running algorithm on G.

Lemma 5. *Let $I =< G, k >$ be an instance of* Edge Clique Partition. *Suppose that $S = \{G_1, G_2, ..., G_m\}$ a set of subgraphs of G. Assume $V_1 \cup V_2 \cup ... \cup V_m = V, E_1 \cup E_2 \cup ... \cup E_m = E$ and $\{v\}$ is the intersection of any pair of graphs $G_i, G_j \in S$. If $m = k$, I is yes if and only if all $V(G_1), V(G_2), \ldots V(G_m)$ are cliques.*

Proof. If all $V(G_1), V(G_2), \ldots V(G_m)$ are cliques, $CP = \{V(G_1), V(G_2), \ldots, V(G_m)\}$ is an edge clique partition of size k for I.

Assume $V(G_1)$ is not a clique. Since pairwise intersections of $G_1, G_2, ..., G_m$ are vertex v, we need at least two cliques to cover edges in $E(G_1)$ and need at least one clique to cover edges from each of $E(G_2), E(G_3), \ldots, E(G_m)$. Then we need at least $k + 1$ cliques to covers edges in G, so I is a no-instance. □

Lemma 6. *Let $I =< G, k >$ be an instance of* Edge Clique Partition. *Suppose that $S = \{G_1, G_2, ..., G_m\}$ a set of subgraphs of G. Assume $V_1 \cup V_2 \cup ... \cup V_m = V, E_1 \cup E_2 \cup ... \cup E_m = E$ and $\{v\}$ is the intersection of any pair of graphs $G_i, G_j \in S$. If $m > k$, I is a no-instance.*

Proof. Since the pairwise intersection of $G_1, G_2, ..., G_m$ are vertex v, we need at least one clique to partition edges from each of $G_1, G_2, \ldots, G_m$. Therefore we need at least $m > k$ cliques to partition edges in G and I is a no-instance.

□

Note that if $m < k$, we can apply Rule 6 to shrink the graph G in instance I.

4.2 Algorithms for k-Edge Clique Cover

Recall that when given a solution of an instance $I =< G, k >$ for k-Edge Clique Cover, for every edge $uv \in E$, u and v appear together in some clique $C_i \in CC$, where $|CC| \leq k$. In general, we call an edge uv *covered* by a clique C in G if $u, v \in C$.

Compared to k-Edge Clique Partition, a solution for an instance of k-Edge Clique Cover allows an edge $uv \in E$ to be covered by multiple cliques in the solution. Note that for a given graph $G = (V, E)$, a clique partition of size k for G is also a clique cover for G. However, a clique cover of size k for G is not guaranteed to be a clique partition for G (see Figure 4.4).

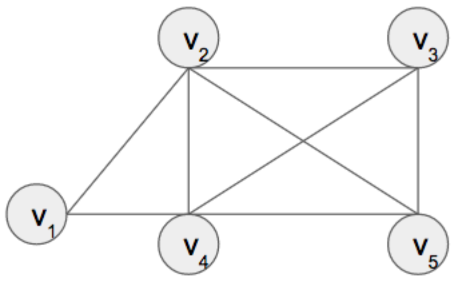


Figure 4.4: A graph $G = (V, E)$ with a minimum clique partition of size 3 and a minimum clique cover of size 2: A minimum clique partition of size 3 for G is $CP = \{\{v_1, v_2\}, \{v_1, v_4\}, \{v_2, v_3\, v_4, v_5\}\}$. CP is also an edge clique cover of size 3 for G since every edge in G is covered at least one clique in CP. However, a (minimum) clique cover for G of size 2 exists, i.e. $CC = \{\{v_1, v_2, v_4\}, \{v_2, v_3, v_4, v_5\}\}$. CC is not a clique partition for G since vertices v_2 and v_4 appear together in more than one cliques of CC.

To allow edges to be covered by several cliques in k-Edge Clique Cover, we allow the possibility of marking an edge e as *covered*. To be able to solve k-Edge Clique Cover in the case where the graph contains edges in a current instance that are marked as covered, we add an auxiliary set A to our instance I and consider solving k-Annoted Edge Clique Cover, the annotated version of k-Edge Clique Cover (defined above). Note that k-Edge Clique Cover is a special case of k-Annoted Edge Clique Cover where $A = \emptyset$.

We now describe an FPT algorithm for k-Annoted Edge Clique Cover that was introduced in [14]. The preprocessing phase of this algorithm consists of four reduction rules that reduce a given instance $I =< G, A, k >$ of k-Annotated Edge Clique Cover to a kernelized instance I' with a graph of size at most 2^k. The preprocessing is followed by a bounded search-tree algorithm that solves I'. We prove that this algorithm solves k-Annotated Edge Clique Cover in time $O(n^4 + 2^{k^{k+2}})$.

4.2.1 Initialization, Reduction Rules and a 2^k-Kernel for k-Annotated Edge Clique Cover

Before introducing reduction rules to be applied on instance $I =< G, A, k >$, we run an initialization phase that computes some useful parameters of graph G.

Initialization. For every edge $uv \in E$, we compute a set $N_{\{u,v\}}$ that stores all common neighbours of vertices u and v. We also record the size of $N_{\{u,v\}}$ for every edge $uv \in E$. Furthermore, compute $c_{\{u,v\}} = |E(N_{\{u,v\}})|$ [14].

After the initialization phase, we apply the following reduction rules to the instance [14].

Reduction Rule 8. *Let $I =< G, A, k >$ be an instance of* k-Annotated Edge Clique Cover*, and let $v \in V$ be a vertex such that $deg(v) = 0$. Then remove v from G.*

Proof. Assume $I' =< G - v, A, k >$ is an instance obtained by applying Rule 8 on I. We prove that I is a yes-instance if and only if I' is a yes-instance.

First, we show if I is a yes-instance then I' is a yes-instance. Given $I =< G, A, k >$, consider vertex v in G with $deg(v) = 0$. Assume CC is a solution for for I with $|CC| = k$. Since v has no incident edges, there is no $C_i \in CC$ with $v \in C_i$. Therefore, CC is a solution of size at most k for I'.

We now show that if I' is a yes-instance then I is a yes-instance. Given instance $I' =< G - v, A, k >$, where CC' is a solution of size k for I'. Adding any new vertex

v with $deg(v) = 0$ to G' does not create any additional uncovered edges. Therefore $CC = CC'$ is a solution of size k for I.

□

From now on we assume that Rule 8 is not applicable to instance I.

Reduction Rule 9. *If $I =< G, A, k >$ is an instance of k-*ANNOTATED EDGE CLIQUE COVER *with $A \neq \emptyset$, and where $v \in V$ is a vertex in G with all incident edges of v are elements in A, then I' is obtained by creating G' from G via removing v and updating the set of covered edges to $A' = A - \{e \in E | e \text{ is incident to } vinG\}$.*

Proof. Let $I' =< G - v, A', k >$ with $A' = A - \{e \in E | e \text{ is incident to } vinG\}$ is an instance obtained by applying Rule 9 on I for a vertex $v \in V$ in G with all incident edges of v are elements in A.

We prove that I is a yes-instance if and only if I' is a yes-instance.

First, we show if I is a yes-instance then I' is a yes-instance. Let CC be a solution of size k for $I =< G, A, k >$, and let v be a vertex in G such that all of v's incident edges are members of the set of covered edges A. We observe that for every clique $C_i \in CC$ such that $v \in C_i$, $C_i - \{v\}$ is also a clique in G. Furthermore, every edge $u'v' \in E$ with $u', v' \neq v$ is covered by a at least one clique in $CC' = \{C_i - \{v\} | C_i \in CC\}$. Therefore CC' is a solution of size at most k for I'.

We next prove that if I' is a yes-instance then I is a yes-instance. Let $I' =< G - v, A', k >$ with $A' = A - \{e \in E | e \text{ is incident to } vinG\}$ be a yes-instance. Let CC' be a solution of size k for I'. Since $E' = E - \{e \in E | e \text{ is incident to } vinG\}$ and $A' = A - \{e \in E | e \text{ is incident to } v\}$, $E - A = E' - A'$. Therefore CC' is also a solution for I.

□

From now on we assume that neither Rule 8 nor Rule 9 are applicable to instance I. Before introducing and proving correctness of rules 10 and 11, we consider the following lemma.

Lemma 7. *Let $I =< G, A, k >$ be an instance of k-*Annotated Edge Clique Cover*, and let CC be a solution of size k for I. Then there is a solution CC' for I, $|CC'| \leq |CC|$, where every member of CC' is a maximal clique.*

Proof. Assume CC is a solution for I. Consider a clique $C_i \in CC$ that is not a maximal clique—if such a member exists (if not, we are done since then $CC' = CC$ is a solution).

We build a maximal clique C_j incrementally by including v in C_i such that $v \in N(C_i)$ and $vv' \in E$ for every $v' \in C_i$. When there is no such v, C_j is maximal since it cannot be further enlarged. At each step we only add a vertex that is adjacent to every vertex of C_i, so C_j is indeed a clique.

Since $C_i \subset C_j$, C_j covers all edges in $E(C_i)$. Therefore replacing every such C_i by a maximal clique C_j leads to a new solution CC' for instance I. □

Reduction Rule 10. *Let $I =< G, A, k >$ be an instance of k-*Annotated Edge Clique Cover*. If there exists an edge uv in G such that $N_{\{u,v\}}$ induces a clique, then include all edges induced by $N_{\{u,v\}} \cup \{u, v\}$ into A and reduce k by one.*

Proof. Let $I =< G, A, k >$ be an instance of k-Annotated Edge Clique Cover with $uv \in E$ and $N_{\{u,v\}}$ induce a clique. Furthermore, let $I' =< G, A', k - 1 >$ with $A' = A \cup E(N_{\{u,v\}} \cup \{u, v\})$ be the instance obtained by applying Rule 10 on I for $uv \in E$. We prove that I is a yes-instance if and only if I' is a yes-instance.

First, we show that if I is a yes-instance then I' is a yes-instance. Given instance $I =< G, A, k >$, assume CC is a solution of size k for I with all its members are maximal cliques (Lemma 7).

Note that if $N_{\{u,v\}}$ induces a clique and $uv \in E$, then $N_{\{u,v\}} \cup \{u, v\}$ is a clique as well.

Furthermore note that $C_i = N_{\{u,v\}} \cup \{u, v\}$ is the unique maximal clique that covers edge uv, since there is no other vertex in G that is adjacent to both u and v.

Since every clique in CC is maximal and C_i is the unique maximal clique that covers edge uv, $C_i \in CC$. Since CC covers $E - A$, $CC - \{C_i\}$ covers $E - A - E(N_{\{u,v\}} \cup$

$\{u, v\}) = E - A'$. Therefore $CC' = CC - \{C_i\}$ is a solution of size at most $k-1$ for I'.

We an now prove that if I' is a yes-instance then I is a yes-instance. Let $I' =< G, A', k-1 >$ be an instance of k-Annotated Edge Clique Cover, and let CC' be a solution of size $k-1$ for I'. Since $A' = A \cup E(N_{\{u,v\}} \cup \{u, v\})$ and clique $C_i = N_{\{u,v\}} \cup \{u, v\}$ covers all edges in $E(N_{\{u,v\}} \cup \{u, v\})$, $CC = CC' \cup \{C_i\}$ is a solution of size at most k for I.

□

From now on we assume that none of the Rules 8–10 applies to instance I.

Reduction Rule 11. *Let $I =< G, A, k >$ be an instance of k-Annotated Edge Clique Cover, and let $uv \in E$ with $N[u] = N[v]$. Then include all of u's incident edges into A.*

Proof. Let $I =< G, A, k >$ be an instance of k-Annotated Edge Clique Cover, and let $uv \in E$ with $N[u] = N[v]$. Let $I' =< G, A', k >$ where $A' = A \cup \{$ all edges incident to $u\}$ be an instance obtained by applying Rule 11 to I on edge uv. We prove that I is a yes-instance if and only if I' is a yes-instance.

We first prove that if I is a yes-instance then I' is a yes-instance. Given an instance $I =< G, A, k >$, let CC be a solution of size k for I. Since $E - A'$ is a subset of $E - A$, if CC covers all edges in $E - A$ then CC covers all edges in $E - A'$. Therefore $CC' = CC$ is a solution of size k for I'.

Now we prove that if I' is a yes-instance then I is a yes-instance. Given an instance $I' =< G, A', k >$, where $A' = A \cup \{$ all edges incident to $u\}$, according to Lemma 7, there exists a solution CC' of size k for I' that is composed of maximal cliques only.

We show that for any member $C_0 \in CC'$ where every clique in CC' is maximal, if $v \in C_0$, then $u \in C_0$. We prove this statement by contradiction. Assume that there is a clique $C_0 \in CC'$ such that $v \in C_0$ but $u \notin C_0$. Because $v \in C_0$, every vertex in clique $C_0 - v$ is adjacent to v. Since $N[u] = N[v]$, every vertex in C_0 is adjacent to u as well. Therefore adding u to C_0 would result in a larger clique. This shows that

C_0 is not maximal, a contradiction to Lemma 7.

Let $C_1, C_2, \ldots, C_h \in CC'$ such that $h < k$, $v \in C_i$ for all $1 \leq i \leq h$. From above we know that $u \in C_1$, $u \in C_2$,. . ., $u \in C_h$. Since $C_1, C_2, \ldots, C_h$ cover all edges incident to v and $N[u] = N[v]$, they cover all edges incident to u as well. Therefore, $CC = CC'$ is a solution to I of size at most k.

□

We call an instance of k-Annotated Edge Clique Cover *reduced* if none of the Rules 8–11 can be applied.

Lemma 8. *Given instance I of* k-Annotated Edge Clique Cover*, in time $O(n^4)$ a reduced instance can be obtained.*

Proof. Before applying the reduction rules introduced above, we pre-process the given instance in the initialization phase and compute, for every edge in E, $N_{\{u,v\}}$ and $c_{\{u,v\}}$. We now show that this initialization can be done in time $O(n^4)$.

For each edge $uv \in E$, we use an adjacency list and a hash table (page 16) to calculate the intersection of $N[u]$ and $N[v]$. This can be done in time $O(n)$. To obtain $c_{\{u,v\}}$, we inspect every edge $e \in E$ and sum up the number of edges with both endpoints in $N_{\{u,v\}}$. This can be done in time $O(n^2)$. Therefore it takes time $O(n^2)$ to compute $N_{\{u,v\}}$ and $c_{\{u,v\}}$ for one edge $uv \in E$. Since there are at most n^2 edges in E, this step takes time $O(n^4)$ in total for the initialization phase.

Next we consider applying Rule 8 on I. To determine and remove all singletons in G, in time $O(n)$ we can inspect all $v \in V$, followed by time $O(1)$ for each singleton found to apply the removal. Note that the removal of singletons does not affect the degree of other vertices. Therefore every vertex needs to be inspected only once. Thus, given graph G using Rule 8 to generate an instance such that Rule 8 is no longer applicable takes time $O(n)$.

Now we consider the application of Rule 9. Since Rule 8 is not applicable to I, we only determine and remove a vertex v, where v is incident to edges only in A. To achieve this goal, we use a hash table to store edges in A. For each vertex $v \in V$, we obtain all its incident edges and inspect for each of them whether or not it is in A. Each inspection takes time $O(1)$, and each v is incident to at most $n-1$ edges. Therefore it takes time $O(n)$ to decide whether or not v is only incident to edges in A.

When applying Rule 9, we only remove vertex v and its incident edges. Assume a vertex $v' \in V$, where $v' \neq v$, is incident to an edge $e \notin A$. After the removal of v, v' is still incident to edge e. Therefore we only inspect every vertex once and perform the removal. This procedure takes time $O(n^2)$. Afterwards, Rule 9 is no more applicable to G.

When applying Rule 9, since some vertex $v \in V$ is removed, we need to update $N_{\{u,v\}}$ and $c_{\{u,v\}}$ for every v's neighbour u'. This can be one in time $O(n)$ since v has at most $n-1$ neighbours.

Note that the removal of v where v was only incident to edges in A, might again create singletons. It takes an additional $O(n)$ time to execute Rule 8 once more to G after applying Rule 9.

Now assume neither Rule 8 nor Rule 9 applies.

To determine whether Rule 10 is applicable on instance I, we need to inspect every edge $uv \in E$ to determine whether or not set $N_{\{u,v\}}$ induces a clique.

For every edge uv, the size of $N_{\{u,v\}}$ and the value of $c_{\{u,v\}}$ are already computed during the initialization phase (and updated when necessary, as discussed above). Let $n' = |N_{\{u,v\}}|$. If $c_{\{u,v\}} = \frac{n'(n'-1)}{2}$, then $N_{\{u,v\}}$ is a clique. Therefore it takes time $O(1)$ to determine whether an edge satisfies the conditions of applying Rule 10.

Note that the application of Rule 10 does not change G (however, it does change A).

We inspect every edge in the graph once for applicability of Rule 10. Since there are at most n^2 edges in the graph, in time $O(n^2)$ we can apply Rule 10 until the rule is no more applicable.

When Rule 10 is not applicable, we in an additional time of $O(n^2)$ we can re-apply

Rule 9 and then Rule 8, since the application of Rule 10 increases the number of edges in A.

Now assume that Rule 11 is the only applicable rule on I. Rule 11 considers whether or not $N[u] = N[v]$, for each $uv \in E$. This can be done in time $O(n)$ by using a hash table to inspect edge uv. The application of Rule 11 does not change G. Therefore each edge needs to be inspected only once. There are at most n^2 edges in the graph. Therefore in time $O(n^3)$ we can determine all edges for which Rule 11 is applicable.

When Rule 11 is not applicable, in an additional time of $O(n^2)$ we can first apply Rule 9 and then Rule 8. We do this since Rule 11 increases the number of edges in A and therefore might invoke the applicability of Rule 9.

Based on above analysis, the $O(n^4)$-time initialization phase is the most expensive operation. After the initialization, in $O(n^3)$ we apply Rules 8–11, which leads to a reduced instance. Thus, the overall running time for the preprocessing of given instance is $O(n^4)$.

□

When none of the Rules 8–11 is further applicable, G is reduced to an instance with at most 2^k vertices. To prove this kernelization result, we first consider the following lemma.

Lemma 9. *Assume u and v are two vertices in a given instance of k-*ANNOTATED EDGE CLIQUE COVER*, denoted as $I =< G, k, A >$. Let CC be a solution for I. If for every $C_i \in CC$, C_i either contains both u and v, or neither u nor v, then $N[u] = N[v]$.*

Proof. We prove this lemma by contradiction. Assume we are given an instance of k-ANNOTATED EDGE CLIQUE COVER, denoted as $I =< G, k, A >$ with solution CC of size k. Assume that in CC for every $C_i \in CC$, C_i either contains both u and v, or neither u nor v. For the sake of contradiction assume that $N[u] \neq N[v]$.

Since $N[u] \neq N[v]$, there exists a vertex v' such that $uv' \in E$ but $vv' \notin E$. Let clique C_i be the clique in CC that covers edge uv'. Then, since $u \in C_i$, according to our assumption also $v \in C_i$. However, $vv' \notin E$, a contradiction to the definition of clique. □

Next we prove that our instance is kernelized when reduced.

Lemma 10. *Assume that none of the Rules 8–11 is applicable to a reduced instance $I =< G, k, A >$ of k-Annotated Edge Clique Cover. Then there are at most 2^k vertices in G, otherwise I is a no-instance.*

Proof. We prove Lemma 10 by contradiction. Assume $I =< G, k, A >$ is a reduced instance with more than 2^k vertices. Since there is no singleton in G (Rule 8) and there are no vertices that are incident only to edges in A (Rule 9), every vertex v in G has to be a member of some $C_i \in CC$, since otherwise v's incident edges are not covered.

Since Rule 11 does not apply, there is no pair of vertices u, v in G such that $N[u] = N[v]$. Now we show that if there are more than 2^k vertices, there is a pair of vertices u, v with $N[u] = N[v]$:

Recall that we assume that there are more than 2^k vertices in V. Let v_{2^k} be the $(2^k + 1)^{th}$ vertex in V. Since there are at most k cliques in CC and there are at most 2^k different ways to arrange a particular vertex into the cliques of CC, it is impossible for v_{2^k} to be part of cliques in CC while being different from all the configurations of vertices $v_0, \ldots, v_{2^k-1}$ in the arrangement.
According to Lemma 9, there is a $v_i \in V$ with $N[v_i] = N[v_{2^k}]$, which contradicts our assumption that $N[u] \neq N[v]$ (since Rule11 is not applicable).

Therefore, a clique cover for I can contain at most 2^k vertices in G if I is a reduced instance. □

4.2.2 A Bounded Tree Algorithm for k-Annotated Edge Clique Cover

Next we use a bounded search tree algorithm on the reduced instance I [14]:

Algorithm 2 Bounded Search Tree Algorithm for k-Annotated Clique Cover

Input: A reduced graph $G = (V, E)$, a set A of edges, an integer k.
Output: A clique cover of size at most k.

```
1:  $A \leftarrow \emptyset, CC \leftarrow \emptyset$.
2:  **return** $C_{cover}(G, k, A, CC)$;
3:
4:  **function** $C_{cover}(G, k, A, CC)$
5:      **if** $CC$ covers $G - A$ **then**: **return** $CC$
6:      **if** $k < 0$ **then**: **return** nil
7:      choose $e_{ij}$ such that $\binom{|N_{\{i,j\}}|}{2} - c_{\{i,j\}}$ is minimum
8:      **for** each maximal clique $C$ in $N[i] \cap N[j]$ **do**
9:          $CC' \leftarrow CC \cup \{C\}$
10:         $X' \leftarrow C_{cover}(G, k-1, A, CC')$
11:         **if** $CC' \neq nil$ **then**: **return** $CC'$
12:     **return** nil
```

Now we explain above algorithm in details and discuss its running time for a reduced instance with $|V| \leq 2^k$ vertices.

As it is shown in Lemma 7, among all the minimum solutions for I, there is a solution CC where every member is a maximal clique. The above algorithm finds such a solution CC as below:

For an edge $e \in E - A$, we branch by including each possible maximal clique that contains e and then solve the obtained instance recursively (Line 10).

In each subinstance, k is reduced by one. When $k = 0$, I is a no-instance if $E' - A' \neq \emptyset$. Otherwise, it is a yes-instance. Therefore the height of the tree is a most k.

The the height of tree is bounded by k, so the size of the tree can be controlled by reducing its width.

At each node, after choosing $e \in E$ (Line 7), we branch by including each possible maximal clique that contains e.

Note that in Line 7, for any edge $v_i v_j \in E$, if $h = \binom{|N_{\{i,j\}}|}{2} - c_{\{i,j\}}$ is minimum, $|E_m|$ is minimum where E_m is the set of missing edges that prevent $N_{\{i,j\}}$ from being a clique. Then $v_i v_j$ is contained in fewest maximal cliques: the removal of one endpoint from each $e \in E_m$ leads a maximal clique that contains both vertices v_i and v_j.

Now we calculate the size of the search tree, that is its maximum number of nodes. In the worst case, the endpoints v_i, v_j of edge $e \in E$ have $n-2$ common neighbours and no pair of their neighbours are adjacent to each other. Then every edge e is contained in $|V|$ maximal cliques. Therefore each node has at most $|V|$ branches. Since the height of the tree is at most k, there are at most $|V|^k$ nodes in the tree. Recall that $|V| \leq 2^k$.

Now we calculate the time to process any node in the tree, and the total running time to solve the reduced instance. When processing each instance (node) in the tree, we need to determine an edge with a minimum number of maximal cliques by calculating $\binom{|N_{\{i,j\}}|}{2} - c_{\{}i, j\}$ for each $e_i j \in E$. Since $N_{\{}i, j\}$ and $c_{\{}i, j\}$ are pre-calculated in initialization phase, it takes time $O(|V|^2)$ to process each node. There are $O(|V|^k)$ nodes in the tree. Therefore this bounded search algorithm takes time $O(|V|^{k+2})$. Since $|V|$ is bounded by 2^k, the overall running time for the bounded search tree algorithm is $O(2^{k^{k+2}})$.

Recall that pre-processing and kernelization are done in $O(n^4)$. Therefore $O(n^4 + 2^{k^{k+2}})$ is the total running time to solve k-Annotated Edge Clique Cover with the bounded search tree algorithm (Algorithm 2).

Chapter 5

Applications of Clique-based Problems

While reviewing the literature with respect to applications for Edge Clique Cover and Edge Clique Partition, we found articles discussing applications of these two problems in different fields. In the following sections, we discuss three representative papers, which are in the fields of Hardware Design, Computational Geometry and Bioinformatics.

5.1 Applications of Clique Cover

As defined in Chapter 3, pairs of vertices in a solution of an instance for Edge Clique Cover can appear together in more than one clique. This means that if we model an application using this computational problem, we do not constrain the number of individual objects shared by any pair of clusters when constructing a solution. This property leads to applications in the fields of compiler optimization, computational geometry and applied statistics. In this section, we present the findings of two articles: “Handling Irregular ILP Within Conventional VLIW Schedulers Using Artificial Resource Constraints” by Subramanian [21] and “Can Visibility Graphs be Represented Compactly” by Panakaj [1].

5.1.1 Handling Irregular ILP Within Conventional VLIW Schedulers Using Artificial Resource Constraints [21]

We discuss an application of Edge Clique Cover when implementing Very Long Instruction Word (VLIW) Processors. The goal of VLIW problems is to find an assignment of resources to operation types while minimizing the number of resources and supporting all operation types (where some of the operation types might be required to be run in parallel) in a given instruction set architecture (ISA). The authors of [21] discuss models for and solutions of VLIW problems by first using Edge Clique Cover, and then converting the obtained solution of Edge Clique Cover to a solution of the given VLIW instance.

The size of VLIW problems is the number of operation types in the given ISA, which is in practice typically very small.

To understand this application, we first introduce the following definitions.

Definition 1. *(ISA) An* instruction set architecture (ISA) *or* computer architecture *is a low level abstract model of a computer. The description of the model uses native data types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O.*

We can also view an ISA as a list of instructions that are all supported by a computer, and we need to consider how to program the instructions and assign resources to realize those instructions.

Famous examples of ISAs include: Hiperion DSP (digital signal processor), DEC VAX-11/780 and Intel 80x60.

Definition 2. *(*Operation Types*) are types of operations supported in a particular ISA.*

For example, a particular ISA may support the following four operations: loading, addition, subtraction and multiplication. The corresponding set of *operations types* is $O = \{LOAD, ADD, SUB, MUL\}$.

Definition 3. (Operation Set) *We define $Q \subset I \times O$ to be the set of operations, where I is the set of identifiers.*

Note that a certain operation type might appear many times to execute a given set of instruction. All operations in an ISA (even those of the same type) are considered different from each other. To distinguish different operations of the same type, in the definition above, there is an identifier in each $q = (i, t) \in Q$.

We consider the following example. Let $I = \{1, 2, 3, 4, 5\}$ and $O = \{LOAD, ADD, SUB, MUL\}$. Then $Q = \{(1, LOAD), (2, ADD), (3, SUB), (4, LOAD)\}$ is a valid operation set.

Definition 4. (Resource Set) *Finite resource set R is used to denote the set of resources that operations may potentially occupy.*

For example, $R = \{ADDER1, LOADER1, ADDER3, ISSUE2\}$, where $ADDER1, LOADER1, \ldots\}$ are resources of the computer. Below are examples of the functionality of some types of resources.

ADDER: performs addition of binary numbers.
LOADER: assigns values to parameters.
ISSUE: obtains values from parameters.
MULTIPLIER: multiplies two binary numbers.

To execute an operation type in a given ISA, we might need to use several resources in a certain order, where each usage of resource is defined as a cycle.

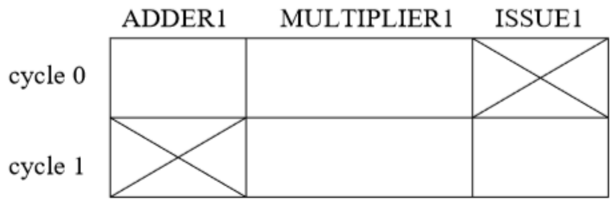
Definition 5. (Resources Usage) *A resource usage for executing an operation of type $o \in O$ is an element u of set $U = N \times R$, where N is the set of cycles, and R is the set of resources.*

For example, for $n \in N$ and $r \in R$ $u = (n, r)$ means that in the n^{th} cycle of our operation, resource $r \in R$ is occupied. Note that a resource can only be occupied at most once in each cycle.

To perform a particular operation type, several cycles might be required, where each cycle occupies exactly one resource.

In such a case we can use a *resource usage table* T, $T \in 2^U$ where 2^U is the power set of U. to show which resource is used in which cycle when executing an operation type. In other words a resource usage table shows which resources are occupied for which cycle of operations.

For example, assume in a given ISA, we require $ISSUE1$ and $ADDER1$ to perform operation ADD. Then $T = \{(0, ISSUE1), (1, ADDER1)\}$ means that ISSUE1 is occupied in Cycle 0 and ADDER1 is occupied in Cycle 1. Figure 5.1 is the corresponding resource usage table (assuming that $ISSUE1$, $ADDER1$ and $MULTIPLIER1$ are the only resources of the machine).



	ADDER1	MULTIPLIER1	ISSUE1
cycle 0			☒
cycle 1	☒		

Figure 5.1: A usage table for operation ADD.

In a given ISA, there might be several alternative usage tables for an operation type. For example, if there are resources $ADDER1$, $ADDER2$, $ISSUE1$ and $MULTIPLIER1$ in the machine, in Cycle 1 of our ADD operation, we can choose to use either $ADDER1$ or $ADDER2$. Then we have two alternative usage tables as Figure 5.1.1.

Note that two operation types $o_i, o_j \in O$ can be executed in parallel if and only if they do not need to use the same resource in the same cycle.

Note that if we assign $r_1 \in R$ to $o_1 \in O$ and $r_1, r_2 \in R$ to $o_2 \in O$, then o_1 and o_2 can run in parallel since we can let o_2 use r_2 and o_1 use r_1 whenever these two operation types run at the same time. But if we would instead assign r_1 to o_2, then r_1 and r_2 cannot be executed in parallel.

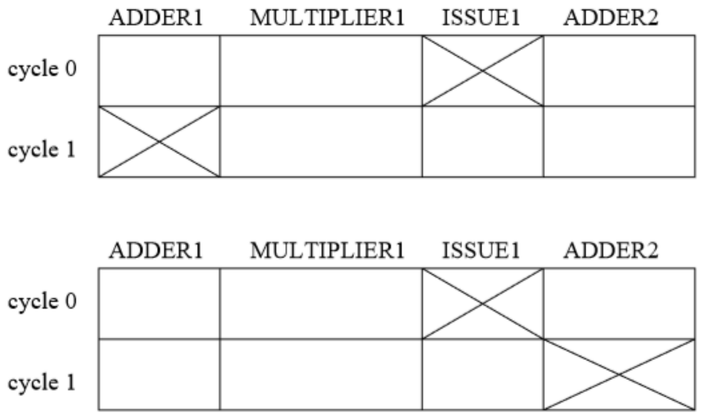


Figure 5.2: Two alternative resource usage tables for operation ADD.

In an ISA, if we do not require any pair of operation types to run in parallel, we need only one set of resources since we can run all operations one by one.

If we want to enable every pair of operation types to be executable in parallel, we need to assign a set of resources to each operation type, which is generally not practical, especially when resources are limited.

Therefore, computer engineers look for a trade-off between resource occupation and the possibility of parallel execution of operation types. When designing the ISA's, they specify which pairs of operation types should be executed in parallel and create a list accordingly. The list is notated as L in this section.

Based on this specification, our task is to assign resources to operation types to enable the parallel execution of every pair of operation types in L. The problem to be solved is called VLIW and defined below.

For the sake of reserving some resources for other operations and saving energy, we do not want to assign more resources of the machine to the ISA than necessary. Thus, the goal of VLIW is to minimize the number of resources to support all parallel operation types in L.

We can formally describe VLIW as follows. Given are a set O of operation types and a list L of pairs of operations types from O that are required to be executed in parallel. The goal is to determine a smallest possible set R of resources such that there exists an assignment A of resources from R to operation types in O while A satisfies these two constraints:

1. $A(o_i) \cap A(o_j) \neq \emptyset$ if $(o_i, o_j) \notin L$.
2. $A(o_i) \cap A(o_j) = \emptyset$ if $(o_i, o_j) \in L$.

Here $A(o)$ denotes the set of resources assigned to operation type o. For example, if resources r_1, r_3 are assigned to operation type o_i, then $A(o_i) = \{r_1, r_3\}$.

VLIW

Input:	A graph $G = (O, L)$ with operation set O as vertices of G and pairs in list L as edges of G.
Parameter:	*None*
Question:	What is a resource set R and an assignment A for R onto O such that the size of R is minimized?

We describe next how to solve an instance of VLIW via Edge Clique Cover.

ECC Algorithm for VLIW:

1. Compute the complement graph $G' = (O, E)$ for G where $E = (O \times O) - L$.
2. Compute a minimum edge clique cover CC for graph G'.
3. For each clique $C_i \in CC$, assign a unique resource r_i to every $o \in C_i$.

For example, given a set $O = \{o_1, o_2, o_3, o_4\}$ and a list $L = \{(o_1, o_4), (o_2, o_4)\}$ and we want to compute a assignment A of minimized R on O. First, we build a graph $G = (O, L)$ as stated above. Then we compute the complement graph $G' = (O, E)$ for G, where $E = \{o_1o_2, o_1o_3, o_2o_3, o_3o_4\}$. The minimum edge clique cover for G' is $CC = \{\{o_1, o_2, o_3\}, \{o_3, o_4\}\}$, the size of which is two. Therefore, the outputs are $R = \{r_1, r_2\}$ and $A(o_1) = r_1$, $A(o_2) = r_1$, $A(o_3) = r_1, r_2$ and $A(o_4) = r_2$. In our solution, operation types in pairs (o_1, o_4) share no resources, so they can be executed in parallel, so as pairs (o_2, o_4).

Before proving the soundness of above algorithm to search for the minimum assignment, we first show the definition of Edge Clique Cover problems.

Edge Clique Cover

Input:	A graph $G = (V, E)$
Parameter:	$None$
Question:	What is the minimum *edge clique cover* CC for graph G such that for every edge $uv \in E$, vertices u and v appear together in at least one clique $C \in CC$?

Since we compute the edge clique cover for $G' = (O, E)$ in our algorithm, each edge in E represents a pair of operation types which is not required to execute in parallel. In fact, we are searching for clusters where each cluster stores operation types do not execute in parallel. Therefore, each cluster only needs to be assigned one resource from R.

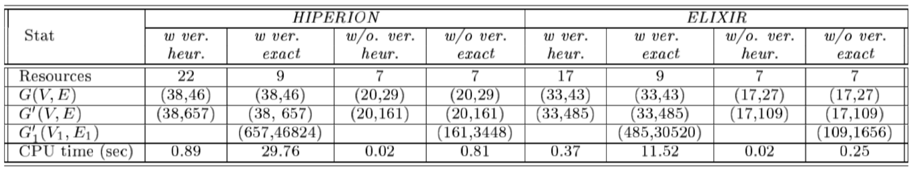
Now we prove the correctness of our algorithm by proving that the assignment generated satisfies the two constraints of VLIW we discussed above.

Proof. We first prove the first constraint. An edge $o_u o_v \in E$ exists if operation types o_u and o_v do not have to be executed in parallel. According to the definition of edge clique cover, edge $o_u o_v$ is covered by a clique $C_i \in CC$ and both of o_u and o_v are assigned with resource r_i. As a consequence, $r_i \in (A(o_u) \cap A(o_v))$, which means $A(o_u) \cap A(o_v)$ is not empty if o_u and o_v are not able to execute in parallel.

The we prove the second constraint. If operation types o_u and o_v can be executed in parallel, then $o_u o_v \notin E$ in our graph construction. Then u and v do not appear together in any $C_i \in CC$, otherwise definition of edge clique cover is violated. Therefore, there is no $r \in R$ assigned to both o_u and o_v, which means that $A(o_u) \cap A(o_v) = \emptyset$ if o_u and o_v can be executed in parallel.

Thus, if we use the algorithm described above to assign resources to operation types, both constraints are satisfies. Assume A is the assignment generated from CC. We cannot find a smaller assignment than A, which can be proved by contradiction and we do not discuss the details here. $\square$

Though EDGE CLIQUE COVER problem provide some sound strategies to solve the VLIW problem, as a well-known NP-Complete problem, it has no algorithms in polynomial running time. The paper [21] discusses a heuristic algorithms to solve EDGE CLIQUE COVER problem, but it has no guarantee about the quality of solution. As it is shown in Figure 5.3, in some data sets, the results from heuristic algorithm are of size twice larger than the smallest solutions from exact algorithm.



Stat	*HIPERION*				*ELIXIR*			
	w ver. heur.	*w ver. exact*	*w/o. ver. heur.*	*w/o ver. exact*	*w ver. heur.*	*w ver. exact*	*w/o. ver. heur.*	*w/o ver. exact*
Resources	22	9	7	7	17	9	7	7
$G(V,E)$	(38,46)	(38,46)	(20,29)	(20,29)	(33,43)	(33,43)	(17,27)	(17,27)
$G'(V,E)$	(38,657)	(38, 657)	(20,161)	(20,161)	(33,485)	(33,485)	(17,109)	(17,109)
$G'_1(V_1,E_1)$		(657,46824)		(161,3448)		(485,30520)		(109,1656)
CPU time (sec)	0.89	29.76	0.02	0.81	0.37	11.52	0.02	0.25

Figure 5.3: Implementation Results of VLIW[21].

However, sometimes engineers do not ask for the smallest assignment of resources to support parallel executions in L. Instead, they reserve a certain amount of resources and want to know whether the resources are sufficient for the ISA. In this case, we can model the problem with decision version of EDGE CLIQUE COVER problem as below, and solve the k-EDGE CLIQUE COVER with FPT algorithm (see Section 4.2).

1. Compute the complement graph $G' = (O, E)$ for G, where $E = O \times O - L$.
2. Let k be the number of available resources.
3. Search solution of k-EDGE CLIQUE COVER for instance $I =< G', k >$. k resources are sufficient to support parallel executions of operation types in L if and only if I is a yes-instance.

5.1.2 Algorithms for Compact Letter Displays: Comparison and Evaluation [13]

In fields like Biology, Social Science and Statistic, scientists design experiments and collect data to examine their hypothesis. These experiments are conducted in different environmental conditions. Each environmental condition is also called a treatment. By analyzing the results of experiments running in various treatments, scientists determine which factors have impact on the outputs of the experiment.

While some treatments are significantly different from each other, others are not. We say that two treatments are significantly different from each other if the difference of experimental results in treatments is not attributed chance. To store the significant difference between treatments and to provide convenience for further data analysis, there is a need to find a data structure that stores the information that which pairs of treatment have significant difference, within a small amount of space.

In this section, we introduce a binary matrix as the data structure to store the significant difference between treatments, as well as discuss how to apply Edge Clique Cover to generate a binary matrix for a given set H of significantly different treatment pairs. We also introduce a heuristic algorithm to find binary matrix for H and compare it with the bounded search tree algorithm for that determines binary matrix for H via Edge Clique Cover and discuss the evaluation of their running time on real-world datasets from [13].

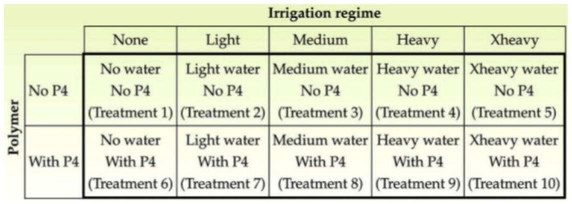
We first introduce some terminology.

Definition 6. *(Factor) In an experiment, a* factor *is an independent variable manipulated by the experimenter. [23]*

Definition 7. *(Levels)* Levels *are the possible values of a factor. In an experiment, each factor contains at least two* levels *[23].*

Definition 8. *(Treatments) A* treatment *T is an element in the Cartesian product of set of factors F and set of levels L, denoted as $T \in F \times L$ [23].*

We call experiments that contain multiple factors multifactor experiments. Since scientists want to determine how different levels of factors affect the experimental results, many experiments conducted are multifactor. Figure 5.4 is an example of a multifactor experiment.



		Irrigation regime				
		None	Light	Medium	Heavy	Xheavy
Polymer	No P4	No water No P4 (Treatment 1)	Light water No P4 (Treatment 2)	Medium water No P4 (Treatment 3)	Heavy water No P4 (Treatment 4)	Xheavy water No P4 (Treatment 5)
	With P4	No water With P4 (Treatment 6)	Light water With P4 (Treatment 7)	Medium water With P4 (Treatment 8)	Heavy water With P4 (Treatment 9)	Xheavy water With P4 (Treatment 10)

Figure 5.4: Experimental Design Table for Weight Gain of Golden Torch Cacti[23]

In Figure 5.4, there are two factors in set $F = \{$'No P4', 'With P4'$\}$ and five levels in set $L = \{$'None','Light', 'Medium','Heavy','Xheavy'$\}$. The size of Cartesian product $F \times L$ is 10. Therefore, there are 10 treatments in this experiment, one of which (Treatment 2) is 'Light water; No P4'.

In an experiment, it is important to determine whether two treatments are significantly different, since treatments have great impact on the outputs of the experiment.

Definition 9. *(Significant Difference) For treatments T_1 and T_2, if there are differences between experimental results under T_1 and under T_2, and the differences are not attributed to chance, then we define treatment T_1 and T_2 to be* significantly different.

After attaining the results under different treatments, experimenters want to analyze the effects of different treatments on their hypothesis. Therefore, they use a set H of treatment pairs to store the significant differences between treatments: $(T_i, T_j) \in H$ if and only if T_i and T_j are significantly different from each other.

Letter display is a common method to display such a set H, in which a *letter display table* Ta is used to exhibit the significant differences between treatments:

1. The rows of Ta are the treatments in the experiment.
2. Each cell of Ta is empty or contains a letter.
3. Each letter can appear in at most one column.
4. If T_i and T_j are significant different then $Ta(i, c) \neq Ta(j, c)$ for every column c.

Treatment Number	Col 1	Col2
T_1	a	
T_2	a	b
T_3	a	b
T_4		b
T_5		b

Table 5.1: Letter Display Table for $H = \{(T_1, T_4), (T_1, T_5)\}$

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$

Figure 5.5: Binary Matrix M for $H = \{(T_1, T_4), (T_1, T_5)\}$

Table 5.1 is an example of a letter display table for a *significant difference set* $H = \{(T_1, T_4), (T_1, T_5)\}$. In Table 5.1, T_1 and T_2 are not significant different. Therefore $Ta(1, 2) = Ta(2, 2) = b$. T_1 and T_4 are significantly different: Row 1 and Row 4 do not share the same letter in any column.

For further simplification, we use a binary matrix M to represent set H of significant difference between treatments:

1. The rows of matrix M are the treatment indices.
2. $M[i][c] = M[j][c] = 1$ for some column c if $(T_i, T_j) \in H$. Otherwise, for every column c, at most one of $M[i][c]$ and $M[i][c]$ equals to 1.

In Figure 5.5, M is a binary matrix representation for set $H = \{(T_1, T_4), (T_1, T_5)\}$. Compared to the letter display table, binary matrix M takes less space to store the relation of significant difference since M has same number of rows and columns as Ta and the space to store a binary bit is only 1/8 of the space to store a character.

Given n treatments where n is a constant, the size of binary matrix M is defined by its number of columns. To minimize the size of M while preserving the relation of significant difference between treatments, we introduce CLD-C problem.

Compact Letter Display (CLD-C) [13]

Input:	A positive integer n and a set H (of size m) of integer pairs where $H = (i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)$ with $1 \leq i_r, j_r \leq n$ for $r = 1, \ldots, m$
Parameter:	None
Question:	Determine a smallest M to display H?

We will show below that CLD-C and Edge Clique Cover are equivalent problems. Due to this equivalence, we can find a solution to CLD-C by solving Edge Clique Cover when given a set H of significant differences between treatments as below:

1. Build a graph $G = (V, E)$, where each vertex $v_i \in V$ is a treatment T_i that appears in H and there is an edge $e_{ij} \in E$ if and only if $(T_i, T_j) \notin H$.
2. Find a minimum edge clique cover for graph G using Algorithm 2.
3. Enumerate each clique in the obtained minimum edge clique cover.
4. Let the size of binary matrix M be defined by the size of a minimum edge clique cover for graph G.
5. We next build M as follows: if v_i is in clique j then $M[i][j] = 1$. Otherwise, $M[i][j] = 0$. Do this until every cell in M is filled.

Given M for H as instructed by above algorithm, assume that M has r rows and c columns.
We show that above algorithm is correct. For this we show that M satisfies the following two constraints:

1. If $(T_i, T_j) \notin H$, $M[i][l] = 1$ and $M[j][l] = 1$ for some $0 \leq l < c$.
2. If $(T_i, T_j) \in H$, $M[i][l] \neq M[j][l]$ or $M[i][l] = M[j][l] = 0$ for all $0 \leq l < c$.

Proof. Assume M is the binary matrix constructed from edge clique cover CC for graph G, where M has r rows and c columns.

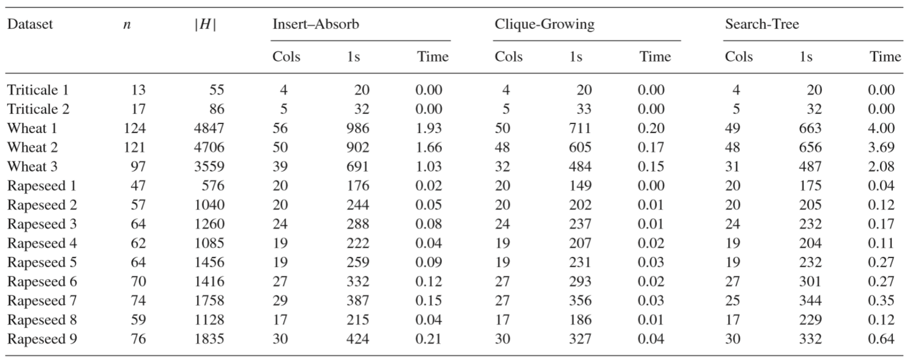
We first prove the M satisfies the first constraint. If $(T_i, T_j) \notin H$, there is is an edge $e_{ij} \in E$. When building a edge clique cover CC for G, vertex i and j appear together in some clique $C_l \in CC$, otherwise e_{ij} is not covered. Since $i, j \in C_l$, $M[i][l] = 1$ and

$M[j][l] = 1$.

We now prove that the M satisfies the second constraint. If $(T_i, T_j) \in H$, edge $e_{ij} \notin E$. Therefore, for every clique $C_l \in CC$, either neither $i, j \notin C_l$, or only one of i and j is in C_l. Then $M[i][l] = M[j][l] = 0$ or $M[i][l] \neq M[j][l]$ for every $0 \leq l < c$.

Since both constraints are satisfied, M is a correct binary matrix to represent H. $\square$

To solve the corresponding Edge Clique Cover Problem when given an CLD-C, we can apply a search tree algorithm. In [13], the authors compare the search tree algorithm to two heuristic algorithms: Insert-Absorb Heuristic and Clique-Growing Heuristic, where Clique-Growing Heuristic applies the concepts of Edge Clique Cover while Insert-Absorb Heuristic does not.



Dataset	n	$\|H\|$	Insert–Absorb			Clique-Growing			Search-Tree		
			Cols	1s	Time	Cols	1s	Time	Cols	1s	Time
Triticale 1	13	55	4	20	0.00	4	20	0.00	4	20	0.00
Triticale 2	17	86	5	32	0.00	5	33	0.00	5	32	0.00
Wheat 1	124	4847	56	986	1.93	50	711	0.20	49	663	4.00
Wheat 2	121	4706	50	902	1.66	48	605	0.17	48	656	3.69
Wheat 3	97	3559	39	691	1.03	32	484	0.15	31	487	2.08
Rapeseed 1	47	576	20	176	0.02	20	149	0.00	20	175	0.04
Rapeseed 2	57	1040	20	244	0.05	20	202	0.01	20	205	0.12
Rapeseed 3	64	1260	24	288	0.08	24	237	0.01	24	232	0.17
Rapeseed 4	62	1085	19	222	0.04	19	207	0.02	19	204	0.11
Rapeseed 5	64	1456	19	259	0.09	19	231	0.03	19	232	0.27
Rapeseed 6	70	1416	27	332	0.12	27	293	0.02	27	301	0.27
Rapeseed 7	74	1758	29	387	0.15	27	356	0.03	25	344	0.35
Rapeseed 8	59	1128	17	215	0.04	17	186	0.01	17	229	0.12
Rapeseed 9	76	1835	30	424	0.21	30	327	0.04	30	332	0.64

Figure 5.6: Comparison of Performances of Insert-Absorb Heuristic, Clique-Growing Heuristic and Search Tree Algorithm on datasets from real-world statistical analyses [13]

Figure 5.6 is the execution results of all three algorithms on different datasets. We can see that Insert-Absorb heuristic has highest running time and leads to a results of poor quality (high number of columns).

Only the search tree algorithm provides the exact solution for CLD-C, but its running time is exponential, which is only slightly smaller than Insert-Absorb heuristic.

Though Clique-Growing heuristic does not guarantee the minimum number of columns in its result, the quality of its output is very similar to the one from search tree algorithm. Moreover, its running time is much smaller than the Insert-Absorb heuristic and the search tree algorithm, which is $O(n^5)$ and can be improved to $O(n^3)$ with an additional sweeping phase [12]. Therefore, Clique-Growing heuristic might be our best choice in this application.

However, if we only need to search for a solution to CLD-C problem of size at most k, we can reduce CLD-C to k- Edge Clique Cover, then use reduction rules in Section 4.2 on the corresponding k-Edge Clique Cover instance then solve the reduced instance with Algorithm 2.

5.2 k-Edge Clique Partition on Graph Data Compression.

When reviewing the literature, we found a number of application papers for Edge Clique Cover (e.g., [1, 13]) and Vertex Clique Parition (e.g., [3, 10]), but not any paper on Edge Clique Partition.

However, in [1], the authors discuss the application of a variant of Edge Clique Cover for determining a compact representation of visibility graphs, where a given visibility graph G is represented by a set of cliques and bipartite cliques to reduce its storing space. This paper inspires us to apply Edge Clique Partition to the problem of compact representation of general graphs: when given a graph G, we want to store only its edge clique partition CP, instead of V and E.

When we use an edge clique partition CP accompanied by a set S, which store singleton in G, to compactly represent a given graph G, we store the graph as sets of vertices where repetition of vertices between sets is limited. In our future work, we will investigate whether this representation have an average size of $O(|V|)$ for general graphs $G = (V, E)$.

In this section, we prove the correctness of our approach for a compact representation of general graphs in Edge Clique Partition, and show its efficiency.

When given a graph $G = (V, E)$, we generate a compact representation CP with S for G as follows:

1. Include all singletons from G into a set S.
2. Compute an edge clique partition CP for G.
3. CP accompanied by S is the compact representation for G.

In above algorithm, we need a set S to store singletons before calculating an edge cliqe partition CP for G since we want to preserve those singletons in G, which will be discarded during the computation of CP.

To recover a compressed graph G from CP and S, we include every vertex that appears in CP and S as vertex of G and then connect every pair of vertices $u, v \in C$ for every clique $C \in CP$. The running time for the recovery is linear to the number of edges in E.

Given a graph $G = (V, E)$. We apply Edge Clique Partition to compress G into CP and S then do the recovery as stated above. Assume $G' = (V', E')$ is a graph recovered from CP. If $V = V'$ and $E = E'$, then CP is a correct compact representation for G.

Proof. We first prove $V = V'$. All the singletons in G are included in S in our algorithm and will be recovered in V'. Every vertex v with $deg(v) > 1$ is incident to an edge $e \in E$, which is covered by some $C \in CP$, so v is in C and will be recovered in V' as well. Therefore, in the recovery, no vertex is missing and $V = V'$.

Then we prove $E \subseteq E'$. For every edge $uv \in E$, u, v appear together in some $C \in CP$. In the recovery, every pair of vertices in C is connected by an edge, so $uv \in E'$. Therefore, $E \subset E'$

We now prove $E' \subseteq E$. For every pair of vertices u, v in cliques of CP, there is an edge $uv \in E$, since otherwise the definition of Edge Clique Partition is violated. In the recovery, we only connect pairs of vertices appear in every $C \in CP$ and include the corresponding edges in E'. Therefore, $E' \subseteq E$.

Finally, $E \subseteq E'$ and $E' \subseteq E$ together lead to $E = E'$ and CP is a correct compact representation for graph G.

□

Though there is no known polynomial time algorithm to generate a minimum edge clique partition for G, to store each G in graph database we only need to compute its minimum edge clique partition CP with a search tree algorithm (e.g. Algorithm 1) once and can use CP to recover G many times. Also, the time to recover G from CP is linear to the number of edges in G. Therefore, edge clique partition CP is a sound compact representation for G.

Moreover, there are many existing polynomial time heuristic algorithms to find an edge clique partition for G. Even a possibly non-minimum edge clique partition might save significant amount space for storing G.

We next prove that it takes no more space to store a graph G with edge clique partition CP than with V and E in an adjacency list, since adjacency list is a more space-efficient method to store G than the adjacency matrix.

Proof. Assume there are m edges in G. Further assume it takes 1 unit of space to store a vertex.

In an adjacency list, it takes 2 units of space to store an edge uv, since u appears in the linked-list rooted at vertex v and v appears in the linked-list rooted at vertex u. The total space to store G is $2m$ units.

Now we consider the worst case for our compact representation. If there are no K_3 subgraphs in G, then every clique in CP is of size two, since we cannot find a larger clique in G. It takes $2m$ units to store G with CP. In this case, to store G with CP takes the same amount of space as with an adjacency list.

If there is a K_ℓ subgraph in G such that $\ell > 2$, it takes $\ell^2 - \ell$ units to store the K_ℓ subgraph since there are $\ell^2 - \ell/2$ edges in K_ℓ. However, if we include K_ℓ with a $C \in CP$, it takes only ℓ units of space, which is smaller than storing all edges of K_ℓ

in E.

Therefore, if there is a clique of size at least three in CP (even CP is not minimum clique partition), it takes less space to store G with CP than with V and E.

□

Our compact representation in Edge Clique Partition not only uses less space than adjacency list, but also provides convenience for some graph-based problems such as dominating set and graph connectivity.

First we discuss the upper bound that our compact representation provides to dominating set. Assume CP is a size-k edge clique partition for a graph $G = (V, E)$. For each $C \in CP$, ever pair of vertices u, v are connected by an edge uv, and CP includes all $v \in V$. Therefore, picking a v from each $C \in CP$ leads to a dominating set for graph G. And k is an upper bound for the dominating set problem on G.

Graph Connectivity

Input:	A graph $G = (V, E)$
Parameter:	None
Question:	Whether G is a connected graph or not? That is, whether there is a path between every pair of vertices $v_i, v_j \in G$.

Then we discuss how to apply our compact representation on graph connectivity problem. Assume we determine whether G is a connected graph and given a size-k edge clique partition CP for G. We can make the decision for G using following algorithm:

1. Build a clique graph $G' = (V', E')$ for CP: each $C_i \in CP$ is an vertex $v_i \in V'$ and $v_0v_1 \in E'$ if and only if $C_0 \cap C_1 \neq \emptyset$.
2. Use Breadth-first-search (BFS) to decide whether G' is a connected graph.
3. G is a connected graph if and only if G' is a connected graph.

In this application, we do not need to recover G and the size of graph connectivity problem is reduced from $|V|$ to k.

Chapter 6

Summary and Future Work

Our project investigated clique-based problems and their applications. We first listed a number of clique-based problems (Chapter 3). Our focus for the project was on the problems k-Edge Clique Partition and k-Edge Clique Cover (Chapter 4). We did not only study and present work from the literature on existing FPT algorithms including their running time analysis, but also discussed applications of these problems found in the literature (Chapter 5). Furthermore, we described some additional reduction rules an properties for k-Edge Clique Partition that we have not seen presented in the literature (Section 4.1.3), and proposed additional applications (Chapter 5).

We hope that this project contributes ideas and motivation to the future research on clique-based problems that, we believe will continue to have a big impact in areas such as data science.

In the short-term propose we propose to continue investigating whether our properties and reduction rules for Edge Clique Partition discussed can yield a faster fixed-parameter running time or a faster practical algorithm. In particular, we are interested faster prepossessing of the problem instances to achieve a faster kernelization producing smaller problem kernels.
Generally, we believe that for both problems, k-Edge Clique Partition and k-Edge Clique Cover, there is potential to design faster and practical FPT algorithms.

During our literature review, we have encountered a number of applications for Edge

Clique Cover, k-Clique, and Vetex Clique Partition. Though there are quite a few articles describing FPT algorithms and their running time complexity of Edge Clique Partition, we are surprised about the lack of literature that discusses applications of Edge Clique Partition.

However, we believe that Edge Clique Partition or variants thereof have a great potential for application, since the problem provides some sound ways of partitioning vertices into cliques while satisfying some constraints on the repetition of vertices. We would be surprised if this this property does not help to model a number of real-world problems.

We end this report with an outlook on applications of Edge Clique Partition.

Application of Edge Clique Partition on the Administration of Clustered Communication of a Distributed Network

For some tasks we require a large amount of computation that is difficult to be executed on an individual computer. In such a situation, we need to consider the application of distributed networks, where a large number of computers work in parallel to execute a given task.

Frequently, there is a need for computers in a distributed network to communicate with each other, and those communications need to be administrated and monitored by master servers: if a communication between two computers is not administrated by a master server, there might appear consensus and inconsistency problems; if the same communication task is administrated by more than two different master servers, besides the possibility of wasting resources, the communication might receive conflicting commands from different master servers. Therefore, there it might be beneficial to consider how to arrange computers into clusters and how to assign one master server to each them.

Edge Clique Partition provides a possible solution to computers clustering problem as follows.

First, we create a graph $G = (V, E)$ where each computer in the distributed network is a vertex $v \in V$ and there is an edge $uv \in E$ if and only if communication between u, v is required to be administrated by a master server. Then we compute a minimum edge clique partition CP for G.

Next we assign a master server to a clique C to administrate all the communications between vertices of C. This ensures that every communication is administrated by exactly one master server while the number of total master servers are minimum.

For future work we suggest to use both real-world distributed network and simulated distributed network in Omnet++ to examine whether EDGE CLIQUE PARTITION provides a good clustering of computers and an efficient assignment of master servers on communications for the clusters. We can evaluate the efficiency by the occupation rate of master servers and the average waiting time of communications to be administrated in the network: a high occupation rate with low average waiting time means a good resource assignment.

Application of EDGE CLIQUE PARTITION on compact representation of graphs

As discussed in Section 5.2, we can represent a graph G compactly by an edge clique partition CP instead of an adjacency list. We have shown the correctness of our compact representation and demonstrated that the space to store CP is no more than to store the adjacency list.

We have shown the best-case and worst-case space complexity of our compact representation. However, to determine whether our compact representations provide benefits on reducing the space of real-world datasets, we think the average space complexity is also important to us. Therefore, we suggest to investigate the average space complexity of our compact representation in the future, as well as testing our compression approach on real-world datasets.

Finally, in the previous chapter we suggested possible applications that benefit from computing an edge clique partition without recovering the original graph. We have not yet run experiments that determine whether our compact representation indeed can benefit those applications in practice.

Bibliography

[1] Agarwal P. and Noga A., “Can visibility graphs be represented compactly?”, Discrete & Computational Geometry 12.3 (1994): 347-365.

[2] Akkoyunlu, A., “The Enumeration of Maximal Cliques of Large Graphs.”, SIAM J. Comput. 2, 1-6, 1973.

[3] Boginski V., Butenko S. and Pardalos P., “Mining market data: A network approach.”, Computer Operation Res. (2006) 33(11):3171–3184 Crossref.

[4] Bondy J. and Murty R., “Graph theory with applications”, Elsevier, Vol. 290. London: Macmillan, 1976.

[5] Branch and bound. (2017, February 7). In Wikipedia, The Free Encyclopedia. Retrieved 06:38, March 20, 2017, from `https://en.wikipedia.org/w/index.php?title=Branch_and_bound&oldid=764241953`

[6] Crescenzi P. and Kann V., A Compendium of NP Optimization Problems, Complexity and Approximation. Combinatorial Optimization Problems and their Approximability Properties, Springer-Verlag, Berlin, 1999.

[7] Downey R, Fellows M. and Stege U., “Parameterized complexity: A framework for systematically confronting computational intractability.”, Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future. Vol. 49. 1999.

[8] Downey R. and Fellows M., Fundamentals of Parameterized Complexity, Texts in Computer Science. Springer, 2013.

[9] Downey R., Fellows M. and Stege U., “Computational tractability: The view from mars.”, Bulletin of the EATCS 69 (1999): 73-97.

[10] Figueroa, A., Bornemann, J. and Jiang, T., “Clustering binary fingerprint vectors with missing values for DNA array data analysis.”, Journal of Computational Biology 11, 887–901 (2004)

[11] Gary R. and Johnson D., Computers and Intractability: A Guide to the Theory of NP-completeness., W. H. Freeman, 1979.

[12] Gramm J., Guo J. and Hüffner N., “Data reduction,exact,and heuristic algorithms for clique cover.”, In:Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX’06), SIAM, Philadelphia, PA, pp. 86–94.

[13] Gramm J. and Guo J.“Algorithms for compact letter displays: Comparison and evaluation.”, Computational Statistics & Data Analysis 52.2 (2007): 725-736.

[14] Gramm J. and Guo J., “Data reduction and exact algorithms for clique cover.”, Journal of Experimental Algorithmics (JEA) 13 (2009).

[15] Jörg F. and Grohe M., Parameterized Complexity Theory, (Texts in Theoretical Computer Science. an EATCS Series). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[16] Karp, M., “Reducibility among combinatorial problems.” Complexity of Computer Computations, pp. 85–103. Plenum Press, New York, NY, USA (1972).

[17] Michalewicz Z. and Fogel B.D., How to Solve It: Modern Heuristics. Springer, Berlin, Germany, second edition, 2004. ISBN 3-540-22494-7.

[18] Mujuni E. and Rosamond F., “Parameterized Complexity of the Clique Partition Problem.”, Proceedings of the Fourteenth Symposium on Computing: The Australasian Theory, Volume 77. Australian Computer Society, Inc., 2008.

[19] Niedermeier R., Invitation to Fixed Parameter Algorithms. (Oxford Lecture Series in Mathematics and Its Applications). Oxford University Press, USA, March 2006.

[20] Orlin J., “Contentment in graph theory: covering graphs with cliques.”, Indagationes Mathematicae (Proceedings). Vol. 80. No. 5. North-Holland, 1977.

[21] Rajagopalan S., Vachharajani M. and Malik S. “Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints.”, Proceedings

of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. ACM, 2000.

[22] Vazirani V., Approximation Algorithms, Springer-Verlag, 2013.

[23] Weiss N., Introductory Statistic, Addison Wesley, 2007.