

Friedrich-Schiller-Universität Jena  
Institut für Informatik  
Lehrstuhl Theoretische Informatik I /  
Komplexitätstheorie

Studienarbeit

An Iterative Compression Algorithm for  
Vertex Cover

von  
Thomas Peiselt

**Aufgabenstellung und Betreuung:**

Prof. Dr. R. Niedermeier und  
Dipl.-Inform. Michael Dom  
und Dipl.-Inform. Falk Hüffner

Jena, den 27.06.2007

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Basics and Definitions . . . . .	2
1.2	Introduction to the Vertex Cover Problem . . . . .	3
1.3	Existing Algorithms . . . . .	5
1.4	Approximating Vertex Cover . . . . .	5
1.5	Fixed-Parameter Tractability . . . . .	6
<b>2</b>	<b>Data Reduction</b>	<b>8</b>
2.1	A Few Simple Rules . . . . .	8
2.1.1	Isolated vertices . . . . .	9
2.1.2	Degree-one vertices . . . . .	9
2.1.3	Degree-two vertices . . . . .	9
2.1.4	Vertices with high degree . . . . .	10
2.1.5	Conclusion . . . . .	10
2.2	Nemhauser-Trotter kernelization . . . . .	11
<b>3</b>	<b>An Iterative Compression Algorithm for the Vertex Cover Problem</b>	<b>14</b>
3.1	Introduction to Iterative Compression . . . . .	14
3.2	A Simple Approach . . . . .	14
3.3	Reduction of the Initial Vertex Cover Size . . . . .	17
3.4	Improvement of the Compression Routine . . . . .	19
3.5	Enumerating All Minimal Vertex Covers of a Graph . . . . .	21
3.6	A Compression Routine Based on Enumerating All Minimal Vertex Covers . . . . .	25
3.7	Limits of the Iterative Compression Approach . . . . .	26
<b>4</b>	<b>Experimental Results</b>	<b>27</b>
4.1	Implementation . . . . .	27
4.2	Search Tree Size . . . . .	27
4.3	Running Time for Different Graph Densities . . . . .	33
<b>5</b>	<b>Summary and Conclusions</b>	<b>34</b>

# 1 Introduction

In 1972, Karp introduced a list of twenty-one NP-complete problems, one of which was the problem of finding a minimum vertex cover in a graph. Given a graph, one must find a smallest set of vertices such that every edge has at least one end vertex in the set. Such a set of vertices is called a minimum vertex cover of the graph and in general can be very difficult to find. In this work, we study a novel approach to finding minimum vertex covers based on a fairly new technique called iterative compression.

The work is organized as follows: in Section 1, we formally introduce the problem, present the state of the art of existing approaches to solve the problem, and give a few examples of practical applications of the VERTEX COVER problem. Section 2 deals with data reduction and problem kernels, a key concept for solving large problem instances. In Section 3, we start by developing a simple algorithm which uses iterative compression to calculate minimum vertex covers for graphs, and gradually enhance it to improve its running time. The section closes with a discussion of the difficulties encountered when applying the concept of iterative compression to the VERTEX COVER problem, and a comparison with the results of existing algorithms presented in Section 1. In Section 4, we present performance results from experiments done with the described approach on various graph types in comparison with a different implementation.

This section starts with a few basic definitions needed throughout this work, followed by an introduction of the VERTEX COVER problem alongside a strongly related problem, the INDEPENDENT SET problem. We present a summary of currently existing algorithms, both for computing exact solutions as well as for calculating approximative results.

## 1.1 Basics and Definitions

**Definition 1 (Graph)** *A graph is a pair  $(V, E)$ , where  $V$  is a set of vertices, and  $E$  is a set of edges between the vertices,  $E = \{\{u, v\} : u, v \in V\}$ .*

Throughout this work,  $n$  denotes the size of the vertex set  $V$  and  $m$  denotes the number of edges in  $E$ . All graphs are simple graphs without loops (edges connecting a vertex with itself) or multi-edges (more than one edge connecting the same pair of vertices).

**Definition 2 (Degree of a Vertex)** *The degree of a vertex  $u$ ,  $\deg_G(u)$  of a graph  $G = (V, E)$  is the number of edges incident to  $u$ :  $\deg_G(u) = |\{e \in E : e \cap \{u\} \neq \emptyset\}|$ . If the reference to the graph  $G$  is clear from the context, we just use  $\deg(u)$  to refer to the degree of a vertex  $u$  of a graph  $G$ .*

**Definition 3 (Induced Subgraph)** *Given a graph  $G = (V, E)$  and a subset of its vertices  $V_0 \subseteq V$ , the subgraph induced by  $V_0$ , denoted  $G[V_0]$ , is the graph consisting of the vertices  $V_0$  and the edges  $E_0 = \{\{u, v\} \in E : u \in V_0, v \in V_0\}$ .*

## 1.2 Introduction to the Vertex Cover Problem

The VERTEX COVER problem (VC) is one of the original 21 NP-complete problems published by Karp in 1972 [21]. It can be used to model many different problems, especially in the analysis and examination of biological data. It is extensively used in constructing phylogenetic trees, phenotype identification, and analysis of micro-array data. Consider the following simple problem:

You are given a number of appointments, and you want to keep as many of these appointments as possible. However, it might be possible that some of them overlap or are too distant to each other to get from one appointment to another one in time. Such a situation can be modelled using a graph in the following way: each appointment is associated with a vertex, and for any two conflicting appointments an edge between the associated vertices is inserted into the graph. Such a graph is generally referred to as a *conflict graph*. Such conflict graphs frequently occur in biochemistry, e.g., when a sample contains a number of DNA sequences, some of which might be erroneous and should be removed. In a more general form, we can state this problem as the data cleaning problem:

**The Data Cleaning Problem:** Assume that we are given a set of data points coming from an arbitrary source, and additional information saying that several pairs of data points contradict each other. We can model this input as a graph where the data points correspond to vertices and an edge corresponds to a conflict between two data points. The task is to clean up the data, i.e., to remove data points until all conflicts are resolved, while keeping as many data points as possible.

A solution to the data cleaning problem is a subset of all the data points, such that from all conflicting pairs of data points at least one is part of the solution. In terms of the graph model, it is a subset of the vertices such that each edge is incident to at least one vertex in the subset. Such a subset is called a VERTEX COVER, which is formally defined as follows:

**Definition 4 (VERTEX COVER)** *A vertex cover of an undirected graph  $G = (V, E)$  is a subset of its vertices  $C \subseteq V$  such that each edge  $e \in E$  has at least one endpoint in  $C$ .*

A *minimum* vertex cover of a graph  $G$  is a vertex cover of minimum size. A *minimal* vertex cover  $C$  of a graph  $G$  is a vertex cover such that no proper subset of  $C$  is a vertex cover of  $G$ . A minimum vertex cover is also a minimal vertex cover, but a minimal vertex cover is not necessarily a minimum vertex cover. The VERTEX COVER problem asks whether a vertex cover of a certain size  $k$  exists for a graph:

**Definition 5 (VERTEX COVER Problem)** *For a given undirected graph  $G = (V, E)$  and a nonnegative integer  $k$ , is there a set  $C \subseteq V$  with  $|C| \leq k$  such that  $C$  is a vertex cover of  $G$ ?*

This definition is stated as a *decision problem*, that means it asks whether there exists a vertex cover of a given size, and the answer is either "yes" or "no".

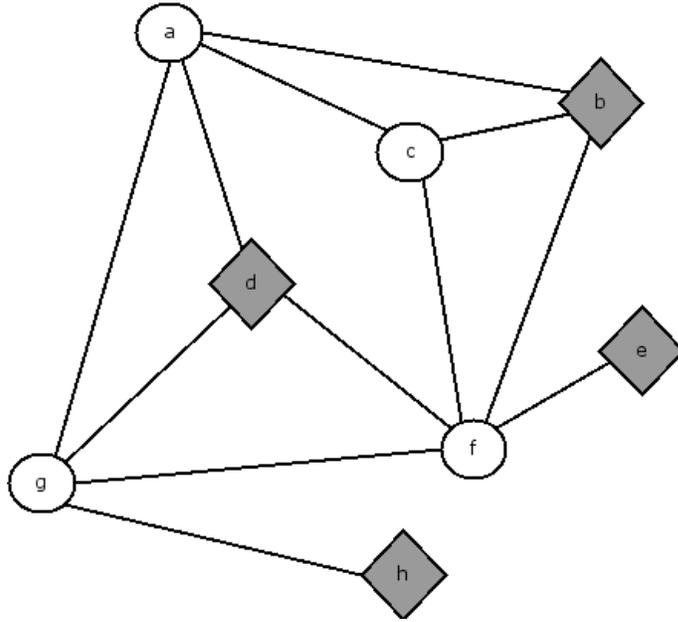


Figure 1: A graph  $G$  with the white vertices forming a vertex cover and the grey vertices forming an independent set

In practical applications however, it is usually more interesting to solve the associated *optimization problem*, which asks to find a vertex cover of minimum size.

Instead of asking for the minimum number of data points that must be removed to solve the data cleaning problem described above, one could also ask for a maximum set of data points that are conflict-free. This corresponds to a maximum set of vertices that do not have any edges in common. Such a set is called an INDEPENDENT SET, which is defined as follows:

**Definition 6 (INDEPENDENT SET)** An independent set  $I$  of an undirected graph  $G = (V, E)$  is a subset of its vertices  $I \subseteq V$  such that no edge  $e \in E$  connects two vertices of  $I$ .

The associated INDEPENDENT SET Problem is defined as follows:

**Definition 7 (INDEPENDENT SET Problem)** For an undirected graph  $G = (V, E)$  and a nonnegative integer  $k$ , is there a set  $I \subseteq V$  with  $|I| \geq k$  such that  $I$  is an independent set for  $G$ ?

The INDEPENDENT SET Problem (IS) asks whether a given graph  $G$  contains a size- $k$  subset  $I$  of its vertices  $V$  such that no pair of these vertices is connected by an edge. This implies that each edge has at least one endpoint in the set  $V \setminus I$ . Therefore  $V \setminus I$  satisfies the definition of a vertex cover, so it is obvious that whenever  $G$  contains an independent set of size  $k$ , it also contains a vertex cover of size  $|V| - k$ . By an analogous argument, the existence of a vertex cover of size  $|V| - k$ .

of size  $k$  for a graph proves the existence of an independent set of size  $|V| - k$ . An algorithm solving VC also solves IS and vice versa.

Figure 1 illustrates the connection between vertex covers and independent sets for a graph  $G$ . The vertices marked as white circles form a vertex cover, because each edge is incident to at least one of them. The remaining vertices (marked in diamond-shaped grey) form an independent set, because there is no edge connecting any two of them.

Both the INDEPENDENT SET problem and the VERTEX COVER problem are NP-complete, which means that we cannot expect to find efficient (polynomial time) algorithms to solve them, unless  $P=NP$ , which is a longstanding open question in complexity theory. For an introduction to this subject, see [16].

### 1.3 Existing Algorithms

Algorithms providing optimal solutions for VERTEX COVER and INDEPENDENT SET have been subject to widespread research, mainly due to the multitude of existing real world applications that can be modelled by these problems. (see section 1.2).

The fastest currently known algorithm for determining the size of a maximum independent set is due to Kratsch et al. and has a running time of  $O(2^{0.288n})$  [15], where  $n$  is the number of vertices of the input graph. Since the complement of an independent set of size  $k$  for a graph  $G$  forms a vertex cover of size  $n - k$  (and vice versa), we can utilize Robson's algorithm to calculate the size of a minimum vertex cover within the same time bounds. However, since the running time of Robson's algorithm is exponential in the input size, the algorithm is hardly practical for larger graphs.

### 1.4 Approximating Vertex Cover

If the available computational power is not sufficient to calculate exact results within tolerable time limits, a solution with lower quality achievable in a shorter time might be an alternative. In this section I describe an algorithm that calculates a vertex cover for a given input graph in polynomial time, where the solution is at most twice as large as any optimal solution.

The algorithm is based upon one simple observation: if you pick an arbitrary edge from a graph, at least one of the edge's endpoints must be part of any vertex cover. If you include both endpoints in the solution, you added at most one extra vertex compared to any possible cover.

It is obvious that Algorithm 1 calculates a vertex cover not more than twice the size of a minimum vertex cover:

As long as the graph contains uncovered edges, we take one of the uncovered edges  $e$  and insert both of its endpoints into the vertex cover. For each edge  $e$  we find while processing the graph, at least one of the incident vertices must be part of the cover. Without loss of generality, let  $u$  be part of an optimal cover for  $G$ . Since the approximation algorithm adds both  $u$  and  $v$  to the cover, the size of the cover is increased by two, covering all edges that would have been

---

**Algorithm 1** *ApproximateVC*( $G$ )

---

*Input:* a graph  $G$

*Output:* a vertex cover  $C$  of  $G$

```
1: while  $E \neq \emptyset$  do
2:   take an arbitrary edge  $e = \{u, v\}$ 
3:    $C \leftarrow C \cup \{u, v\}$ 
4:    $V \leftarrow V - \{u, v\}$ 
5:    $E \leftarrow \{e \in E : e \cap \{u, v\} = \emptyset\}$ 
6: end while
7: return  $C$ 
```

---

covered when only taking  $u$  (to obtain an optimal solution) and possibly some more (all edges incident to  $v$  and not to  $u$ ). Additionally, any optimal vertex cover for  $G \setminus \{u, v\}$  is not larger than any optimal vertex cover for  $G \setminus \{u\}$ . By repeatedly applying this procedure we always increase the cover size by two, knowing that at least one of these two vertices must be part of an optimal cover of the remaining graph. It is clear that our approximation leads to a vertex cover of at most twice the size of any optimal solution.

Significant improvements over the result obtained with the simple observation presented above showed to be nontrivial. In 1985, Monien and Speckemeyer presented an algorithm to approximate vertex cover within a factor of  $2 - \Theta(\frac{\log n}{\log \log n})$  [23]. 20 years later, Karakostas reduced this factor to  $2 - \Theta(\frac{1}{\sqrt{\log n}})$  [20]. Another interesting challenge arising in conjunction with the approximation of hard problems is to show lower bounds on the approximation quality that can be achieved in polynomial time. A recent result from Dinur and Safra shows that it is NP-hard to approximate vertex cover to within a factor of 1.3606 [10].

Approximative solutions twice as large as an optimal solution are inadequate for many applications. In the next section, we present a promising approach called fixed-parameter tractability, which permits the calculation of exact results for the VERTEX COVER problem with a reasonable running time if the size of the solution is not too large.

## 1.5 Fixed-Parameter Tractability

Algorithms with a running time exponentially increasing with the input size are infeasible when problem instances become too large. As seen in Section 1.4, the quality of current approximation algorithms is not good enough in many situations. To circumvent these problems, recent research [12, 25, 14] focuses on a new concept referred to as fixed-parameter tractability. The fundamental idea is to design algorithms that have a running time exponentially growing with some problem-specific parameter of the input independent of the actual input size. A so-called parameterized problem consists not only of the input instance, but has an additional component as input, which is referred to as the parameter. The formal definition reads as follows:

**Definition 8 (Parameterized Problem)** *A parameterized problem is a lan-*

guage  $L \subseteq \Sigma^* \times \Sigma^*$ , where  $\Sigma$  is a finite alphabet. The second component is called the parameter of the problem.

In the context of this work, the parameter for the VERTEX COVER problem is the size of the solution, which is a nonnegative integer. An instance of the parameterized VERTEX COVER problem then is a pair  $(G, k)$ , where  $G$  is a graph and the question is: “Does  $G$  have a vertex cover of size  $k$ ?”. Every computational problem can be rephrased into a parameterized problem by simply adding an arbitrary second component to the input. Since we are interested in parameterized problems that have a running time exponentially depending on the parameter, we define a complexity class FPT, which contains all parameterized problems that have the property of being fixed-parameter tractable:

**Definition 9 (Fixed-parameter tractable)** *A parameterized problem is fixed-parameter tractable if the question “ $(x_1, x_2) \in L$ ?” can be decided in running time  $f(|x_2|) \cdot |x_1|^{O(1)}$ , where  $f$  is an arbitrary function on nonnegative integers. The corresponding complexity class is called FPT.*

In contrast to traditional complexity theory, where the running time is dominated by the size of the input, fixed-parameter tractable problems have a running time polynomially bound by the size of the input, whenever the parameter  $k$  is fixed. This makes large and previously intractable inputs computable in a reasonable time if  $f(k)$  is sufficiently small.

The VERTEX COVER problem is fixed-parameter tractable with the vertex cover size as the parameter [13]. Chen et al. recently presented an algorithm running in  $O(1.2738^k + k \cdot n)$  time [7].

## 2 Data Reduction

An important technique when dealing with computationally hard problems is to reduce the size of the instance by applying a (polynomial-time) preprocessing step prior to applying a (computationally expensive) algorithm. Algorithms with exponential running time can heavily benefit from even a mediocre reduction of the size of the input, bringing otherwise intractable instances down to a size where current computers can solve the problems in a reasonable amount of time. The VERTEX COVER problem has been subject of extensive research, which resulted in several different preprocessing algorithms. In this section we introduce two techniques ranging from simple rules depending on the degree of specific vertices to a more sophisticated algorithm based on a theorem by Nemhauser and Trotter [24]. Both techniques presented in this section are part of our implementation used to evaluate the running time and search tree sizes in Section 4. Additionally, the preprocessing rules shown in Section 2.1 are a prerequisite for the implementation of another algorithm to solve the VERTEX COVER problem that has also been implemented and is used in comparison with our algorithm in Section 4. Before presenting these preprocessing rules, we need the following definition of a problem kernel [25]:

**Definition 10 (Problem Kernel)** *Let  $L$  be a parameterized problem. Reduction to problem kernel then means to replace the instance  $(I, k)$  by a instance  $(I', k')$  (the problem kernel) such that  $k' \leq k$ ,  $|I'| \leq g(k)$  with some function  $g$  only depending on  $k$ , and  $(I, k) \in L$  iff  $(I', k') \in L$ . Furthermore, we require that the reduction from  $(I, k)$  to  $(I', k')$  is computable in polynomial time  $T(|I|, k)$ .*

Note that every problem is fixed-parameter tractable if it has a problem kernel by the above definition: the problem can be solved by applying the kernelization algorithm and solving the preprocessed instance, using  $T(|I|, k) + f(|I'|) = T(|I|, k) + f(g(k))$  time. Since  $T$  is a polynomial function, and the value of  $f$  only depends on the parameter  $k$  and not on the size of the input instance, the running time satisfies the definition of a fixed-parameter tractable problem.

In what follows, we denote the problem instance resulting from applying a data reduction rule to an instance of the VERTEX COVER problem  $(G = (V, E), k)$  with  $(G' = (V', E'), k')$ .

### 2.1 A Few Simple Rules

In this section, we present a few preprocessing rules first proposed by Buss and Goldsmith [5], allowing to reduce the size of the problem when the degree of a vertex is small ( $\leq 2$ ) or very large ( $> k$ ). After repeatedly applying these preprocessing rules, the vertex degree of all vertices  $v$  in the resulting graph  $G' = (V', E')$  will be bounded by  $3 \leq \deg(v) \leq k$ , and it can be shown that  $n' \leq \frac{k^2}{3} + k$  [4].

We start with an instance  $(G = (V, E), k)$  of VERTEX COVER and an empty set  $C$ . Applying the rules below results in a possibly smaller instance  $(G' =$

$(V', E'), k'$ ) of VERTEX COVER, and a set  $C \subset V$  such that there exists a minimum vertex cover  $C'$  of  $G$  where  $C \subseteq C'$ .

### 2.1.1 Isolated vertices

For any vertex cover  $D$  that includes a vertex  $u$  without neighbors,  $D \setminus \{u\}$  is a smaller vertex cover for the same graph, since  $u$  has no incident edges that need to be covered. Thus, every optimal vertex cover for  $G \setminus \{u\}$  is also an optimal vertex cover for  $G$ .

*Preprocessing rule: If  $G$  contains a vertex  $u$  without neighbours, remove it from  $G$ .*

### 2.1.2 Degree-one vertices

If a graph  $G = (V, E)$  contains a vertex  $u$  with only one neighbor  $v$ , taking  $v$  into the cover  $C$  always leads to an optimal solution, because taking  $u$  only covers one edge, taking  $v$  covers the same edge plus possibly additional edges incident to  $v$ . Since either  $u$  or  $v$  must be part of the cover, we can just take  $v$ .

*Preprocessing rule: If  $G$  contains a vertex  $u$  that has only one neighbor  $v$ , remove  $u$  and  $v$  from  $G$ , set  $k' \leftarrow k' - 1$ , and add  $v$  to the vertex cover  $C$  of  $G$ .*

### 2.1.3 Degree-two vertices

Chen et al. [6] proposed a rule handling a graph  $G = (V, E)$  which includes a vertex  $u$  with exactly two neighbors,  $v$  and  $w$ . There are two possible situations:

1.  $\{v, w\} \in E$ : Two of the three vertices  $u, v, w$  must be part of any vertex cover. So at least one of the two vertices  $v, w$  must be part of an optimal solution. Without loss of generality, let  $v$  be that vertex. When we add  $v$  to the cover and remove it from the graph  $G$ , the resulting graph  $G \setminus \{v\}$  contains  $w$  as the only neighbor of  $u$ . We can now apply the preprocessing rule for degree-one vertices by adding  $w$  to the vertex cover and removing it from the graph  $G$ . So there is an optimal vertex cover  $C$  such that  $u \notin C, v \in C, w \in C$ . Removing all three vertices  $u, v$  and  $w$  from  $G$  results in a graph  $G' = (V', E')$  with  $|V'| = |V| - 3$  and  $k' = k - 2$ .

*Preprocessing rule: If  $G$  contains a vertex  $u$  with two neighbors  $v$  and  $w$  connected by an edge, remove  $u, v, w$  from  $G$ , add  $v$  and  $w$  to  $C$ , and set  $k' \leftarrow k' - 2$ .*

2.  $\{v, w\} \notin E$ : We build a new graph  $G'$  by contracting the three vertices  $u, v$  and  $w$  into a new vertex  $z$ , as shown in Figure 2. Given an optimal vertex cover  $C_{G'}$  of  $G'$ , we can calculate an optimal vertex cover  $C$  of  $G$  as follows: if  $z \in C_{G'}$ ,  $C$  contains both  $w$  and  $v$ . If  $z \notin C_{G'}$ , the vertex cover  $C$  for  $G$  contains  $u$ , because all edges incident to both  $w$  and  $v$  are either covered by  $u$  or by the neighbors of  $z$  in  $G'$ . When applying this preprocessing rule, we simplify our vertex cover instance by reducing the

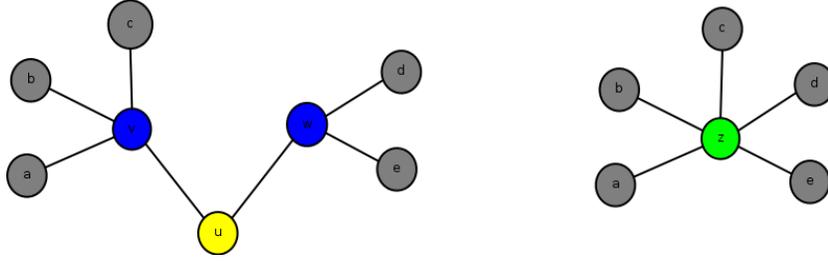


Figure 2: A vertex  $u$  with degree two and two neighbors  $v$  and  $w$  that are not connected can be seen on the left, the result of the application of the appropriate preprocessing rule can be seen on the right side.

number of vertices by two (removing  $u$ ,  $v$  and  $w$ , introducing  $z$ ) and the possible size of the cover by one.

The correctness of this rule can be seen as follows: First, consider the case that there exists a minimum vertex cover  $C$  of  $G$  with size  $k$  which does not contain  $u$ . To cover the edges incident to  $u$ , both  $v$  and  $w$  must be part of  $C$ . The reduced graph  $G'$  then has a vertex cover  $C'$  of size  $k - 1$ , because  $C \setminus \{v, w\} \cup \{z\}$  is a vertex cover for  $G'$  (all edges incident to  $v$  or  $w$  in  $G$  are covered by  $z$  in  $G'$ ). Second, consider the case where every minimum vertex cover for  $G$  contains the vertex  $u$ , and let  $C$  denote any of these minimum vertex covers with size  $k$ .  $C$  does not include  $v$  (or  $w$ ), because then a different vertex cover not including  $u$  but instead  $w$  (or  $v$ ) would exist, and we could apply the first case.  $G'$  then has a cover  $C'$  of size  $k - 1$ , because  $C \setminus \{u\}$  is a vertex cover for  $G'$ , since each edge not adjacent to  $u$  is covered by vertices in  $C$ .

*Preprocessing rule:* If  $G$  contains a vertex  $u$  with the two neighbors  $v$  and  $w$  that are not connected by an edge, remove vertices  $u$ ,  $v$  and  $w$ , introduce a new vertex  $z$  connected to all vertices previously adjacent to  $v$  or  $w$ , and set  $k' \leftarrow k' - 1$ .

#### 2.1.4 Vertices with high degree

Consider a vertex  $v$  with  $\deg(v) > k$ . Since each edge incident to  $v$  must be covered, we must take either  $v$  or all of its neighbors into the cover. But we cannot take all neighbors of  $v$  into the cover, because we are only allowed to add  $k$  vertices to the cover. This means that  $v$  must be part of any vertex cover of size  $\leq k$ .

*Preprocessing rule:* If  $G$  contains a vertex  $u$  with more than  $k$  neighbors, remove  $u$  from the graph  $G$ , add it to the cover  $C$  and set  $k' \leftarrow k' - 1$ .

#### 2.1.5 Conclusion

Applying these rules results in a graph  $G' = (V', E')$  where each vertex  $u \in V'$  has a degree of at least three and at most  $k$ . Such a reduced instance can be

calculated in  $O(n \cdot k)$  time, and constitutes a problem kernel of  $O(k^2)$  size [6]:

**Theorem 1** *If  $G = (V, E)$  is a graph with a vertex cover of size  $k$ , and for all vertices  $u \in V$  it holds that  $3 \leq \deg(u) \leq k$ , then  $|V| \leq \frac{k^2}{3} + k$ .*

**Proof:** Let  $C$  be a vertex cover of size  $k$  for  $G = (V, E)$ . Then  $I = V \setminus C$  forms an independent set of size  $n - k$  ( $n = |V|$ ). Because  $|C| = k$  and for all  $u \in C$   $\deg(u) \leq k$ , it is clear that:

$$k^2 = |C| \cdot k \geq \sum_{u \in C} \deg(u).$$

Since each edge  $e \in E$  has at least one endpoint in  $C$ , and at most one endpoint in  $I$ , it follows that:

$$\sum_{u \in C} \deg(u) \geq \sum_{v \in I} \deg(v).$$

Since  $3 \leq \deg(u)$  for all  $u \in I$  we have

$$\sum_{v \in I} \deg(v) \geq 3 \cdot |I|.$$

It follows that

$$k^2 \geq 3 \cdot |I| \text{ and } |I| \leq \frac{k^2}{3},$$

and finally

$$|V| = n = |I| + |C| \leq \frac{k^2}{3} + k. \quad \square$$

As a consequence, any reduced instance of VERTEX COVER ( $G' = (V', E'), k$ ) where  $|V'| > \frac{k^2}{3} + k$  does not have a vertex cover of size  $\leq k$ .

## 2.2 Nemhauser-Trotter kernelization

The most powerful known kernelization algorithm is based on a theorem by Nemhauser and Trotter [24] and achieves a problem kernel of size at most  $2k$ .

**Theorem 2 (Nemhauser-Trotter)** *For a graph  $G = (V, E)$  with  $n$  vertices and  $m$  edges, two disjoint sets  $C_0 \subseteq V$  and  $V_0 \subseteq V$  can be computed in time  $O(\sqrt{n} \cdot m)$ , such that the following three properties hold.*

1. *Let  $D \subseteq V_0$  be a vertex cover of the subgraph  $G[V_0]$ . Then  $C = D \cup C_0$  is a vertex cover of  $G$ .*
2. *There is a minimum vertex cover  $S$  of  $G$  with  $C_0 \subseteq S$ .*
3. *The subgraph  $G[V_0]$  has a minimum vertex cover of size at least  $|V_0|/2$ .*

For a proof to the Nemhauser-Trotter theorem, see [22] or [24]. The following method calculates the disjoint sets  $C_0$  and  $V_0$  for an input graph  $G = (V, E)$ :

1. Define a bipartite graph  $B = (V_B, E_B)$  with  $V_B = V \cup V'$ , where  $V'$  is a copy of  $V$ , and  $E_B = \{\{x, y'\} : \{x, y\} \in V\} \cup \{\{x', y\} : \{x, y\} \in V\}$ .
2. Calculate a minimum vertex cover  $C_B$  for the graph  $B$ .
3. Define  $C_0$  and  $V_0$  as follows:
  - $C_0 = \{x \in V : x \in C_B \text{ and } x' \in C_B\}$
  - $V_0 = \{x \in V : \text{either } x \in C_B \text{ or } x' \in C_B\}$

A minimum vertex cover can be computed via computing a maximum bipartite matching, according to a theorem by Koenig and Egervary. Computing a maximum matching for a bipartite graph takes  $O(\sqrt{|V|} \cdot |E|)$  time using the Hopcroft-Karp algorithm (see, for example, [8]). The graph  $G[V_0]$ , which has a vertex cover of size at most  $V_0/2$ , is the problem kernel.

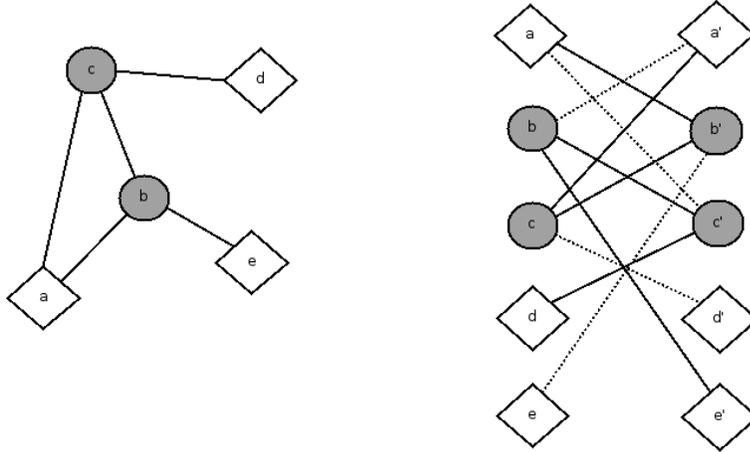


Figure 3: A graph  $G$  with a minimum vertex cover of size two and the associated bipartite graph  $B$  with a maximum matching of size four.

A graph  $G$  and its associated bipartite graph  $B$  are depicted in Figure 3. The grey circle-shaped vertices build a minimum vertex cover of size two for the original graph  $G$ , and the dotted lines form a maximum matching of size four. Any minimum vertex cover for  $B$  has the same size as the maximum matching. A possible minimum vertex cover would be  $\{b, b', c, c'\}$ . In this (perfect) case, there exists a minimum vertex cover including both  $b$  and  $c$  and not including  $a$ ,  $d$  or  $e$ , so the kernelization yields an empty problem kernel.

Note that a problem kernel based on this algorithm also allows for a factor-2-approximation of VERTEX COVER on input  $G$  by simply using  $C_0 \cup V_0$  as a vertex cover of  $G$ . The vertex set  $C_0 \cup V_0$  is a vertex cover of  $G$ , because  $V_0$  is a (trivial) vertex cover for  $G[V_0]$  and by the first property of the theorem of Nemhauser/Trotter, every set  $C_0 \cup D$  is a vertex cover for  $G$  when  $D$  is a vertex

cover for  $G[V_0]$ . Additionally, every vertex cover for  $G$  must have a size of at least  $C_0 + \frac{V_0}{2}$  (by the third property of the theorem), so  $C_0 \cup V_0$  is at most twice the size of any vertex cover of  $G$ .

### 3 An Iterative Compression Algorithm for the Vertex Cover Problem

We start by describing the basic idea behind iterative compression algorithms in general, along with a few examples on where it has been successfully utilized to develop fast algorithms for hard problems. A very simple and rather slow algorithm for vertex cover using a compression step will follow. Then we work our way up to a more advanced algorithm, which utilizes a few simple observations to reduce the size of the search space.

#### 3.1 Introduction to Iterative Compression

The basic idea of *iterative compression* is to construct smaller solutions from previously calculated larger ones. This technique was first described in a work by Reed, Smith, and Vetta [27] for the GRAPH BIPARTIZATION problem. Consider the following compression routine for VERTEX COVER:

*Input:* a graph  $G$ , a nonnegative integer  $k$ , and a vertex cover  $C$  of  $G$ .

*Output:* Vertex cover  $C'$  of  $G$ , where  $|C'| \leq k$  or "NO", if no such vertex cover exists.

Iterative compression has been successfully used to develop fast algorithms for GRAPH BIPARTIZATION [19, 27], FEEDBACK VERTEX SET [9, 18, 11], and FEEDBACK ARC SET [11].

#### 3.2 A Simple Approach

In this section we introduce an algorithm that uses a compression routine to solve VERTEX COVER.

Before starting with the algorithm description, we introduce some necessary notation, which will be used throughout the remainder of this section. The input graph for the presented algorithms will be called  $G = (V, E)$ , where  $n$  is the number of vertices in  $V$ ,  $m$  denotes the number of edges in  $E$ , and the vertices in  $V$  are denoted  $v_1, v_2, \dots, v_n$ . The subgraph of  $G$  containing only the first  $i$  vertices is denoted  $G_i$ , i.e.,  $G_i = G[\{v_1, v_2, \dots, v_i\}]$ .

Assume that you are given a compression routine *Compress*, as described above. The first idea to construct an algorithm for the VERTEX COVER problem which utilizes the compression routine could be as follows:

We start with a vertex cover  $C'$  of  $G$ , which originates from applying the approximation algorithm presented in Section 1.4. We then apply a compression step to reduce the size of the solution to  $k$ .

**Complexity:** The approximation algorithm runs in  $O(m \cdot n)$ , so the complexity of the algorithm mainly depends on the running time of the compression

---

**Algorithm 2** *VertexCover1*( $G$ )

---

*Input:* a graph  $G$  and a nonnegative integer  $k$

*Output:* a vertex cover  $C$  of  $G$  of size at most  $k$ , or “NO” if no such cover exists.

- 1:  $C' = \text{Approximate VC}(G)$
  - 2: **return**  $\text{Compress}(G, k, C')$
- 

step which has been used as a black box so far. We now present a possible implementation of the compression routine that has a running time of  $O(2^{|C'|} \cdot m\sqrt{n})$ , which means that Algorithm 2 can calculate a vertex cover of size at most  $k$  of a graph  $G$  in time  $O(2^{2k} \cdot m\sqrt{n} + m \cdot n)$ .

Similarly to other algorithms based on iterative compression [19, 27, 18, 9], our compression routine attempts to create vertex covers of the graph  $G$  based on subsets of the previously existing vertex cover  $C$ . The idea is simple: the vertices that are not part of the vertex cover  $C$  are an independent set of the graph  $G$ , so there are no edges with both endpoints in  $V \setminus C$ . If we select a subset  $C' \subset C$  in such a way that each edge which originally has both endpoints in  $C$  has at least one endpoint in  $C'$ , we can construct a graph  $G[V \setminus C']$  by deleting all vertices in  $C'$  from  $G$ . The resulting graph is bipartite, because each edge has one endpoint in  $C \setminus C'$  and one endpoint in  $V \setminus C$ . A vertex cover of a bipartite graph can be computed in polynomial time, so we can extend the partial solution  $C'$  to a vertex cover of  $G$  by calculating a vertex cover  $C_B$  for the bipartite graph  $G[V \setminus C']$  and adding  $C_B$  to  $C_C$ .

**Lemma 1** *Given a graph  $G = (V, E)$ , a nonnegative integer  $k$  and a vertex cover  $C$  of  $G$ , a vertex cover  $C'$  of  $G$  with size at most  $k$  can be computed in  $O(2^{|C|}\sqrt{n} \cdot m)$  time, if it exists.*

*Proof:*

---

**Algorithm 3** *Compress*( $G, k, C$ )

---

*Input:* a graph  $G$ , a vertex cover  $C$  of  $G$ , and a nonnegative integer  $k$ .

*Output:* a vertex cover  $C'$  of  $G$  of size at most  $k$ , or “NO” if no such cover exists.

- 1: **for all**  $C_C$  such that  $C_C \subseteq C$  **do**
  - 2:     **if**  $C_C$  is a vertex cover of  $G[C]$  **then**
  - 3:          $C_B \leftarrow \text{VertexCover}(G[V \setminus C_C])$
  - 4:         **if**  $|C_B \cup C_C| \leq k$  **then**
  - 5:             **return**  $C_B \cup C_C$
  - 6:         **end if**
  - 7:     **end if**
  - 8: **end for**
  - 9: **return** “NO”
- 

Algorithm 3 finds a vertex cover  $C'$  of  $G$  with size at most  $k$ , and returns “NO” if such a vertex cover does not exist. The algorithm checks each possible

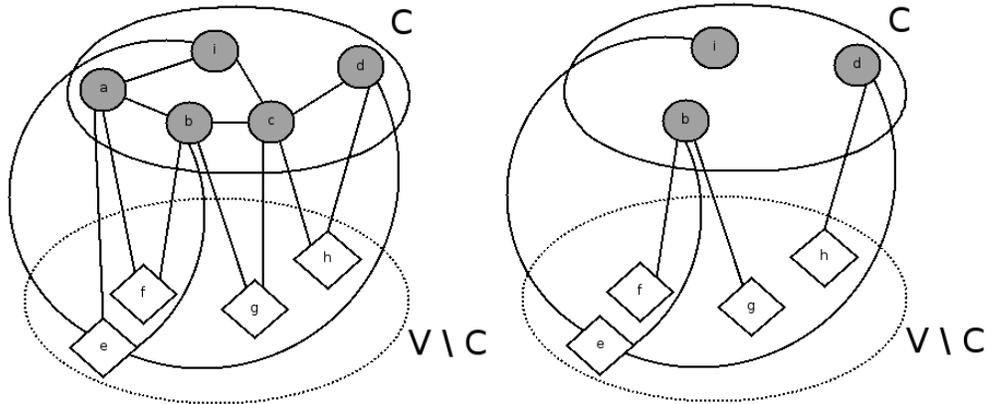


Figure 4: A graph  $G$  with a vertex cover  $C$ , alongside the bipartite subgraph  $G_B$  when  $C_C = \{a, c\}$

subset  $C_C$  of  $C$ . If  $C_C$  is a vertex cover for the subgraph  $G[C]$ , the graph  $G_B = G[V \setminus C_C]$  is bipartite, because each edge has one endpoint in  $C \setminus C_C$  and one endpoint in  $V \setminus C$ . Then it computes a minimum vertex cover  $C_B$  of  $G_B$ . The vertex set  $C_C \cup C_B$  forms a vertex cover of  $G$ , because each edge is either a part of the graph  $G_B$  and thus covered by a vertex in  $C_B$ , or it has both endpoints in the vertex set  $C$  and is thus covered by a vertex in  $C_C$ , which is a vertex cover of  $G[C]$ . Figure 4 shows an example graph  $G$  and an associated bipartite subgraph  $G_B$ , where the vertex set  $\{a, c\}$  is a vertex cover of  $G[C]$ .

To show the correctness of *Compress*, it is necessary to show the following two properties:

1. if no vertex cover of  $G$  smaller than  $k + 1$  exists, the algorithm returns "NO".
2. if at least one vertex cover of  $G$  with size at most  $k$  exists, *Compress* returns any such vertex cover.

Each vertex set constructed by Algorithm 3 is a valid vertex cover because each edge is either covered by a vertex in  $C_C$  or by a vertex in  $C_B$ . Each edge incident to a vertex in  $C_C$  is covered, because each vertex in  $C_C$  is part of the vertex cover. On the other hand, each edge not incident to a vertex in  $C_C$  must have both its endpoints in the bipartite graph  $G_B = G[V \setminus C_C]$ , and is thus covered by a vertex in the vertex cover  $C_B$  of the bipartite graph  $G_B$ . If no vertex cover of size at most  $k$  exists, the algorithm thus clearly returns "NO".

To show the second property, assume that  $G$  has a vertex cover  $C'$  which is not larger than  $k$ . Since the vertex set  $C' \cap C$  is a vertex cover of  $G[C]$ , algorithm 3 calculates a minimum vertex cover  $C_B$  of the bipartite graph  $G_B = G[V \setminus C_C]$  for  $C_C = C' \cap C$  in line 3. The vertex cover  $C_B \cup C_C$  is not larger than  $C'$ , because  $C_B$  is a minimum vertex cover of  $G_B$ , and  $C' \setminus C_C$  is also a vertex cover of  $G_B$ , and thus  $|C_B| \leq |C' \setminus C_C|$ . Our algorithm will return  $C_B \cup C_C$  which has size at most  $k$ .  $\square$

**Lemma 2** VERTEX COVER can be solved in  $O(2^{2k} \cdot m\sqrt{n} + m \cdot n)$  running time using iterative compression.

*Proof:* A graph  $G = (V, E)$  with  $|V| = n$  vertices has at most  $2^n$  different vertex covers, because each subset of  $V$  is a candidate for being a vertex cover of  $G$ . *Compress* has to check each of the  $2^{|C|}$  subsets of the old vertex cover  $C$ , and calculate the minimum vertex cover for the resulting bipartite subgraph if the subset  $C_C$  is a valid vertex cover of  $G[C]$ . A minimum vertex cover for a bipartite graph can be computed with the help of a maximum matching, and takes  $O(m\sqrt{n})$  time. *Compress* therefore takes  $O(2^{|C|}m\sqrt{n})$  maximum time. The size of the vertex covers in *VertexCover1* are  $\leq 2k$ , so the running time of our first algorithm can be bounded from above by  $O(2^{2k} \cdot m\sqrt{n} + m \cdot n)$ .  $\square$

### 3.3 Reduction of the Initial Vertex Cover Size

The running time of our first algorithm is dominated by the factor  $2^{2k}$ , which results from the fact that the *Compress* method is used to compress vertex covers of size up to  $2k$ . In this section, we present a different approach to solve the vertex cover problem, which never attempts to compress solutions that are more than  $k + 1$  vertices large.

In Algorithm 2, we started with a vertex cover for the entire graph  $G$ . Such a vertex cover, calculated by the approximation algorithm presented in Section 1.4, has a size of up to  $2k$ . Since no algorithm is known that can approximate VERTEX COVER with a constant factor smaller than 2, we need a different strategy to reduce the size of the initial vertex cover that must be compressed to improve the running time. Consider a graph  $G = (V, E)$ , with a minimum vertex cover  $C$ . If we remove an arbitrary vertex  $v \in V$ , the resulting graph  $G'$  has no vertex cover smaller than  $|C| - 1$ , because for each vertex cover  $C'$  of  $G'$ ,  $C' \cup \{v\}$  is also a vertex cover of  $G$ . On the other hand, consider that we already have a minimum vertex cover  $C'$  of  $G'$ . From  $C'$  we can immediately construct a vertex cover  $C$  of  $G$ , by adding the vertex  $v$  to  $C'$  ( $C = C' \cup \{v\}$ ). If the size of  $C'$  is larger than  $k$ , we already know that no vertex cover of size  $k$  for the graph  $G$  can exist, and if  $|C'| \leq k$  then  $|C| \leq k + 1$  which means that a call to *Compress* takes  $O(2^k \cdot m\sqrt{n})$  time.

Consider Figure 5, which depicts the situation where a new vertex  $v$  (marked as a rectangle) has been added to a graph containing eight vertices. The previous graph contained a minimum vertex cover of size four (the gray vertices), which is not a vertex cover for the new graph because some edges incident to  $v$  are not covered. So  $v$  is added to the cover, and a compression step is applied. On the right side, the compression step has been successfully applied, and a new vertex cover of size four has been found for the graph (including the gray vertices, excluding the white vertices). Using this concept, we can construct an algorithm that iteratively constructs larger graphs from smaller graphs, while keeping a minimum vertex cover for the graph built in each iteration.

Algorithm 4 starts with an empty graph  $G_0$  and a cover  $C_0 = \emptyset$ . During each step  $i$ , a new vertex  $v_i$  will be added, resulting in a new graph  $G_i$ . A vertex cover

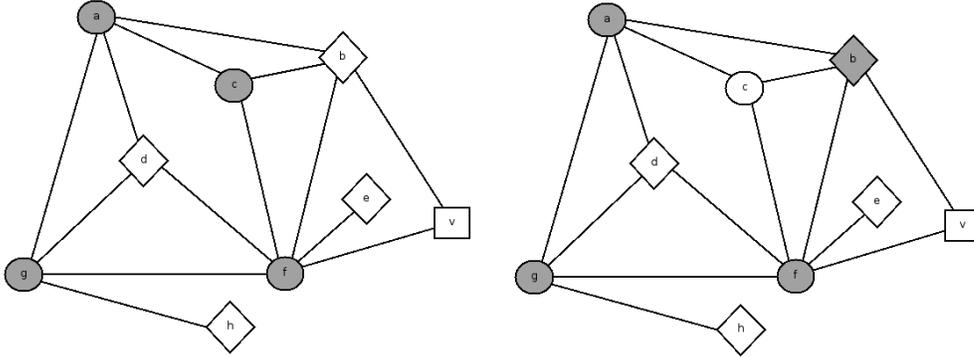


Figure 5: An example for a (successful) compression step, the graph  $G$  on the left has a cover of size 4, and the vertex labeled  $v$  was just added and has uncovered incident edges. On the right side, the gray vertices form a size-4 vertex cover of  $G$

---

**Algorithm 4** *Iterative – Compression*( $G, k$ )

---

*Input:* a graph  $G$  and a nonnegative integer  $k$ .

*Output:* a vertex cover  $C$  of  $G$  of size at most  $k$ , or “NO” if no such cover exists.

```

1:  $C_0 \leftarrow \emptyset$ 
2: for  $i = 1$  to  $n$  do
3:    $C'_i \leftarrow C_{i-1} \cup \{v_i\}$ 
4:   if COMPRESS ( $G_i, |C_{i-1}|, C'_i$ ) returns “NO” then
5:      $C_i \leftarrow C_{i-1}$ 
6:   else
7:      $C_i \leftarrow$  compressed solution
8:   end if
9:   if  $|C_i| > k$  then
10:    return “NO”
11:  end if
12: end for
13: return  $C_n$ 

```

---

$C'_i = C_{i-1} \cup \{v_i\}$  will be constructed for  $G_i$ , where  $|C'_i| = |C_{i-1} \cup \{v_i\}| \leq |C_i| + 1$ . Finally, a compression step is applied to create a minimum vertex cover  $C_i$  of  $G_i$ . When  $C_i$  becomes larger than  $k$ , the graph  $G$  does not have a vertex cover of size  $k$ , so we immediately return “NO”.

**Lemma 3** *Using iterative compression, we can calculate a minimum vertex cover in  $O(n \cdot 2^k \cdot m\sqrt{n})$  time.*

*Proof:* Algorithm 4 computes a minimum vertex cover of the input graph  $G$ . The **for** loop is executed  $n$  times, and during each iteration, *Compress* is called exactly once. Algorithm 4 never calls the *Compress* method with a vertex cover of size larger than  $k + 1$ , so the running time of a call to *Compress* will

never exceed  $O(2^k \cdot m\sqrt{n})$  time. *Iterative-Compression* thus has a running time of  $O(n \cdot 2^k \cdot m\sqrt{n})$ .  $\square$

### 3.4 Improvement of the Compression Routine

Our previous results indicate a running time of  $O(n \cdot 2^k \cdot m\sqrt{n})$ , which is dominated by the running time of the *Compress* method, because it has to check all possible subsets of the old vertex cover  $C$  of the graph  $G$ . We argued that the method has to check all possible vertex covers for the subgraph  $G[C]$  of  $G$ . However, in this section we show that it is sufficient to check all *minimal* vertex covers of the graph  $G[C]$  to compute a minimum vertex cover of  $G$ .

**Lemma 4** *Let  $C'$  be a vertex cover of the subgraph  $G[C]$  and let  $C'' \supsetneq C'$  be another vertex cover of  $G[C]$ . Additionally, let  $C'_B$  be a minimum vertex cover of the bipartite subgraph  $G'_B = (V'_B, E'_B) = G[V \setminus C']$ , and let  $C''_B$  be a minimum vertex cover of the bipartite subgraph  $G''_B = (V''_B, E''_B) = G[V \setminus C'']$ . Then both  $C' \cup C'_B$  and  $C'' \cup C''_B$  are vertex covers of  $G$ , and  $|C' \cup C'_B| \leq |C'' \cup C''_B|$*

*Proof:* We already know that both  $C' \cup C'_B$  and  $C'' \cup C''_B$  are vertex covers of  $G$  according to Lemma 1.

Note that  $G''_B$  is a subgraph of  $G'_B$ , and  $V'_B \setminus V''_B = C'' \setminus C'$ . We now prove that the vertex set  $C' \cup C'_B$  is not larger than the vertex set  $C'' \cup C''_B$ . We begin by showing that  $C''_B \cup (C'' \setminus C')$  is a vertex cover of  $G'_B$ :

- Each edge  $e \in E'_B$  that has both endpoints in  $V''_B$  also exists in  $G''_B$ , and is thus covered by a vertex in  $C''_B$ .
- All other edges have at least one endpoint in  $V'_B \setminus V''_B$ , and are thus covered by a vertex in  $C'' \setminus C'$ , because  $C'' \setminus C' = V'_B \setminus V''_B$ .

Since  $C'_B$  is a minimum vertex cover of  $G'_B$ , and  $C''_B \cup (C'' \setminus C')$  is also a vertex cover of  $G'_B$ , it follows that:

$$|C'_B| \leq |C''_B \cup (C'' \setminus C')|.$$

And finally, since  $C'' \cup C''_B = C''_B \cup (C'' \setminus C') \cup C'$ :

$$|C' \cup C'_B| \leq |C'' \cup C''_B \cup (C'' \setminus C')| = |C'' \cup C''_B|.$$

which completes the proof.  $\square$

Instead of iterating over all subsets of  $C$ , as done in Algorithm 4, it is sufficient to iterate over all *minimal* vertex covers of  $G[C]$ , because every other vertex cover is a proper superset of one or more minimal vertex covers. We now present an improved compression routine that has a running time of  $O(1.73^{|C|} \cdot m\sqrt{n})$  time.

We have already seen that a graph  $G_B = G[V \setminus C_C]$  is bipartite when  $C_C$  is a vertex cover of the subgraph  $G[C]$  and  $C$  is a vertex cover of  $G$ . On the other

hand, if  $G_B$  is not bipartite, there must exist an edge in  $G[C \setminus C_C]$ . Consider a graph  $G = (V, E)$  with a vertex cover  $C$  and an edge  $e = \{u, v\}$ , where both  $u$  and  $v$  are in  $C$ . At least one of the two vertices  $u$  and  $v$  must be part of any vertex cover of the subgraph  $G[C]$ , because  $e$  must be covered by one of its endpoints. An algorithm taking advantage of the existence of the edge  $e$  could branch according to an edge  $e = \{u, v\}$  in  $G[C]$  in the following way:

1. Put both  $u$  and  $v$  into the vertex cover  $C_C$ .
2. Put  $u$  into the vertex cover  $C_C$ , but not  $v$ .
3. Put  $v$  into the vertex cover  $C_C$ , but not  $u$ .

We can stop branching when the subgraph  $G[C \setminus C_C]$  does not contain any edges, because then  $G[V \setminus C_C]$  is a bipartite graph, and according to Lemma 4, it is not necessary to consider a vertex cover  $C_D \supset C_C$ , as extending  $C_D$  to a vertex cover of  $G$  would never result in a smaller vertex cover of  $G$ , compared to extending  $C_C$ .

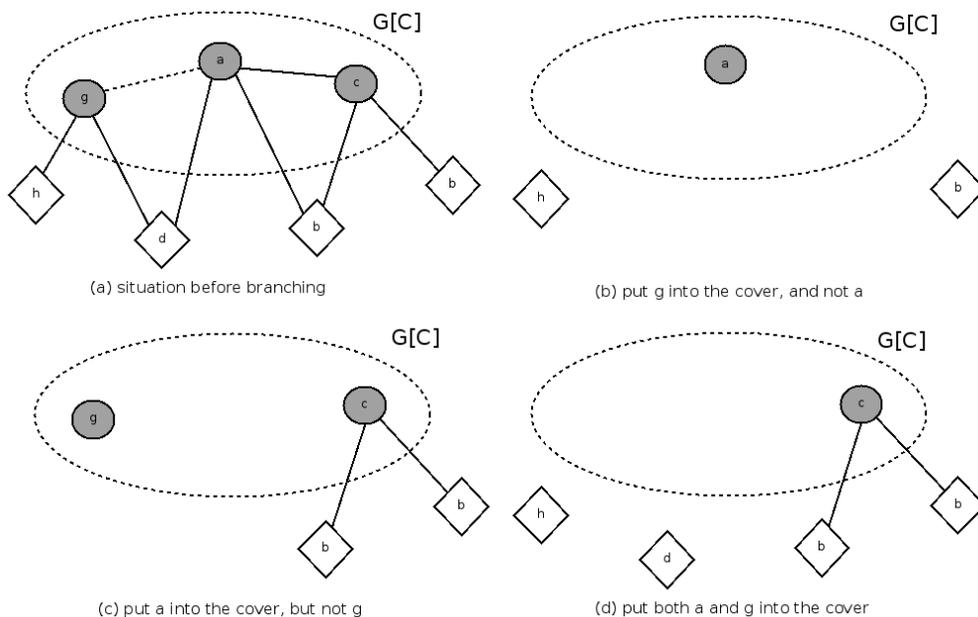


Figure 6: (a) shows the graph before branching regarding the edge  $\{a, g\}$ , (b), (c) and (d) show the resulting graph after applying the three different cases.

Figure 6 illustrates the possible situations when branching regarding the edge  $\{a, g\}$ . Note that whenever we choose to *not* put a vertex into a possible vertex cover, all its neighbors must be part of the cover, because all incident edges must be covered.

To analyze the size of the search tree for Algorithm 5 it is sufficient to determine the branching vector for the different cases that can occur during its execution [17]. For each vertex  $v \in C$ , we have to decide whether it becomes a

---

**Algorithm 5** *Compression*( $G, k, C, C_C$ )

---

*Input:* a graph  $G$ , a nonnegative integer  $k$  a vertex cover  $C$  of  $G$  and a vertex set  $C_C \subseteq C$ .

*Output:* a vertex cover  $C'$  of  $G$  of size at most  $k$ , or “NO” if no such cover exists.

- 1: **if** There is an edge  $e = \{u, v\}$  in  $G[C \setminus C_C]$  **then**
  - 2:     COMPRESSION( $G, k, C, C_C \cup \{u\} \cup \{v\}$ )
  - 3:     COMPRESSION( $G, k, C, C_C \cup \{u\} \cup \{w \in V \setminus C_C : w \text{ is a neighbor of } v\}$ )
  
  - 4:     COMPRESSION( $G, k, C, C_C \cup \{v\} \cup \{w \in V \setminus C_C : w \text{ is a neighbor of } u\}$ )
  - 5: **else**
  - 6:     calculate vertex cover for bipartite graph  $G[V \setminus C_C]$
  - 7: **end if**
- 

part of the potential vertex cover  $C_C$  or not. Every time we branch regarding an edge  $e = \{u, v\}$  where  $u, v \in C \setminus C_C$ , we reduce the number of vertices in  $C$  for which this decision must be made by at least two:

- when both  $u$  and  $v$  become part of the vertex cover  $C_C$ , we reduce the number of vertices in  $C \setminus C_C$  by two.
- when  $u$  becomes a part of the vertex cover, but  $v$  does not, we reduce the number of vertices in  $C \setminus C_C$  by one, but we additionally put every neighbor of  $v$  into the vertex cover, so the vertex  $v$  does not have any incident edges, so it will never be considered in any further branching decisions.
- When taking  $v$  and not  $u$  into the vertex cover, a reasoning symmetrical to the previous case applies.

This results in a branching vector of  $(2, 2, 2)$ , which translates into a running time of  $O(1.73^{|C|} \cdot m\sqrt{n})$  for each call to COMPRESSION.

**Lemma 5** *A minimum VERTEX COVER can be computed in  $O(n \cdot 1.73^k \cdot m\sqrt{n})$  using iterative compression.*

*Proof:* Using Algorithm 4 with the improved compression routine presented in this section, we can derive an upper bound of  $O(n \cdot 1.73^k \cdot m\sqrt{n})$  time, because we call Algorithm 5  $n$  times, and each call costs at most  $O(1.73^{|C|} \cdot m\sqrt{n})$  time, where  $|C| \leq k + 1$ .  $\square$

### 3.5 Enumerating All Minimal Vertex Covers of a Graph

In Section 3.4 we showed that it is not necessary to iterate over all subsets of a vertex cover  $C$  during a compression step. We developed a compression routine that takes advantage of this observation by branching in regard to edges of  $G[C]$  and achieves a running time of  $O(1.73^k \cdot m\sqrt{n})$ . However, our algorithm still creates vertex covers that are not minimal for  $G[C]$ , doing unnecessary processing by computing vertex covers for bipartite subgraphs that are no potential

candidates for optimal solutions. As depicted in Figure 6, Algorithm 5 creates three different vertex covers of  $G[C]$  when branching on the edge  $e = \{a, g\}$ :  $\{a, g\}$ ,  $\{c, g\}$  and  $\{a\}$ . However,  $\{a, g\} \supset \{a\}$  is not a minimal vertex cover.

In this section, we present an algorithm that enumerates all minimal vertex covers of a graph  $G$ . We will utilize it in Section 3.6 to develop an improved algorithm for VERTEX COVER based on iterative compression. Our algorithm will construct a search tree where each leaf of the tree represents a vertex cover  $C$  of  $G$  as follows: Given a graph  $G$  and an initially empty set  $C$ , we search for different structures in  $G$ . Depending on the structure we find, one or more vertices from  $G$  are put into the set  $C$  and get deleted from  $G$ . We recursively apply this logic until the graph does not contain any edges. The set  $C$  then is a vertex cover of the original input graph  $G$ .

We search for the following structures in the graph  $G$  and apply the first matching rule:

1. There is a vertex  $v$  in  $G$  with a degree of at least 3.

*Rule:* Either put  $v$  or all its neighbors into the vertex set  $C$ . Note that in the second case, we can also remove  $v$  from  $G$ , because it does not have any incident edges.

$\Rightarrow$  The resulting branching vector is  $(1, \deg(v) + 1)$  and at least  $(1, 4)$ .

2. There is an isolated vertex  $u$  in  $G$ . Such a vertex can be removed, as it can not be part of any minimal vertex cover. Applying this rule does not involve any branching.

*Rule:* Remove  $u$  from  $G$ .

3. There is a vertex  $u$  with degree 1 and a neighbor  $v$ . We branch depending on the degree of  $v$ :

- (a)  $\deg(v) = 1$ : Both vertices have no other neighbors, and we need exactly one of the two vertices to cover the edge (see Figure 7).

*Rule:* Branch by either putting  $u$  into the vertex set  $C$  or putting  $v$  into the set. Note that in each of the two resulting graphs, the remaining vertex is isolated and can be removed.

$\Rightarrow$  This results in a branching vector of  $(2, 2)$ .

- (b)  $\deg(v) = 2$ : In this case,  $v$  has one other neighbor,  $w$ .

*Rule:* Branch by either putting  $v$  into the cover, or both its neighbors into the cover. When putting  $v$  into the cover,  $u$  does not have any incident edges in  $G$ , so it can be removed from  $G$ . In the other case,  $u$  and  $w$  are part of the cover  $C$ , and  $v$  does not have any incident edges, so it can also be deleted from  $G$ . That constellation is depicted in Figure 8.

$\Rightarrow$  The resulting branching vector is  $(2, 3)$ . When  $w$  has no other neighbors in  $G$  (as is the case in Figure 8), the branching vector is  $(3, 3)$ .

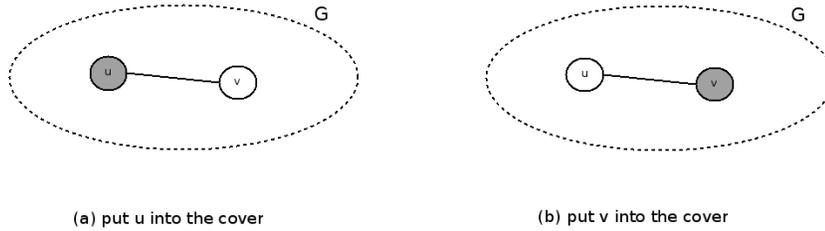


Figure 7: branching on an isolated edge in  $G$ .

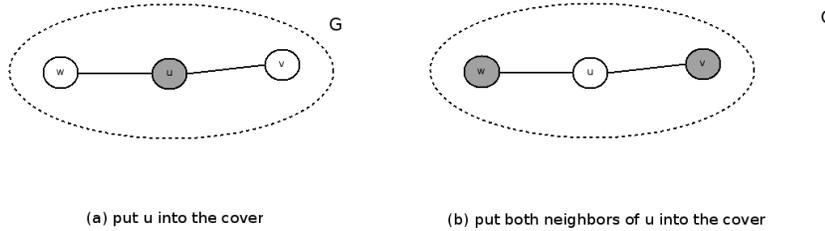


Figure 8: branching with respect to the neighbor  $u$  of a degree-one vertex  $v$ , where  $\deg(u) = 2$

4. There is no vertex with degree at most one, and no vertex with degree at least three, so all vertices of  $G$  have a degree of two.

*Rule:* Take an arbitrary vertex  $u$  with degree 2 with neighbors  $v$  and  $w$ , and branch by either putting it into the cover, or putting both its neighbors into the cover.

This would yield a branching vector of  $(1, 3)$ , because in the first case, we only remove  $u$  from  $G$ , and in the second case, both  $v$  and  $w$  are removed, whilst  $u$  becomes an isolated vertex and can also be removed from  $G$ . However, when adding  $u$  to  $C$ , the remaining structure contains one of the following constellations:

- (a) There is an edge  $e = \{v, w\}$ , and both  $v$  and  $w$  have no other neighbors (see Figure 9). In a later stage of the execution of the algorithm, we eventually apply case 3 (a) to the edge  $e$ , resulting in a branching vector of  $(3, 3, 3)$  for the successive handling of these cases. Note that no operation (except for the application of case 1 to the edge  $e$ ) ever modifies the vertices  $v$  or  $w$  or their neighbors in  $G$ , so it is not necessary to argue that the edge  $e$  is dealt with immediately after taking  $u$  into the cover – it is sufficient to know that it will be handled in each path of the search tree rooted at  $G$ .
- (b) Both  $v$  and  $w$  have degree one, but are not adjacent. Figure 10 illustrates a possible situation for this case. Because we know that when handling such a case, no edges with degree three or more can exist, it is evident that there is exactly one simple path connecting  $v$

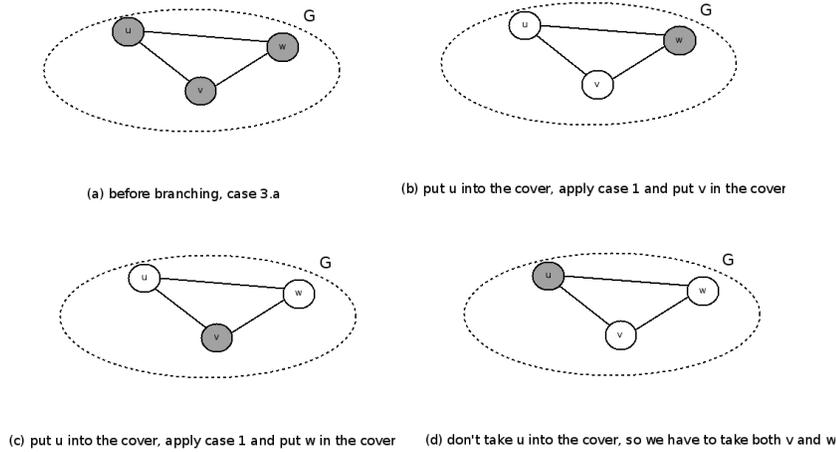


Figure 9: Branching in case of a  $K_3$ . The white vertices are part of  $C$ , the grey vertices are part of  $V$

and  $w$  in  $G$ , and all vertices on this path have a degree of exactly two. In a later stage of the algorithm, this path will be handled by a series of applications of case 3 (b), possibly followed by an application of case 3 (a). We argue that our algorithm immediately applies case 3 (b) to  $v$  and its neighbor, which means that the branching vector for the consecutive execution of these two branching operations is  $(3, 3, 4)$  in the case that  $v$  and  $w$  don't have a common neighbor. In the other case ( $v$  and  $w$  have a common neighbor  $x$ ), we also apply case 3, but the branching vector for this structure is  $(3, 3)$ , resulting in a combined branching vector of  $(3, 4, 4)$ .

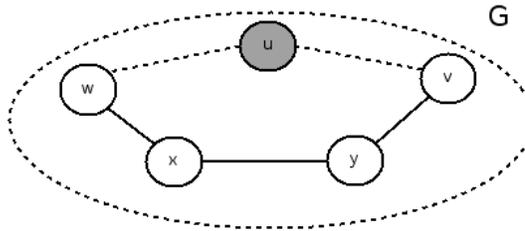


Figure 10:  $u$  with non-adjacent neighbors  $v$  and  $w$

The size of the search tree constructed using the case distinction above is bounded from above by  $O(1.443^n)$ , which is due to the branching vector  $(3, 3, 3)$  found in case 4 (a). We can immediately derive the following lemma:

**Lemma 6 (Number of minimal vertex covers)** *A graph  $G = (V, E)$  has at most  $O(1.443^n)$  minimal vertex covers, where  $n$  is the number of vertices in  $G$ .*

We showed how to enumerate all minimal vertex covers of a graph  $G$ , but it is important to note that not every leaf represents a minimal vertex cover. Figure 11 illustrates such a situation where our algorithm constructs a vertex cover which is not minimal: At first, we apply case 1, because  $u$  has a degree of at least three. In the case that we put  $u$  into the cover, the resulting graph  $G'$  has three isolated edges, which means that one of the minimal vertex covers of  $G'$  constructed by our algorithm is  $\{x, y, z\}$ . However, since the set  $C$  already contains  $u$  when called with the input graph  $G'$ , the resulting vertex cover contains  $\{u, x, y, z\}$ , which is a superset of the minimal vertex cover  $\{x, y, z\}$ .

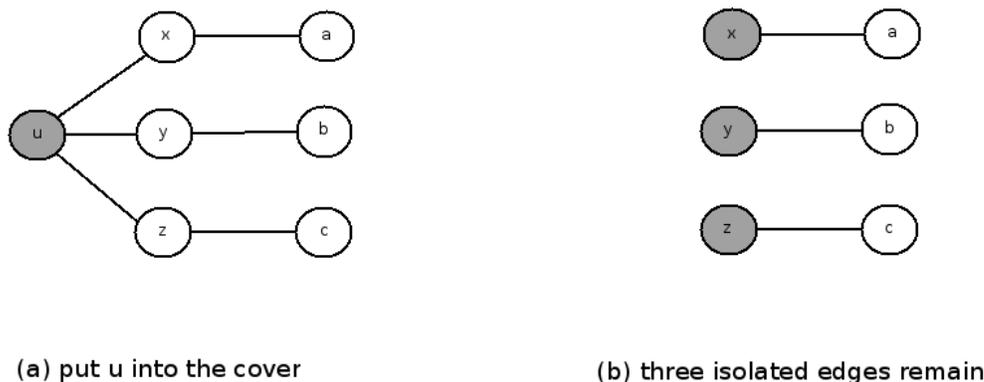


Figure 11: Construction of a vertex cover which is not minimal

### 3.6 A Compression Routine Based on Enumerating All Minimal Vertex Covers

Based on the algorithm presented in Section 3.5 it is a straightforward task to develop a compression routine for an input consisting of a graph  $G = (V, E)$ , a vertex cover  $C$  of  $G$  and a nonnegative integer  $k$ :

1. enumerate all minimal vertex covers  $C'$  of  $G[C]$
2. for each minimal vertex cover  $C'$ , compute a minimum vertex cover  $C_B$  for the bipartite graph  $G_B = G[V \setminus C']$
3. check whether  $|C' \cup C_B| \leq k$  and if it is, return  $C' \cup C_B$
4. if none of the vertex covers of  $G$  has a size of at most  $k$ , return “NO”;

As shown in Section 3.5 we can enumerate all minimal vertex covers in  $O(1.443^{|C|})$  time. For each of these vertex covers  $C'$ , we have to compute a minimum vertex cover for a bipartite graph  $G_B$ , which can be done in  $O(m\sqrt{n})$  time.

**Lemma 7** *Using iterative compression, a minimum vertex cover of size at most  $k$  can be computed in  $O(n \cdot 1.443^k \cdot m\sqrt{n})$  time.*

*Proof:* Using Algorithm 4 with the improved compression routine outlined above, we can derive an upper bound of  $O(n \cdot 1.443^k \cdot m\sqrt{n})$  time, because we need to apply the compression step  $n$  times, and each call costs at most  $O(1.443^{|C|} \cdot m\sqrt{n})$  time, where  $|C| \leq k + 1$ .  $\square$

### 3.7 Limits of the Iterative Compression Approach

The fastest parameterized algorithm to solve the VERTEX COVER problem is due to Chen et al., and computes a vertex cover of size at most  $k$  in  $O(1.2738^k + k \cdot n)$  time [7]. The algorithm based on iterative compression is drastically worse and not feasible for practical applications. The running time of our approach depends on enumerating all minimal vertex covers for a graph which has a size of at most  $k + 1$ . There exist graphs with  $n$  vertices that have up to  $3^{n/3} = 1.443^n$  minimal vertex covers, e.g. a graph containing a number of cliques of size 3, that are not connected with each other (Figure 12 gives an example).

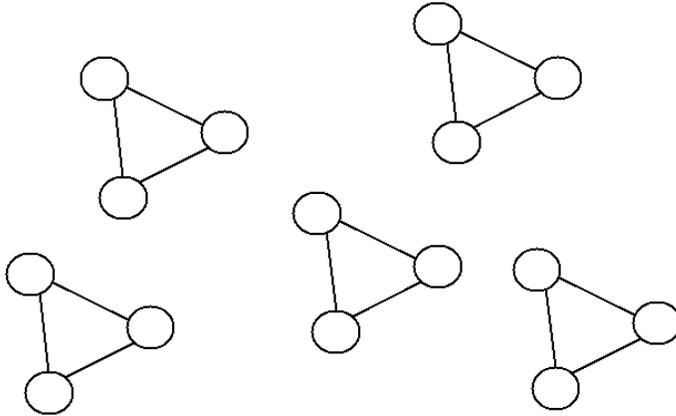


Figure 12: The worst case: A graph with  $3^{n/3}$  minimal (and minimum) vertex covers.

If we only take the structure of the subgraph  $G[C]$  into account to determine minimum vertex covers of the graph  $G$ , it is impossible to tell which of the minimal vertex covers of the graph depicted in Figure 12 potentially leads to a successful compression, forcing us to identify each of them, followed by computing a minimum vertex cover for the resulting bipartite graph. In this sense, our algorithm presented in Section 3.6 is optimal using the approach based on iterative compression under the following constraints:

1. Each minimal vertex cover of the graph  $G[C]$  is a potential candidate for a minimum vertex cover of the graph  $G$ , and
2. Our algorithm only takes structures of the subgraph  $G[C]$  into account.

The only information we have about vertices that are not part of an existing vertex cover of the graph  $G$  (that is, vertices in  $V \setminus C$ ) is that they form an independent set. I was unable to derive any rules involving substructures of

the graph  $G$  containing such vertices, which would allow to skip minimal vertex covers of the graph  $G[C]$  to improve the running time.

## 4 Experimental Results

In this section we evaluate the running time of the algorithm described in Section 3.6 for different graph types. We measure the size of the search trees generated during the execution of the algorithm, and compare it with a different algorithm based on a simple case distinction.

### 4.1 Implementation

Our implementation was done in C++, using the Boost Graph Library [1], which offers a wide range of features, from a selection of methods to create different types of random graphs to the possibility of creating function templates which allow writing algorithms that are independent of the actual graph implementation. That way, it is possible to write an algorithm and later exchange the underlying data structures, e.g., by using a graph based on an adjacency matrix instead of adjacency lists or keeping the vertices in an array, a linked list or a balanced search tree. The Boost Graph Library also offers a multitude of algorithms, ranging from prewritten graph traversal routines for depth-first and breadth-first search to a variety of shortest path algorithms.

Our implementation is about 1500 lines, which includes algorithms to compute maximum matchings (necessary for calculating vertex covers of bipartite graphs), bipartite vertex cover, and the implementation of both algorithms used in the remainder of this section.

### 4.2 Search Tree Size

In Section 3.6 we showed an upper bound of  $O(1.443^k \cdot m\sqrt{n})$  for the complexity of a single compression step (and the size of the associated search tree) by analyzing branching vectors that occur during the execution of the compression algorithm. As this result is based on a worst-case scenario, it can only be used as an upper bound for the performance of the algorithm.

Throughout the following section, we present experimental results to show actual sizes of the search trees created by the presented algorithm. Three different graph models are used for our tests: random graphs, small-world graphs and scale-free graphs [26]. Both small-world graphs and scale-free graphs model typical properties of a multitude of real-world phenomena and are thus interesting to study. The Boost Graph Library is used to create the graphs. All tests are run using the iterative compression approach as described in Section 3.6, and, additionally, using a simple algorithm, which works as follows: by applying the preprocessing rules presented in Section 2.1, we create a problem instance where each vertex has a degree of at least 3, and branch by either taking an arbitrary vertex  $u$  or all its neighbors into the vertex cover. This results in a branching vector of at least  $(1, 3)$  and a running time of  $O^*(1.47^k)$ . For each of the three graph types we create a number of instances with a constant number of vertices

and use both algorithms to compute a minimum vertex cover for each instance. During the execution of the algorithm, we count the number of recursive calls, which corresponds to the size of the search tree generated. We present the data in two different ways: First, we relate the absolute search tree size to the size of the calculated vertex cover, which should show an exponentially growing chart, because the running time is growing exponentially with the size of the vertex cover. In a second chart, we are interested in something different: given a search tree of size  $s$  and a vertex cover size  $k$ , what is the value  $c$  such that  $c^k = s$ ? Our running time analysis in Section 3.6 indicates that we can expect a value of  $c \leq 1.443$ , which is obtained from the branching vector associated with the worst case, but we will see that  $c$  largely depends on the graph density and usually is a lot smaller.

**Random graphs:** Random graphs are generated using the method *generate\_random\_graph* provided by the boost graph library. It takes two arguments: the number  $n$  of vertices and the number  $m$  of edges. The method works as follows: a graph  $G$  with  $n$  vertices and no edges is generated. Thereafter,  $m$  edges are generated, by randomly selecting two endpoints from the set of vertices, and inserting the edge if it does not exist. Note that the procedure potentially creates multiple edges between a pair of vertices, but does not repeat the process if an edge already exists, so the resulting graph usually has less than  $m$  edges.

Results are based on 1000 graphs with 100 vertices and  $m$  uniformly distributed from  $n$  to  $n * (n - 1)/2$ , which is the number of edges in a complete graph. Increasing the number of edges (density) of a graph also increases the average vertex cover size, and we expect to observe the running time of both algorithms increasing with the number of edges.

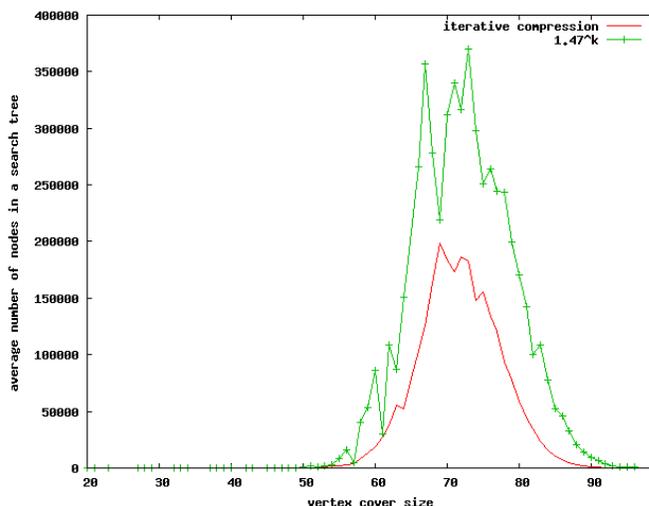


Figure 13: Average search tree sizes for random graphs with 100 vertices and varying density

Figure 13 and 14 show that the search trees produced by the iterative com-

pression approach are smaller than those constructed during the execution of the  $O(1.47^k)$  algorithm, which could be expected considering that the algorithm based on iterative compression has a somewhat smaller upper bound for the running time. We also note that both algorithms show the expected exponential growth in the search tree size up to the point where the density becomes very high. After that point, the search tree size declines again. The search tree size is determined by the branching vectors that occur during the actual execution of the algorithm. Increasing graph density results in vertices with higher degree, which means that when applying the case distinction described in Section 3.5, rule 1 will be applied more often. Since the branching vector for this rule is  $(1, \deg(v) + 1)$  for  $\deg(v) \geq 3$ , the search tree size decreases when the degrees of the vertices increase.

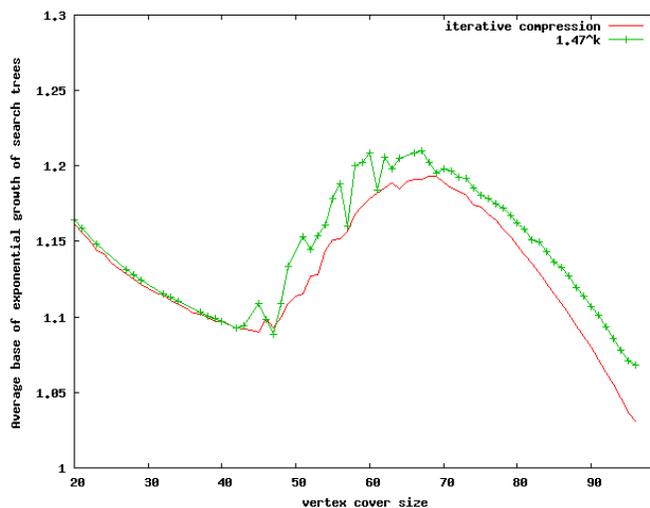


Figure 14: Average size of the base of the exponential growth for random graphs

**Small-world Graphs:** Small-world graphs frequently occur in natural phenomena, such as road maps, power grids, telephone call graphs or social influence graphs. They exhibit a high clustering coefficient and a small mean-shortest-path length. The clustering coefficient of a vertex  $v$  is the ratio of the number of edges existing between neighbors of  $v$  to the maximum number of such edges possible. A high ratio indicates that most neighbors of a vertex have a similar set of neighbors – a typical situation for social networks, often called the “all-my-friends-know-each-other” property. A small mean shortest-path length indicates that the average distance between any two vertices of a graph is small, e.g. when compared to the average shortest-path length of a random graph of same size and density. The Boost Graph Library offers a function template named *small\_world\_iterator* [2], which was used to create random small-world graphs in this work. It takes three arguments: the number of vertices  $n$ , a number of neighbors  $l$ , and a probability  $p$ , and works as follows: The initial graph created by this procedure consists of a ring graph where each vertex is connected to its

$l$  nearest neighbors (see Figure 15 for illustration). Then each edge is rewired (i.e., one of its endpoints changes) with probability  $p$ .

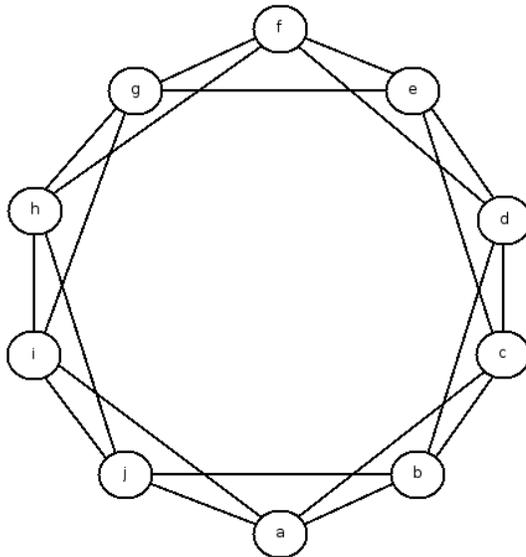


Figure 15: A ring graph with 10 vertices for  $l = 4$

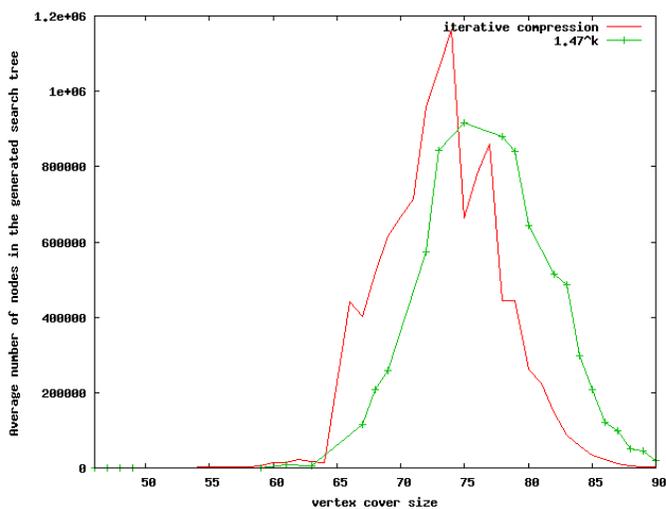


Figure 16: Average search tree size for small-world graphs with 100 vertices and varying density

Compared to random graphs, small-world graphs exhibit a slightly different behavior: First, the search tree sizes are larger than those of the random graphs, which means that it seems to be harder to calculate vertex covers of small-world graphs, at least for the algorithms used in this section. Second, the algorithm based on iterative compression creates larger search trees for graphs with lower

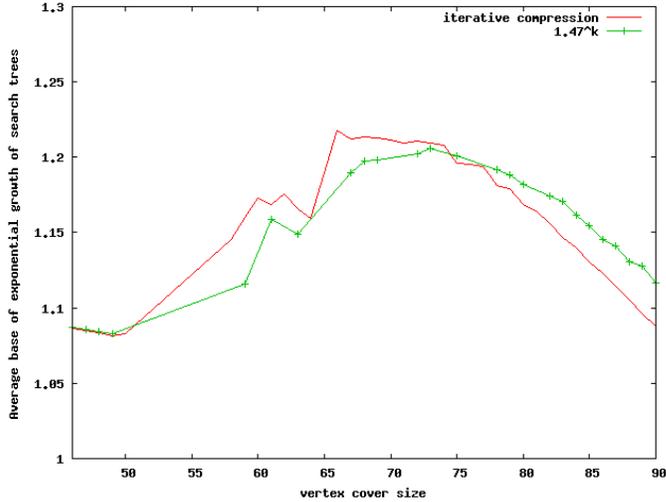


Figure 17: Average size of the base of the exponential growth for small-world graphs with 100 vertices

density, compared to the other algorithm. For graphs with high density however, the situation changes, and the search tree sizes of the algorithm presented in Section 3.6 are below those of the algorithm we compare with.

**Scale-free Graphs:** Scale-free graphs typically have a small number of vertices with very high degree (so-called hubs) and a large number of vertices with a low degree. They frequently occur in a wide range of real-world networks e.g., social networks, computer networks or neural networks. The Boost Graph Library provides a function template named *plod\_generator* [3], which takes three arguments: the number of vertices  $n$ , and two float values  $\alpha$  and  $\beta$ . The Power Law Out Degree algorithm (as described in [26]) is used to create the graph. For each vertex, it assigns a degree credit, drawn from a power-law distribution. Each vertex is assigned a credit of  $\beta \cdot x^\alpha$ , where  $x$  is a random number drawn from the range  $[0, n - 1]$ . It then creates edges between vertices, deducting a credit from both endpoints of an edge. A higher value of  $\beta$  increases the average degree of each vertex (as it increases the number of credits assigned to each vertex), the value  $\alpha$  controls how steeply the curve falls off. The web graph has  $\alpha = 2.72$ , which has been used to create the graphs in this section, and all graphs had 150 vertices.

Figure 18 shows the search tree sizes for scale-free graphs. Surprisingly, the number of recursive calls for both algorithms is much lower than those of the other two graph types, which allowed us to measure the running time for larger graphs. Another interesting difference can be seen in Figure 19: The size of the exponential base of the search tree size decreases with increasing vertex cover size and density, whereas for both small-world graphs and random graphs (as seen in Figures 17 and 14, respectively) this value increases with the size of the vertex cover up to a point when the vertex cover is about 2/3rd the size

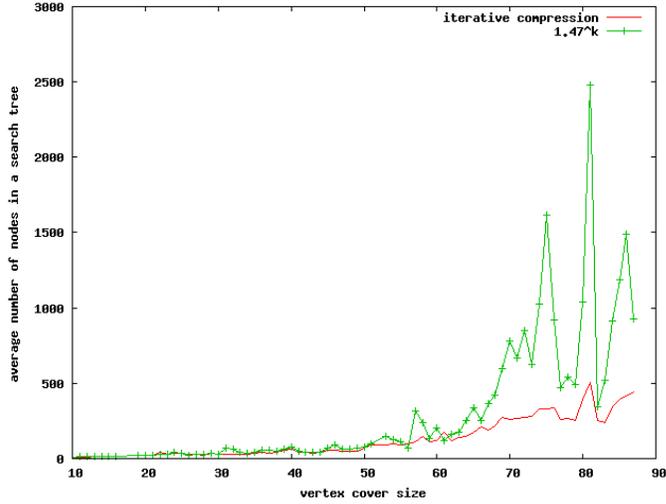


Figure 18: Search tree sizes for scale-free graphs with 150 vertices and varying density

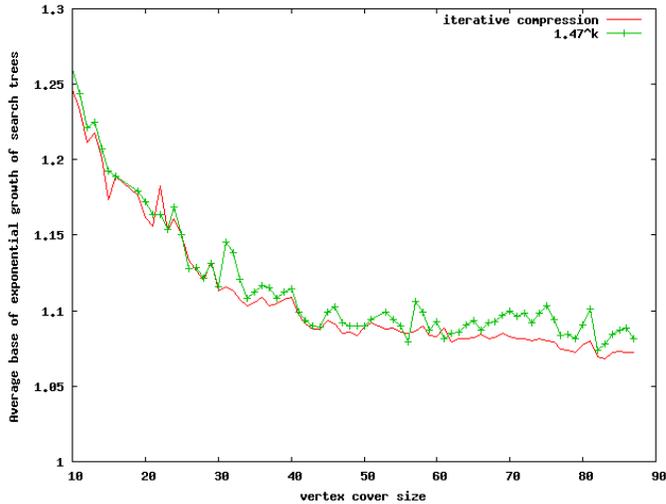


Figure 19: Average size of the base of the exponential growth for scale-free graphs with 150 vertices

of the graph, and declines when further increasing density. The surprisingly small search tree size can be attributed to the high-degree vertices found in scale-free graphs: Both algorithms branch with respect to such a high-degree vertex  $v$  by either putting it or all its neighbors into the vertex cover, resulting in branching vectors of the form  $(1, \deg(v))$ , where  $\deg(v)$  is much larger compared to the vertex degrees that are expected to be found in both small-world graphs and random graphs. The preprocessing rules described in Section 2.1 further explain the reduced search tree size: Since the vertex degrees are distributed

following a power law, we expect most vertices to have a low degree – if the degree is smaller than 3, we can apply the appropriate preprocessing rule and omit branching altogether. Since both algorithms remove vertices from the graph while branching, and both algorithms apply the preprocessing rules each time before processing a new node of the search tree, we expect to often be able to apply the preprocessing rules, even if most vertices have a degree larger than 2 in the initial graph. The second observation can also be explained by the increased branching vector size resulting from the more efficient branching due to the degree of the hub vertices increasing with the graph density.

### 4.3 Running Time for Different Graph Densities

In this section, we analyse the running time of our algorithm by calculating minimum vertex covers for graphs of varying size and density, and measure the number of seconds each calculation takes. We generated graphs with three different densities: low density ( $3n$  edges), medium density ( $n^{1.5}$  edges) and high density ( $0.2 \cdot n^2$  edges). For each of these densities and each  $n$  in  $[70, 150]$  we create 10 different graphs, using the method *generate\_random\_graph*, which is described in Section 4.2. We interrupted the computation when the running time increased over a treshold of 80 seconds per graph. We used computers with an AMD 1.4 Ghz processor from the linux pool of our faculty to obtain these numbers.

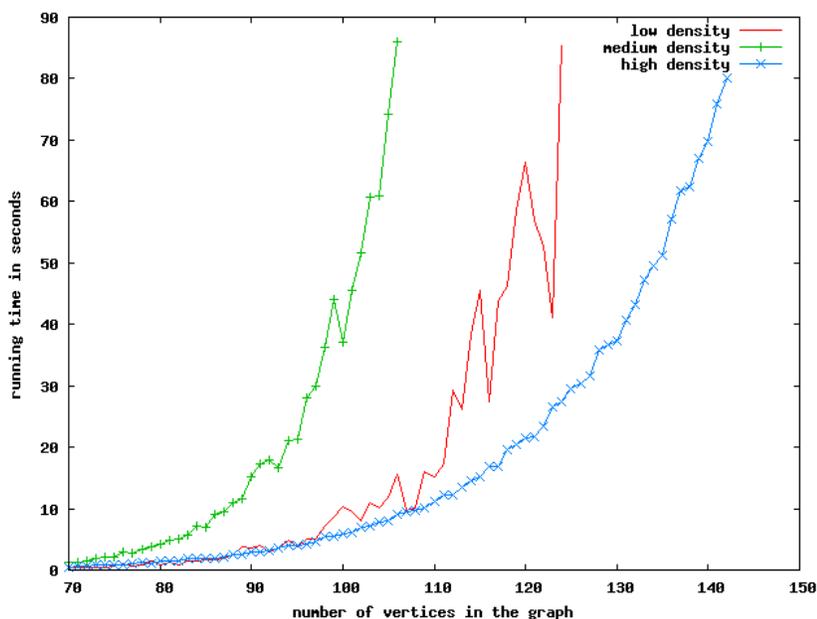


Figure 20: Minimum vertex cover calculation running time on random graphs of varying density

As we could expect from the experimental analysis of the search tree sizes, Figure 20 clearly shows that graphs of medium density take the most computing

time. However, it is surprising that they take even longer to compute than graphs with high density, considering that graphs with higher density usually have larger vertex covers. In fact, our algorithm was able to compute graphs with high density even faster than those with low density – despite an even higher discrepancy in the size of the vertex covers between these kinds of graphs, which is the dominating factor for the upper bound of the running time of our algorithm as shown in Section 3.6.

## 5 Summary and Conclusions

In Section 3 we developed an algorithm for the well-studied VERTEX COVER problem based on a fairly new technique called iterative compression. For the running time of our algorithm we established an upper bound of  $O(1.443^k \cdot n \cdot m\sqrt{n})$ , which results from the necessity to enumerate all minimal vertex covers of a graph with at most  $k + 1$  vertices. In Section 4 we evaluated our implementation with graphs of three different types, namely random graphs, small world graphs and scale-free graphs. To evaluate the running time and especially the size of the constructed search trees under different scenarios, graphs of varying density were created, and the number of recursive calls were counted. We noticed that both algorithms used in our evaluation were capable of handling scale-free graphs more efficiently than both small-world graphs and random graphs, which can be explained by the structure of scale-free graphs, that contain a few high-degree vertices. As expected from the upper bounds of the running time, the algorithm based on iterative compression creates search trees which are smaller than those created by the other algorithm, with a noticeable exception for small-world graphs with low density. In general, the figures about the search tree sizes were far below the numbers that could be expected from the theoretical analysis done in Section 3.6, but the algorithm proposed in this work is still by far worse than the best algorithms available (e.g., [7]), which can solve instances containing more than 1000 vertices. As discussed in Section 3.7, improving the compression step might only be possible if we develop rules that are not only based on the substructures found in the subgraph of a graph  $G$  induced by the existing vertex cover  $C$ , but are based on substructures of the whole graph  $G$ .

## References

- [1] Boost graph library. [www.boost.org](http://www.boost.org).
- [2] Boost graph library online documentation. [http://boost.org/libs/graph/doc/small\\_world\\_generator.html](http://boost.org/libs/graph/doc/small_world_generator.html).
- [3] Boost graph library online documentation. [http://boost.org/libs/graph/doc/plod\\_generator.html](http://boost.org/libs/graph/doc/plod_generator.html).
- [4] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Christopher T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In *ALLENEX/ANALC*, pages 62–69, 2004.
- [5] Jonathan F. Buss and Judy Goldsmith. Nondeterminism within P. *SIAM J. Comput.*, 22(3):560–572, 1993.
- [6] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover: further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.
- [7] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved parameterized upper bounds for vertex cover. In *MFCS*, pages 238–249, 2006.
- [8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [9] F. Dehne, M. Fellows, M. Langston, F. Rosamond, and K. Stevens. An  $O(2^{O(k)}n^3)$  FPT algorithm for the undirected feedback vertex set problem. *Theory of Computing Systems*, to appear.
- [10] Irit Dinur and Shmuel Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–485, 2005.
- [11] Michael Dom, Jiong Guo, Falk Hüffner, Rolf Niedermeier, and Anke Truß. Fixed-parameter tractability results for feedback set problems in tournaments. In *CIAC*, pages 320–331, 2006.
- [12] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.
- [13] Michael R. Fellows and Michael A. Langston. Nonconstructive advances in polynomial-time complexity. *Inf. Process. Lett.*, 26(3):155–162, 1987.
- [14] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, Berlin ; Heidelberg [u.a.], 2006.
- [15] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Measure and conquer: a simple  $O(2^{0.288n})$  independent set algorithm. In *SODA*, pages 18–25, 2006.

- [16] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
- [17] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for hard graph modification problems, 2004.
- [18] Jiong Guo, Jens Gramm, Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *J. Comput. Syst. Sci.*, 72(8):1386–1396, 2006.
- [19] Falk Hüffner. Algorithm engineering for optimal graph bipartization. In *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA '05)*, volume 3503 of *LNCS*, pages 240–252. Springer, 2005.
- [20] George Karakostas. A better approximation ratio for the vertex cover problem. *Electronic Colloquium on Computational Complexity (ECCC)*, (084), 2004.
- [21] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [22] Samir Khuller. Algorithms column: the vertex cover problem. *SIGACT News*, 33(2):31–33, 2002.
- [23] B. Monien and E. Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22:115–123, 1985.
- [24] G.L. Nemhauser and L.E. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.
- [25] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.
- [26] Christopher R. Palmer and J. Gregory Steffan. Generating network topologies that obey power laws. In *Proceedings of GLOBECOM '2000*, November 2000.
- [27] B. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Oper. Res. Lett.*, 32(4):299–301, 2004.