Labeled Search Trees and Amortized Analysis: Improved Upper Bounds for NP-hard Problems

Jianer Chen* Iyad A. Kanj† Ge Xia*

Abstract

A sequence of exact algorithms to solve the Vertex Cover and Maximum Independent Set problems have been proposed in the literature. All these algorithms appeal to a very conservative analysis that considers the size of the search tree, under a worst-case scenario, to derive an upper bound on the running time of the algorithm. In this paper we propose a different approach to analyze the size of the search tree. We use amortized analysis to show how simple algorithms, if analyzed properly, may perform much better than the upper bounds on their running time derived by considering only a worst-case scenario. This approach allows us to present a simple algorithm of running time $O(1.194^k k^2 + n)$ for the parameterized Vertex Cover problem on degree-3 graphs, and a simple algorithm of running time $O(1.1255^n)$ for the Maximum Independent Set problem on degree-3 graphs. Both algorithms improve the previous best algorithms for the problems.

Key words. vertex cover, independent set, exact algorithm, parameterized algorithm

1 Introduction

Recently, there has been considerable interest in developing improved exact algorithms for solving well-known NP-hard problems [8, 15]. This line of efforts was motivated by both practical and theoretical research in computational sciences. Practically, there are certain applications that require solving NP-hard problems precisely [11], while theoretically, this line of research may lead to a deeper understanding of the structure of NP-hard problems [5, 10, 12, 14].

Two of the most extensively studied problems in this line of research are the Maximum Independent Set and the Vertex Cover problems. There has been steady research in the last two decades on improved algorithms for Maximum Independent Set (given a graph G, find a maximum independent set in G) [3, 13, 19, 20, 23]. For general graphs, the best algorithm for Maximum Independent Set is due to Robson [19], whose algorithm runs in time $O(1.211^n)$. Beigel [3] developed an algorithm of running time $O(1.083^e)$ for the problem, where e is the number of edges in the graph. Applying this algorithm to degree-3 graphs, we get the currently best algorithm of running time $O(1.1259^n)$ for Maximum Independent Set on degree-3 graphs.

The Vertex Cover problem (given a graph G and a parameter k, decide if G has a vertex cover of k vertices) has drawn much attention recently in the study of parameterized complexity of NP-hard problems [10]. This is also due to its applications in fields like computational biochemistry [16]. Since the development of the first parameterized algorithm by Sam Buss, which has running time $O(kn + 2^k k^{2k+2})$ and was decribed in [4], there has been an impressive list of improved algorithms for the problem [2, 6, 7, 9, 18, 21]. Currently, the best parameterized algorithm for Vertex Cover

*Supported in part by NSF under the grants CCR-0311590 and CCF-0430683. Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. Email: {chen,gexia}@cs.tamu.edu.

†The Corresponding author. School of CTI, DePaul University, 243 S. Wabash Avenue, Chicago, IL 60604-2301 Email: ikanj@cs.depaul.edu. Supported in part by DePaul University Competitive Research Grant.

has running time $O(kn + 1.285^k)$ for general graphs [6], and the best parameterized algorithm for Vertex Cover on degree-3 graphs has running time $O(kn + 1.237^k)$ [7].

The most popular technique for solving NP-hard problems precisely is the *branch-and-search* process, which can be depicted by a search tree model described as follows. Each node of the search tree is associated with an instance of the problem. At a node α in the tree the search process considers a local structure in the problem instance associated with α, and enumerates some feasible partial solutions to the instance based on the specific local structure. Each such enumeration induces a new reduced problem instance that is associated with a child of the node α in the search tree. The search process is then applied recursively to the children of α. The complexity of a branch-and-search process, which is roughly the size of the search tree, depends mainly on two things: how effectively the feasible partial solutions are enumerated, and how efficiently the instance size is reduced. In particular, all exact algorithms proposed in the literature for the Maximum Independent Set problem and the Vertex Cover problem are based on this strategy, and most improvements were obtained by more effective enumerations of feasible partial solutions and/or more efficient reductions in the size of the problem instance [2, 6, 19, 23].

A desirable local structure may not exist at a stage of the branch-and-search process. In this case, the branch-and-search process has to pick a less favorable local structure and make a less effective branch and/or less efficient instance-size reduction. Most proposed branch-and-search algorithms for NP-hard problems were analyzed based on the worst-case performance. That is, the computational complexity of the algorithm was derived based on the worst local structure occurring in the search process. This worst-case analysis for a branch-and-search process is very conservative — the worst cases can appear very rarely in the entire process, while most other cases permit much better branching and reductions.

In the current paper, we suggest new methods to analyze the branch-and-search process. First of all, we label the nodes of a search tree to record the reduction in the parameter size for each branching process. We then perform an amortized analysis on each path in the search tree. This allows us to capture the following notion: an operation by itself may be very costly in terms of the size of the search tree that it corresponds to, however, this operation might be very beneficial in terms of introducing many efficient branches and reductions in the entire process. Therefore, the expensive operation can be well balanced by the induced efficient operations.

This analysis has also enabled us to consider new algorithm strategies in a branch-and-search process. In particular, now we do not have to always strictly avoid expensive operations. To illustrate our analysis and algorithmic techniques, we propose a very simple branch-and-search algorithm for Vertex Cover on degree-3 graphs, abbreviated VC-3. The algorithm also induces a new algorithm for Maximum Independent Set on degree-3 graphs, abbreviated IS-3. Using the new analysis and algorithmic strategies, we are able to show that the new algorithms improve the best existing algorithms in the literature. More specifically, our algorithm for VC-3 runs in time $O(n + 1.194^k k^2)$, improving the previous best algorithm of running time $O(kn + 1.237^k)$ [7], and our algorithm for IS-3 runs in time $O(1.1255^n)$, improving the previous best algorithm of running time $O(1.1259^n)$ [3].

We would like to further comment on why we picked VC-3 and IS-3 as our candidates. As we mentioned before, Vertex Cover and Maximum Independent Set are among the most extensively studied NP-hard problems with many proposed algorithms [2, 4, 6, 9, 13, 18, 19, 20, 21, 23]. In particular, Vertex Cover and Maximum Independent Set on graphs of degrees 3 and 4 have received a lot of attention recently [3, 6, 7]. In spite of the restriction imposed on graph degrees (being bounded by 3 or 4), improvements on the previous upper bounds for these problems can be challenging and meticulous. Moreover, most of the algorithms for Vertex Cover and Maximum

INDEPENDENT SET on general graphs end up reducing the problem to that on low-degree graphs [6, 18, 19]. Thus, a simple and uniform algorithm that induces significant improvements on the existing bounds for these problems is of high interest, and shows the power and effectiveness of the new analysis and algorithmic methods. In addition, recent research has shown that these problems are "complete" in terms of their worst case running time for a large group of well-known NP-hard problems [5, 12, 14]. More specifically, combining the results in [12], [14], and [5], one can show that if IS-3 can be solved in time $O((1+\epsilon)^n)$, or if VC-3 can be solved in time $(1+\epsilon)^k p(n)$ (p is a polynomial), for every constant $\epsilon > 0$, then k-SAT, MAXIMUM INDEPENDENT SET, and VERTEX COVER can all be solved in subexponential time, which seems very unlikely. Hence, it is believed that there are constants $c_1, c_2 > 0$, such that IS-3 and VC-3 have no exact algorithms of running time $O((1+c_1)^n)$ and $(1+c_2)^k p(n)$, respectively. Thus, further improvement in the base of the exponential function in the running time of the algorithms that solve these problems may lead to better understanding of the problems and their associated complexity class.

2 The main algorithm

Let $G = (V, E)$ be an undirected graph. Denote by $|G|$ the number of vertices in G. A subgraph H of G is *induced* by a subset V_H of vertices in G if H consists of the vertex set V_H and all edges in G that have both ends in V_H. A subgraph H of G is an *induced subgraph* if it is induced by a subset of vertices in G. For a subgraph H of G, denote by $G - H$ the subgraph of G obtained by removing all vertices in H. For a vertex u in G, denote by $N(u)$ the set of neighbors of u and by $d(u)$ the degree of u. A set C of vertices in G is a *vertex cover* for G if every edge in G has at least one endpoint in C. Denote by $\tau(G)$ the size of a minimum vertex cover of the graph G. An instance of the VERTEX COVER problem consists of a pair (G, k) asking whether $\tau(G) \leq k$. The VC-3 problem is the VERTEX COVER problem on graphs whose vertex degree is bounded by 3. We will assume, without loss of generality, that the graph G in an instance (G, k) of VC-3 contains no isolated vertices (such vertices can be removed in $O(|G|)$ preprocessing time). The number of edges $|E|$ in G then satisfies $|E| \geq |G|/2$ (note that G may not be connected). The degree of G is bounded by 3, and hence, every vertex in G can cover at most three edges. This means that, in order for a vertex cover of size k to exist in G, k must be at least as large as $|E|/3$ (and hence, $k \geq |G|/6$); otherwise, we can report that the answer to the instance (G, k) is negative.

The following proposition from [7] is based on a theorem by Nemhauser and Trotter [17].

Proposition 2.1 ([7]) *There is an algorithm of running time $O(k\sqrt{k})$ that, given an instance (G, k) of the* VC-3 *problem, constructs another instance (G_1, k_1) of* VC-3 *with $k_1 \leq k$ and $|G_1| \leq 2k_1$, such that $\tau(G) \leq k$ if and only if $\tau(G_1) \leq k_1$.*

Proposition 2.1 allows us to assume, without loss of generality, that in an instance (G, k) of the VC-3 problem, the graph G contains at most $2k$ vertices.

Let v be a degree-2 vertex in the graph with two neighbors u and w such that u and w are not adjacent. We construct a new graph G' as follows: remove the vertices v, u, and w and introduce a new vertex v_0 that is adjacent to all neighbors of u and w in G (of course except the vertex v). We say that the graph G' is obtained from the graph G by *folding* the vertex v. See Figure 1 for an illustration of this operation. We have the following lemma [6].

Lemma 2.2 ([6]) *Let G' be a graph obtained by folding a degree-2 vertex v in a graph G, where the two neighbors of v are not adjacent to each other. Then $\tau(G) = \tau(G') + 1$.*

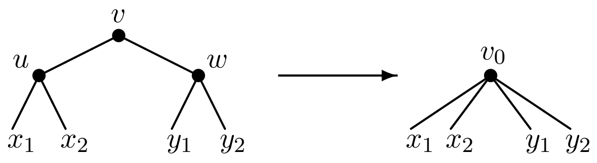

Figure 1: Vertex folding

Following the terminology of Tutte [24], we define the *binding set* of an induced subgraph H of a graph G to be the set of vertices in H that have neighbors not in H. We first discuss how a small induced subgraph with a small binding set helps identifying vertices that are in a minimum vertex cover.

Lemma 2.3 *Let (G, k) be an instance of* VC-3 *where G has no vertex of degree less than 2. If G has an induced subgraph H with a binding set of at most 2 vertices and $4 \leq |H| \leq 50$,*[1] *then in constant time we can construct an instance (G', k') of* VC-3 *with a reduced parameter $k' < k$, such that G has a vertex cover of k vertices if and only if G' has a vertex cover of k' vertices.*

PROOF. First we discuss the case where the binding set of H consists of one vertex v. Consider the algorithm **BindingSet1()** in Figure 2. The algorithm runs in constant time since $|H| \leq 50$. If H has a minimum vertex cover containing the vertex v, then let C_H be this vertex cover, otherwise let C_H be any minimum vertex cover of H. In both cases, removing the vertex set C_H from the graph G (and all isolated vertices resulted from this process) gives the graph G'. Thus, it suffices to show that there is a minimum vertex cover C of the graph G that contains the entire set C_H: in this case $C - C_H$ makes a minimum vertex cover for the graph G' and $|C - C_H| = \tau(G) - \tau(H)$.

```
**BindingSet1**$(G, H, v)$        {∗ $\{v\}$ is the binding set of the induced subgraph $H$ of $G$ ∗}
if $H$ has a minimum vertex cover containing the vertex $v$ then
    $G' = G - H$;  $k' = k - \tau(H)$;
else $G' = G - (H - \{v\})$;  $k' = k - \tau(H)$.
```

Figure 2: Removing an induced subgraph whose binding set has only one vertex

Let C_G be any minimum vertex cover of G, then $C_G \cap V_H$ is a vertex cover for H, and hence $|C_G \cap V_H| \geq \tau(H)$. If the minimum vertex cover C_H of H contains v, then replacing $C_G \cap V_H$ in C_G by C_H gives a minimum vertex cover for G that contains C_H. On the other hand, suppose C_H does not contain v, i.e., H has no minimum vertex cover containing v. Then in case $v \notin C_G$, replacing $C_G \cap V_H$ in C_G by C_H gives a minimum vertex cover for G that contains C_H; while in case $v \in C_G$, $|C_G \cap V_H| \geq \tau(H) + 1$, and replacing $C_G \cap V_H$ in C_G by C_H plus the vertex v gives a minimum vertex cover of G that contains C_H. Thus, in both cases, there is a minimum vertex cover of G that contains C_H. This proves the lemma for the case where H has a binding set of one vertex.

Now suppose the binding set of H has two vertices u and v. Consider the algorithm in Figure 3, which examines all possible situations in which vertices u and v are contained in minimum vertex covers of H.

[1]The constant 50 used here can be replaced by any sufficiently large constant without affecting the correctness of the results in this paper.

```
**BindingSet2**$(G, H, u, v)$        $\{*\ \{u, v\}$ is the binding set of the induced subgraph $H$ of $G\ *\}$
1. if $H$ has a minimum vertex cover $C_1$ containing both $u$ and $v$
      then $G' = G - H$;  $k' = k - \tau(H)$;
2. else if $H$ has a minimum vertex cover $C_2$ containing $u$ but no minimum vertex cover containing $v$
      then $G' = G - (H - \{v\})$;  $k' = k - \tau(H)$;
3. else if $H$ has a minimum vertex cover $C_3$ containing $v$ but no minimum vertex cover containing $u$
      then $G' = G - (H - \{u\})$;  $k' = k - \tau(H)$;
4. else if $H$ has a minimum vertex cover $C_4$ containing $u$ and a minimum vertex cover $C_4'$ containing $v$
      then $G' = G - (H - \{u, v\})$;  $k' = k - \tau(H) + 1$; if $[u, v]$ is not an edge, add an edge $[u, v]$ to $G'$;
5. else $\{*$ every minimum vertex cover of $H$ contains neither $u$ nor $v\ *\}$
      let $\overline{C}_H$ be a smallest vertex cover of $H$ that contains both $u$ and $v$;
      if $|\overline{C}_H| = \tau(H) + 2$ then $G' = G - (H - \{u, v\})$;  $k' = k - \tau(H)$;
                                   else $G' = G - (H - \{u, v\})$;  $k' = k - \tau(H) + 1$;
                                        add a new vertex $w$ and two edges $[w, u]$ and $[w, v]$ to the graph $G'$.
```

Figure 3: Removing an induced subgraph whose binding set consists of two vertices

For each of the cases 1-3, we only need to verify that the corresponding minimum vertex cover of H is entirely contained in a minimum vertex cover of G. For this, let C_G be any minimum vertex cover of the graph G. In case 1, replacing $C_G \cap V_H$ in C_G by C_1 gives a minimum vertex cover of G that contains C_1. For case 2, if C_G does not contain v, then replacing $C_G \cap V_H$ in C_G by C_2 gives a minimum vertex cover for G; while if C_G contains v then $|C_G \cap V_H| \geq \tau(H) + 1$ (since H has no minimum vertex cover containing v), thus replacing $C_G \cap V_H$ in C_G by C_2 plus v gives a minimum vertex cover of G. The proof for case 3 is completely similar to that for case 2.

Consider case 4. Since $[u, v]$ is an edge in G', every vertex cover of G' must contain at least one of u and v. Moreover, if G has a minimum vertex cover C that contains neither u nor v, then replacing $C \cap V_H$ in C by C_4 gives a minimum vertex cover of G that contains u. Thus, the graph G has a minimum vertex cover C_G that contains at least one of u and v. If C_G contains u but not v, replacing $C_G \cap V_H$ in C_G by C_4 gives a minimum vertex cover C_G' for G satisfying that $(C_G' - C_4) \cup \{u\}$ is a minimum vertex cover for G'. Therefore, $\tau(G') = \tau(G) - |C_4| + 1 = \tau(G) - \tau(H) + 1$. The case in which C_G contains v but not u can be verified similarly using C_4' instead of C_4. Finally, suppose that C_G contains both u and v. Since case 1 has been excluded and $C_G \cap V_H$ is a vertex cover for H that contains both u and v, we have $|C_G \cap V_H| \geq \tau(H) + 1$. Therefore, replacing $C_G \cap V_H$ in C_G by C_4 plus v gives a minimum vertex cover C_G'' for G satisfying that $(C_G'' - C_4) \cup \{u\}$ is a minimum vertex cover for the graph G'. Thus again $\tau(G') = \tau(G) - \tau(H) + 1$.

For case 5, let C_H be any minimum vertex cover of H. First consider the subcase $|\overline{C}_H| = \tau(H) + 2$. Let C_G be any minimum vertex cover of G. If C_G contains u but not v, then $|C_G \cap V_H| \geq \tau(H) + 1$ (since no minimum vertex cover of H contains u), so replacing $C_G \cap V_H$ in C_G by C_H plus u gives a minimum vertex cover of G. The case that C_G contains v but not u can be verified similarly. Finally, if C_G contains both u and v, then $|C_G \cap V_H| \geq |\overline{C}_H| = \tau(H) + 2$, and replacing $C_G \cap V_H$ in C_G by C_H plus u and v gives a minimum vertex cover for G. Therefore, in case $|\overline{C}_H| = \tau(H) + 2$, we can simply remove C_H and reduce the parameter by $\tau(H)$. Now consider the subcase $|\overline{C}_H| = \tau(H) + 1$. In this case, the graph G has a minimum vertex cover C_G that either contains both u and v or contains neither: if a minimum vertex cover C of G contains exactly one of u and v, then $|C \cap V_H| \geq \tau(H) + 1$ and replacing $C \cap V_H$ in C by $\overline{C}_H$ gives a minimum vertex cover of G containing both u and v. Moreover, because of the new degree-2 vertex w, the graph G' has a minimum vertex cover C that either contains both u and v or contains neither. If C_G

contains both u and v, replacing $C_G \cap V_H$ in C_G by $\overline{C}_H$ gives a minimum vertex cover C'_G of G satisfying that $(C'_G - \overline{C}_H) \cup \{u, v\}$ is a minimum vertex cover for G' of size $\tau(G) - \tau(H) + 1$, while in case C_G contains neither u nor v, the set $(C_G - C_G \cap V_H) \cup \{w\}$ is a minimum vertex cover for G' of size $\tau(G) - \tau(H) + 1$.

Finally, we note that since $|H| \geq 4$ and G has no vertex of degree less than 2, we have $\tau(H) \geq 2$. Therefore, in all cases we have $k' < k$. □

We note that the condition that the vertex degree is bounded by 3 is not used in the proof of Lemma 2.3. Therefore, the lemma remains valid for instances of the general Vertex Cover problem.

Before we present our main algorithm, we introduce some definitions and terminologies.

Definition 2.4 Let G be a graph in which no vertex has degree larger than 3.

(1) A vertex folding operation is *safe* if it does not create vertices of degree larger than 3.

(2) A cycle of length l in G is an *alternating cycle* if it contains exactly $\lfloor l/2 \rfloor$ degree-2 vertices of which no two are adjacent.

(3) An *alternating tree* T in G is a tree that is an induced subgraph in G such that all degree-1 vertices in T are of degree 3 in G and no two adjacent vertices in T are of the same degree in G. An alternating tree T is *maximal* if no alternating tree contains T as a proper subgraph.

Our main algorithm is a branch-and-search process, given in Figure 4. Each stage of the algorithm starts with an instance (G, k) of VC-3, and tries to reduce the parameter k by identifying a set S of vertices that are entirely contained in a minimum vertex cover of G, and including the vertex set S in the objective minimum vertex cover for G, which will be called *the partial cover* for G, then recursively works on the reduced instances. The subroutine **Fold(v)** simply applies the safe folding operation to a degree-2 vertex v. We also implicitly assume that after each step, the algorithm calls a subroutine **Clean**, which eliminates all isolated vertices and degree-1 vertices (a degree-1 vertex is eliminated by including its neighbor in the partial cover), and updates the graph G, the partial cover, and the parameter k accordingly. In particular, we will assume that at the beginning of each step, the graph contains no vertices of degree less than 2.

If a vertex set S is identified such that either there is a minimum vertex cover containing the entire S or there is a minimum vertex cover containing no vertex in S, then we can *branch on the set S*. This means that the algorithm constructs two instances of VC-3, one by including the set S in the partial cover and the other by excluding the set S from the partial cover, and in the latter case, every vertex that is adjacent to a vertex in S should be included in the partial cover. The algorithm then recursively works on the two reduced instances.

We explain how each step in the subroutine **Reducing** is done efficiently. The conditions in step A and step C can be verified by checking each degree-2 vertex and each edge in the graph G, respectively. The conditions in step B can be verified by partitioning the graph G into connected components. The conditions in step E can be checked in linear time using the following procedure. First we apply a linear time algorithm (see [1], section 5.3) to the graph G, which identifies all cut-points and constructs all 2-connected components of G. By examining each 2-connected component, we can check if there is any induced subgraph with a binding set of a single vertex that satisfies the conditions in step E. Similarly, applying the linear time algorithm in [22] to the graph G identifies all cut-pairs, and constructs all the 3-connected components in G. By examining each 3-connected component, we can find out if there is any induced subgraph with a binding set of two vertices that satisfies the conditions in step E. To check the conditions for step D, we run the following subroutine: first remove from G the set T of all edges whose two ends are of degree 3 in G and

```
VC3-solver
Input: an instance $(G, k)$ of VC-3
Output: a vertex cover $C$ of $G$ of size bounded by $k$ in case it exists
1. while Reducing is applicable do apply Reducing;
2. if there is a maximal alternating tree $T$ of at least 4 vertices in $G$
   then branch on the vertices in $T$ that are of degree 3 in $G$;
3. else if there is a degree-2 vertex $v$ then branch on the two neighbors of $v$;
4. else branch on a degree-3 vertex $v$.

Reducing
A. while there exists a degree-2 vertex $v$ such that folding $v$ is safe do Fold($v$);
B. if $G$ has a component $H$ with $|H| \leq 50$ then include a minimum vertex cover of $H$ in the cover;
C. else if there are two adjacent triangles $(u, v, w)$ and $(u, v, z)$ then include $v$ in the cover;
D. else if there is an alternating cycle $K$ in $G$ then include all degree-3 vertices on $K$ in the cover;
E. else if $G$ has an induced subgraph $H$ with a binding set of at most two vertices, and such that
   $4 \leq |H| \leq 50$ then call the subroutine BindingSet1() or BindingSet2().
```

Figure 4: The algorithm VC3-solver

"smooth" each degree-2 vertex v in G by removing v and adding a new edge connecting its two neighbors. Let the resulting graph be G'. Now every alternating cycle C in the original graph G corresponds either to a cycle in G' (in this case C is of even length) or to an edge $[u, v]$ in T where u and v belong to the same connected component of G' (in this case, C is of odd length). Since the number of edges in G is bounded by $O(k)$, all these conditions can be verified in time $O(k)$.

An explanation for step 2 of the algorithm **VC3-solver** is needed. Because of step 1, there is no alternating cycle in the graph G. Since an alternating tree of at least 4 vertices contains at least one degree-3 vertex in G that is of degree larger than 1 in the tree, we can check each degree-3 vertex in G that has at least two degree-2 neighbors. A simple breadth-first-search style construction from such a degree-3 vertex will give a maximal alternating tree in linear time.

Theorem 2.5 *The algorithm* **VC3-solver** *solves the* VC-3 *problem correctly.*

Proof. We first discuss the subroutine **Reducing**. The correctness of step B is obvious, and the correctness of step A and step E is given by Lemma 2.2 and Lemma 2.3, respectively. For step C, since every minimum vertex cover of G must contain at least one of u and v, by the symmetry in the structure, we can simply include v. Finally, consider step D. Let W be the set of all degree-3 vertices in the alternating cycle K, $|W| = \lceil l/2 \rceil$, where l is the length of K. Since every minimum vertex cover C_G of G contains at least $\lceil l/2 \rceil$ vertices in K, replacing $C_G \cap K$ in C_G by W gives a minimum vertex cover containing W. This verifies the correctness of step D.

What remains is to verify the correctness of each step in the main algorithm **VC3-solver**. For this, we show that, in each of the branching steps 2–4, at least one of the outcomes of the branching includes only vertices in a minimum vertex cover of the current graph, into the partial cover.

In step 4 we branch at a degree-3 vertex v by either including v in the cover, or excluding it and including all its neighbors. This step is correct since for any vertex v in the graph, it is true that any minimum vertex cover either contains v, or does not contain v and contains all its neighbors. For step 3, let u and w be the two neighbors of the vertex v. Each minimum vertex cover C_G of G contains at most two of v, u, and w. If C_G contains only one of v, u, and w, then the vertex in

C_G must be v so both u and w are not in C_G. If C_G contains two of v, u, w, we can always replace these two vertices in C_G by u and w to get a minimum vertex cover of G that contains both u and w. This verifies the correctness of step 3. Finally, consider case 2. Let W be the set of all degree-3 vertices in the alternating tree T. Suppose that G has a minimum vertex cover C_G that contains some vertex v in W but not the entire W. Let N_i be the set of vertices in T such that, for each vertex u in N_i, the unique path from v to u in T has length i. By the definition of an alternating tree, all vertices in N_i are of degree 2 in G if i is odd and of degree 3 in G if i is even. Since v is in the minimum vertex cover C_G, removing v makes all vertices in N_1 become of degree 1. By the observation given earlier, we can safely include all vertices in N_2 in the minimum vertex cover. Now removing all vertices in N_2 makes all vertices in N_3 become of degree 1, so we can include all vertices in N_4 in the minimum vertex cover, and so on. This process will eventually include all vertices in W in the minimum vertex cover, and give a minimum vertex cover of G that contains the entire set W. This verifies that there is a minimum vertex cover of G that either contains the entire set W or contains no vertex in W, and proves the correctness of step 2. □

The main goal of this paper is to show that the number of leaves in the search tree of the algorithm **VC3-solver** on an instance (G, k) of VC-3 is $O(1.194^k)$. This will be done in Proposition 3.11. We first note that the following conditions can be assumed on the input (G, k) to the algorithm **VC3-solver**.

Assumption 2.6 *Let (G, k) be an instance of* VC-3. *We can assume that when the algorithm* **VC3-solver** *is initially called on the instance (G, k) the following holds true:* (1) *the parameter k passed is not larger than the size of a minimum vertex cover of G; and* (2) *G is connected.*

Suppose first that G is connected. Condition (1) can be justified as follows. We start calling the algorithm on G with $k' = 1, 2, \dots, k$. The first time the algorithm returns a vertex cover of size k', we stop (note that the vertex cover returned in this case must be a minimum vertex cover). Otherwise, no vertex cover of size bounded by k exists. Each call to the algorithm satisfies condition (1). It will be shown in Proposition 3.11 that the number of the leaves in the search tree of the algorithm when called on an instance (G, k) is $O(1.194^k)$. The number of leaves in the search tree in the previous calls to the algorithm becomes bounded by $c \cdot 1.194^1 + c \cdot 1.194^2 + \ldots + c \cdot 1.194^k = O(1.194^k)$ (where c is a positive constant). Hence, the upper bound on the number of leaves in the search tree with the new modification to the algorithm is unchanged. Now to justify (2), suppose that there are $G_1, \dots, G_r$ components in G with $|G_i| = n_i$. By Proposition 2.1, we may assume that the size of a minimum vertex cover of G_i, $\tau(G_i)$, is $\geq n_i/2$. We will also assume that $\tau(G_i) \geq 4$ (a component G_i with $\tau(G_i) < 4$ has its size bounded by 8, and thus can be removed in constant time). We call the algorithm on G_1, with $k_1 = n_1/2, n_1/2 + 1, \dots, k$. The first time the algorithm returns a vertex cover of size k_1, we stop. If the algorithm fails to return a vertex cover in each of these cases, then no vertex cover of size bounded by k exists. Otherwise, the algorithm returns a minimum vertex cover of G_1 of size $4 \leq k_1 \leq k$. Now we call the algorithm on G_2 with $k_2 = n_2/2, n_2/2+1, \dots, k - k_1$, and so on. It is now true that on each call to the algorithm on a graph component, conditions (1) and (2) hold true. The number of leaves in the search tree is $O(1.194^{k_1} + \cdots + 1.194^{k_r})$. We show next that $1.194^{k_1} + \cdots + 1.194^{k_r} \leq 1.194^{k_1 + \cdots + k_r}$, which gives that the number of leaves in the search tree is $O(1.194^{k_1 + \cdots + k_r}) = O(1.194^k)$.

Since $k_i \geq 4$ for all i, we have $1.194^{k_i} \geq 2$. For any two numbers $a \geq 2$ and $b \geq 2$, we have $ab - (a + b) = (a - 1)(b - 1) - 1 \geq 0$, which gives $a + b \leq ab$. Using this inequality repeatedly gives

$$\begin{aligned}1.194^{k_1} + 1.194^{k_2} + 1.194^{k_3} + \cdots + 1.194^{k_r} &\leq 1.194^{k_1+k_2} + 1.194^{k_3} + \cdots + 1.194^{k_r} \\ &\leq 1.194^{k_1+k_2+k_3} + \cdots + 1.194^{k_r} \\ &\leq \cdots \leq 1.194^{k_1+k_2+k_3+\cdots+k_r}.\end{aligned}$$

3 Analysis of the algorithm

We analyze the time complexity of the algorithm **VC3-solver** in this section. Let $\mathcal{T}$ be the search tree for the algorithm **VC3-solver** on the input instance (G, k). Let α be a node in the search tree with an associated parameter k'. If we perform a two-sided branch at α by reducing the parameter k' in each branch by a and b, respectively, then such a branch will be called an (a, b) *branch*. We will always assume that in an (a, b) branch, we have $a \leq b$. We say that an (a, b) branch is *not worse than* an (a', b') branch if $a \geq a'$ and $b \geq b'$.

Differing from the common analysis techniques based on the worst-case scenario, we present next a novel way for analyzing the size of the search tree. This can be achieved by looking at the set of operations performed by the algorithm as an interleaved set of operations. This allows us to counter-balance the effect of inefficient operations with efficient ones, thus providing a better upper bound on the size of the search tree. Our goal is to show that the number of leaves in the search tree $\mathcal{T}$ is $O(r^k)$, where $r \leq 1.194$ is the unique positive root of the polynomial $x^k - x^{k-3} - x^{k-5}$, or equivalently, the unique positive root of the polynomial $x^5 - x^2 - 1$.

The graph G is called *clean* if no vertex of degree 0 or 1 exists in G. The graph G is called *nice* if it is clean and no safe folding is applicable to any vertex in G. We will divide the operations performed by the algorithm into four categories.

1. **Folding** operations: the operations performed in step A of the subroutine **Reducing**.
2. $(1, 3)$ **branching** operations: the operations performed in step 4 of **VC3-solver** when we branch on a degree-3 vertex. These operations occur only when the graph becomes 3-regular.
3. $(2, 5)$ **branching** operations: the operations performed in step 3 of **VC3-solver** when we pick a degree-2 vertex and branch on its neighbors. Note that at this point of the algorithm the graph is nice, and hence, no safe folding is applicable. Also, step D of **Reducing** is not applicable. This means that the two vertices that we branch on have five neighbors, and the branch in this case is a $(2, 5)$-branch.
4. The operations performed in: steps B-E of **Reducing**, step 2 of **VC3-solver**, and those performed by the subroutine **Clean**.

The operations will be referred to by their categories. For example, a category-1 operation denotes a folding operation, and a category-4 operation denotes one of the operations listed in number 4 above.

Let i be an operation[2] in any of the above categories. We define the following parameters for operation i: e_i the number of edges removed in operation i, v_i the number of vertices removed

[2]When looking at the search tree, a branching operation will denote the two sides of the branch, whereas when looking at a certain path in the search tree, one side of a branching operation will be considered an operation by itself. It should be clear from the context what is meant by a branching operation (i.e., either one side of the branch or the whole branch).

in operation i, and k_i the reduction in the parameter by operation i. We define the *surplus* s_i of operation i as follows. If i is a non-branching operation that reduces the parameter by k_i, then $s_i = k_i$. If i is the a-side (resp. b-side) of a branching operation (a, b), where $a \leq b$, then $s_i = a - 3$ (resp. $s_i = b - 5$). Informally speaking, s_i is the addition or reduction in the parameter, relative to a $(3, 5)$-branch, that is gained or lost in the operation i. We define the *amortized cost* m_i of operation i by $m_i = 5e_i - 6v_i + 6s_i - 3k_i$. Note that if the operation i is followed by **Clean**, we will combine the amortized cost of **Clean** with m_i. Also note that for any non-branching operation $s_i = k_i$, therefore the amortized cost of such an operation is $m_i = 5e_i - 6v_i + 3k_i$.

The amortized cost m_i defined above will be used to measure the cost related to operation i including the benefit cost generated by operation i, the cost gained by operation i from other previous operations, and the cost relative to attaining the target parameter reduction of the operation. Based on the principle of "gain more then pay more", we use the gain in the parameter reduction related to the operation to measure the corresponding cost. Write $s_i = k_i - \delta_i$, where δ_i is the "target value" in the parameter reduction for operation i (e.g., for an (a, b) branch, where $a \leq b$, the target value for the a-side operation is 3, and for the b-side operation is 5). Rewrite the formula as $m_i = (5e_i - 6v_i) + 3s_i - 3\delta_i$. We consider the three parts in the formula for the amortized cost m_i. (A) The term $(5e_i - 6v_i)$ in m_i: observe that for a clean graph of n vertices and m edges, if the edge/vertex ratio m/n is less than $6/5$, then a safe folding operation is applicable (see Proposition 3.13). Thus, if the operation i removes e_i edges and v_i vertices such that $e_i/v_i > 6/5$ (or, equivalently $5e_i - 6v_i > 0$), then the operation i will lower the edge/vertex ratio in the remaining subgraph and increase the possibility of safe folding, which will benefit later steps of the algorithm. Therefore, the term $(5e_i - 6v_i)$ in m_i describes the cost of the operation i that will benefit later steps of the algorithm. (B) The surplus s_i: the value of s_i represents the gain in the parameter reduction that is beyond the target value. Note that in the algorithm, each operation i with a positive surplus must have taken the advantage of a certain special graph structure which had been generated by previous operations. Moreover, after the operation i, the favored structure disappears. Therefore, the value s_i can be regarded as the cost of previous operations to generate the favored structure consumed by the operation i. For example, a safe folding operation takes the advantage of two adjacent degree-2 vertices (which are generated by previous operations), gains a surplus 1, but eliminates the favored structure (i.e., the two adjacent degree-2 vertices). Therefore, the value s_i describes the cost of previous operations that benefited the operation i. (C) The value δ_i: since the cost of the operation i spent for gaining the target parameter reduction δ_i is excluded from the amortized cost, the term $-\delta_i$ becomes a term in the amortized cost m_i.

Based on the above discussion, it is natural to define the amortized cost as a linear function of $(5e_i - 6v_i)$, s_i, and $-\delta_i$. The remaining question is to determine the coefficients of these terms, i.e., to determine how these terms are proportionally related. We give an intuitive explanation here. The entity s_i counts the extra reduction in the parameter value, and the entity δ_i denotes the targeted reduction in the parameter value. Therefore, both s_i and δ_i refer to the reduction in the parameter value, and hence, it makes sense to give them the same coefficient in the formula for m_i. Now how is the term $(5e_i - 6v_i)$ related to the value s_i (and δ_i)? A careful analysis of the algorithm (see the proofs of Proposition 3.12 and Lemma 3.14) shows that it is proper to equate a value 3 in $(5e_i - 6v_i)$ to a value 1 in s_i. We use the 1-side operation of a $(1, 3)$ branch as an example. Here we have $e_i = 3$ and $v_i = 1$, thus $(5e_i - 6v_i) = 9$. On the other hand, the operation creates three degree-2 vertices, each may induce a folding that reduces the parameter by 1. Therefore, a value 9 in the term $(5e_i - 6v_i)$ seems to correspond to a value 3 in the parameter reduction. The same conclusion can be derived for the 2-side operation of a $(2, 5)$ branch.

The above explains the main intuition behind the formulation of the amortized cost as $m_i =$

$(5e_i - 6v_i) + 3s_i - 3\delta_i$, which is equivalent to $m_i = 5e_i - 6v_i + 6s_i - 3k_i$.

Lemma 3.1 *Let C_0 be a connected component in G, and let m_0 be the amortized cost incurred by invoking* **Clean** *on C_0. If C_0 is not a tree then $m_0 \geq 0$, and if C_0 is a tree then $m_0 \geq -6$.*

PROOF. Suppose first that C_0 is a non-tree connected component in G. Let e_0, v_0, k_0 be the parameters of the operation of applying **Clean** to C_0. Since **Clean** is a non-branching operation, we have $m_0 = 5e_0 - 6v_0 + 3k_0$. If **Clean** removes the whole component C_0, then since C_0 is connected and is not a tree, we have $e_0 \geq v_0$. Also, $k_0 \geq e_0/3$ since every removed edge must be covered by the vertices that have been included in the vertex cover, and each vertex can cover at most 3 edges. It follows that the amortized cost $m_0 = 5e_0 - 6v_0 + 3k_0 \geq 0$. Now suppose that **Clean** does not remove the whole component C_0. Then any connected induced subgraph C' of C_0 that is removed by **Clean** must have at least one edge connecting it to $V(C_0) - V(C')$, which is also removed by **Clean**. It follows that the number of edges e' removed when removing C' is at least as large as the number of vertices v' in C'. Also, the reduction in the parameter k' incurred in C' is $k' \geq e'/3$ by the same argument as above. It follows that the amortized cost m' induced by m_0 on every connected subgraph C' of C removed by **Clean** is non-negative. The amortized cost m_0 on C_0 is the summation of the amortized cost on each connected subgraph removed by **Clean** (this follows from the linearity of the expression for the amortized cost and the monotonicity of addition). It follows that the amortized cost m_0 incurred by cleaning a non-tree component is always non-negative.

Suppose now that C_0 is a tree. In this case **Clean** removes the whole component C_0. It follows that $e_0 = v_0 - 1$. This, together with $k_0 \geq e_0/3$, gives $m_0 = 5e_0 - 6v_0 + 3k_0 \geq -6$. □

Lemma 3.2 *A non-branching operation on a connected component of a clean graph G has a non-negative amortized cost.*

PROOF. Since G is clean, every connected component of G is also clean, and hence, is not a tree. It follows, by a similar argument to that in Lemma 3.1, that the induced amortized cost on every connected subgraph of G removed by the operation plus **Clean** is non-negative. Hence, the total amortized cost is non-negative. □

Fact 3.3 *A tree with exactly two degree-1 vertices is a path between the two degree-1 vertices.*

Lemma 3.4 *On a nice graph G, an operation i performed in step E of* **Reducing** *followed by an invocation to* **Clean** *has a non-negative amortized cost m_i. In particular, the amortized cost of step 4 of the procedure* **BindingSet2()** *is at least 6.*

PROOF. In step E of **Reducing**, the algorithm removes a subgraph from G and possibly adds some edges and vertices to the graph. We need to verify that the amortized cost of such an operation is non-negative. In the cases when the operation does not add any vertices or edges to the graph, the fact that the amortized cost is non-negative follows from Lemma 3.2. We only need to show this statement for step 4 of **BindingSet2()** when one edge is added, and step 5 of **BindingSet2()**, when one vertex and two edges are added. We show the statement for step 4 of **BindingSet2()**. The proof that this statement holds true for step 5 of **BindingSet2()** is very similar. The operation in step 4 removes $(H - \{u, v\})$ from G and adds an edge $[u, v]$ if this edge does not already exist. If the edge $[u, v]$ already exists, then no edge is added and we are done. Suppose that there is no edge $[u, v]$ in G. Note that H cannot be a tree, otherwise, since the operation is performed on a clean connected component of the graph, H would have exactly two degree-1 vertices namely u and v, and by Fact 3.3, H must be a chain (note that a tree with more than one vertex must have at least two degree-1 vertices). Since $|H| \geq 4$, this would imply that there were two adjacent degree-2 vertices in the graph prior to this operation contradicting the fact that no safe folding is applicable at this stage of the algorithm. Thus, we must have $e_H \geq v_H$, where e_H and v_H are the number of edges and vertices in H, respectively. The operation removes $e_H - 1$ edges (e_H edges from H, and $[u, v]$ is added), $v_H - 2$ vertices, and reduces the parameter by k_H. Since each of the k_H vertices included in the vertex cover can cover at most 3 edges, we must have $k_H \geq (e_H - 1)/3$. Since the operation is a non-branching operation, its amortized cost $m_i = 5(e_H - 1) - 6(v_H - 2) + 3k_H \geq 6e_H - 6v_H + 6 \geq 6$. Also, since prior to this operation the graph was clean, the resulting graph is also clean, and hence, the subroutine **Clean** is not applicable. This completes the proof. □

Proposition 3.5 *Let G be a nice graph, and let $\mathcal{S}$ be a collection of disjoint induced trees in G that are joined to $G - \mathcal{S}$ by l edges. Then $|\mathcal{S}| \leq 4l - 7$.*

PROOF. It suffices to prove the proposition for the case when $\mathcal{S}$ contains a single induced tree T. The proof for the general case follows by applying the statement to each induced tree in $\mathcal{S}$.

For an induced tree T, let L_T be the set of vertices of degree less than 2 in the tree T, and let C_T be the set of edges with one end in T and the other end in $G - T$. We prove, by induction on $|T|$, the following statement:

> **Statement A.** $|T| \leq 4|C_T| - 7$. More precisely, if a vertex in L_T has degree 3 in the graph G, then $|T| \leq 4|C_T| - 10$, and if all vertices in L_T have degree less than 3 in G, then $|T| \leq 4|C_T| - 7$.

First note that the graph G is nice, and hence, G has no vertices of degree less than 2. When $|T| = 1$, if the vertex v in T has degree 3 in G then $|C_T| = 3$, and if the vertex v has degree 2 in G then $|C_T| = 2$. Therefore, **Statement A** holds true when $|T| = 1$. When $|T| = 2$, T consists of a single edge $[u, w]$, and $|C_T| \geq 3$, since the nice graph G cannot have two adjacent degree-2 vertices u and w. Therefore, **Statement A** holds true when $|T| = 2$.

Now consider the general case $|T| \geq 3$. First suppose that there is a vertex w in L_T such that w is of degree 3 in G. Then one edge $[w, u]$ incident on w is in T (because $|T| > 1$), and the other two edges $[w, w_1]$ and $[w, w_2]$ incident on w belong to C_T. Consider the tree $T' = T - \{w\}$ in G. We have $|T'| = |T| - 1$ and $|C_{T'}| = |C_T| - 1$ ($C_{T'}$ is obtained from C_T by removing the two edges $[w, w_1]$ and $[w, w_2]$ and adding the edge $[w, u]$). By the inductive hypothesis, $|T'| \leq 4|C_{T'}| - 7$, which gives directly that $|T| \leq 4|C_T| - 10$.

Now suppose that all vertices in L_T have degree 2 in G. Pick a longest path in T with endpoints r and w. Both r and w must be in L_T, and hence, have degree 2 in G. Let u be the neighbor of w in the tree T (the vertex u must exist, and must be different from r, because $|T| \geq 3$ and a longest path in T from r to w has length at least 2).

Let the other edge incident on w be $[w, w_1]$. Since the graph G is nice, the vertex u must be of degree 3 in G (otherwise, w and u would be two adjacent degree-2 vertices in G). Let the edge incident on u but not on the path joining r to w be $[u, u_1]$. If u_1 is not in T, then consider the tree $T' = T - \{w\}$, and note that u is in $L_{T'}$. We have $|T'| = |T| - 1$, and $|C_{T'}| = |C_T|$ ($C_{T'}$ is obtained from C_T by removing the edge $[w, w_1]$ and adding the edge $[u, w]$). Since u is of degree 3 in G, by the inductive hypothesis, we have $|T'| \leq 4|C_{T'}| - 10$, which gives $|T| \leq 4|C_T| - 9 < 4|C_T| - 7$. Suppose now that u_1 is in the tree T. Then u_1 must be in L_T (otherwise, the path from r to w

would not be a longest path in T), and u_1 has degree 2 in G. Consider the tree $T'' = T - \{w, u_1\}$ in G. We have $|T''| = |T| - 2$, and $|C_{T''}| = |C_T|$ ($C_{T''}$ is obtained from C_T by removing the edge $[w, w_1]$ and the edge joining u_1 to $G - T$, and adding two edges $[u, w]$ and $[u, u_1]$). Now the vertex u is in $L_{T''}$, and u is of degree 3 in G. By the inductive hypothesis, $|T''| \leq 4|C_{T''}| - 10$, which gives $|T| \leq 4|C_T| - 8 < 4|C_T| - 7$.

This completes the inductive proof of **Statement A** and the proof of the proposition. $\square$

Lemma 3.6 *On a nice graph G, an operation i performed in step 2 of* **VC3-solver** *followed by an invocation to* **Clean** *is not worse than a $(3, 5)$-branch, and its amortized cost m_i is non-negative.*

PROOF. We first prove a general result for alternating trees. Suppose that T is an alternating tree with at least 3 vertices. Let D_2 and D_3 be the sets of vertices in T of degree 2 and degree 3 in G, respectively, and let $x = |D_3|$. Let Y be the set of neighbors of D_3 that are not in T, i.e., $Y = N(D_3) - D_2$, and let $y = |Y|$. We first show, by induction on $|T|$, that (1) $|D_2| = x - 1$ and hence $|T| = 2x - 1$; and (2) there are exactly $(x + 2)$ edges between T and Y.

For the base case $|T| = 3$, from the definition of an alternating tree, the tree T must be a chain $[u_1, u_2, u_3]$ of three vertices, where u_1 and u_3 are of degree 1 in T and degree 3 in G, and u_2 is of degree 2 in both T and G. Moreover, there are four edges joining T to $G - T$, namely those edges joining u_1 and u_3 to the vertices in $G - T$. Therefore, we have $x = |D_3| = 2$, $|D_2| = 1$, and the number of edges between T and Y is 4. Thus, statements (1) and (2) hold true in this case.

We note that the case $|T| = 4$ is impossible: if T has three degree-1 vertices (which are of degree 3 in G), then the fourth vertex in T must be connected to all the three degree-1 vertices, and hence cannot be of degree 2, so T is not an alternating tree; while if T has two degree-1 vertices, then the other two vertices in T must be of degree 2 and adjacent, so again T would not be an alternating tree.

Therefore, for a general case for an alternating tree T with $|T| > 3$, we must have $|T| \geq 5$. Let w be any vertex of degree 1 in T. By the definition of alternating trees, w is of degree 3 in the graph G. The vertex w is adjacent to a degree-2 vertex u in the tree T and adjacent to two other vertices w_1 and w_2 in $G - T$. Let the other neighbor of u in T be u_1, which is a degree-3 vertex in G. Consider the tree $T' = T - \{w, u\}$ in G. Then $|T'| = |T| - 2 \geq 3$. Moreover, the tree T' is an alternating tree: the degree-3 vertex u_1 now becomes of degree 1 in T', and the degrees of the vertices in T' still alternate. Let D'_2 and D'_3 be the sets of vertices in T' of degree 2 and degree 3 in G, respectively. Then $|D'_2| = |D_2| - 1$ and $|D'_3| = |D_3| - 1$. Moreover, the number of edges β' between T' and $G - T'$ is exactly one less than the number of edges β between T and $G - T$ (the set of edges between T' and $G - T'$ is obtained from the set of edges between T and $G - T$ by removing the two edges $[w, w_1]$ and $[w, w_2]$ and adding the edge $[u_1, u]$). By the inductive hypothesis, we have $|D'_2| = |D'_3| - 1$ and $\beta' = |D'_3| + 2$, which gives directly that $|D_2| = |D_3| - 1 = x - 1$ and $\beta = |D_3| + 2 = x + 2$. This completes the proof of statements (1) and (2).

Now we are ready to prove the statement of the lemma. Since the number of vertices in an alternating tree is $2x - 1$, which is an odd number, and since $|T|$ is assumed to be ≥ 4 in step 2 of **VC3-solver**, we have $|T| \geq 5$, and hence, $x \geq 3$. Part (2), and the fact that $x \geq 3$, imply that there are at least five edges between T and Y. Since every vertex in the graph has degree bounded by 3, we have $y \geq 2$.

If $y = 2$, then $x \leq 4$, and the subgraph H induced by $V(T) \cup Y$ has size at most 9. Since no isolated components of size ≤ 50 exist at this point of the algorithm by step B in **Reducing**, the binding set of H has size bounded by 2 (the binding set of H is a subset of Y). Since $4 \leq |H| \leq 50$, this is not possible at this point of the algorithm by step E of **Reducing**. It follows that $y \geq 3$, and branching in step 2 of **VC3-solver** on D_3 gives a $(|D_3|, |D_2| + |Y|) = (x, x-1+y)$ branch, which is not worse than a $(3,5)$-branch since both x and y are at least 3.

What is left is showing that the amortized cost m_i of operation i is non-negative. Consider first the side of the branch where we include the vertices in D_3 in the partial cover. The vertices removed by this branch are those in T whose number is $v_i = 2x - 1$. The edges removed are those in T plus the edges between T and Y. These edges are exactly the edges incident on the vertices in D_3. Since no two degree-3 vertices in T are adjacent, it follows that the number of edges e_i removed by the branch is $3x$. Moreover, the reduction k_i in the parameter is x, and the surplus is $x - 3$. Now let $\mathcal{S}$ be the set of tree components in the resulting graph $G - T$, and let t_i be the number of tree components in $\mathcal{S}$. By Lemma 3.1, the amortized cost of **Clean** on a non-tree component is non-negative, and on a tree component is at least -6. It follows that the amortized cost of operation i including the invocation of **Clean** is

$$\begin{aligned} m_i &\geq 5e_i - 6v_i + 6s_i - 3k_i - 6t_i \\ &\geq 5(3x) - 6(2x-1) + 6(x-3) - 3x - 6t_i \\ &= 6x - 12 - 6t_i. \end{aligned} \tag{1}$$

Observe that the tree components in $\mathcal{S}$ are disjoint, and each tree component must be connected by at least two edges to T (since no degree-1 vertices exist in G). It follows from this observation that there cannot be more than $\lfloor (x+2)/2 \rfloor$ tree components in $\mathcal{S}$, and hence, $t_i \leq \lfloor (x+2)/2 \rfloor$. If $x \geq 6$, then from Inequality (1), we get $m_i \geq 0$. Suppose now that $x \leq 5$. We claim that in this case either there exists a non-tree component in $G - T$ that is joined to T by at least three edges, or there exist at least two non-tree components in $G - T$. If all components in $G - T$ are tree components, i.e., $G - T = \mathcal{S}$, then $\mathcal{S}$ is a collection of disjoint induced trees that are joined to T by at most $x + 2 \leq 7$ edges satisfying the conditions of Proposition 3.5 with $l = 7$. It follows in this case that the number of vertices in $\mathcal{S}$ is bounded by 21, and hence, the total number of vertices in the graph component induced by $V(T) \cup V(\mathcal{S})$ is bounded by 30. This is not possible at this point of the algorithm due to the fact that step B in **Reducing** was not applicable. Now suppose that there is exactly one non-tree component C_0 in $G - T$ that is joined by exactly two edges to T. By a similar argument to the above, the graph induced by $V(T) \cup V(\mathcal{S})$ has at most 22 vertices (and at least 4 vertices), and is connected to C_0 by exactly two edges, which means that it has a binding set of size at most 2. This is again not possible by step E of **Reducing**. It follows that the claim holds true. An immediate consequence of this claim is that $t_i \leq \lfloor (x+2-3)/2 \rfloor = \lfloor (x-1)/2 \rfloor$. Combining this with (1), we get $m_i \geq 3x - 9 \geq 0$ because $x \geq 3$.

Now on the other side of the branch we include the neighbors of D_3: D_2 and Y. Let e_Y be the number of edges connecting the vertices of Y, and z the number of edges between the graph induced by $V(T) \cup Y$ and the remaining graph. It is not difficult to verify that in this side of the branch the number of edges e_i removed is $3x + z + e_Y$, the number of vertices v_i removed is $2x - 1 + y$, and the reduction in the parameter k_i is $x - 1 + y$. Let $\mathcal{S}$ be the set of tree components in $(G - T) - Y$, and t_i the number of tree components in $\mathcal{S}$. Now

$$\begin{aligned} m_i &\geq 5e_i - 6v_i + 6s_i - 3k_i - 6t_i \\ &\geq 5(3x + z + e_Y) - 6(2x - 1 + y) + 6(x - 1 + y - 5) - 3(x - 1 + y) - 6t_i \\ &\geq 6x - 3y + 5e_Y + 5z - 6t_i - 27. \end{aligned} \tag{2}$$

Since the alternating tree is maximal, all vertices in Y have degree 3. By counting the sum of the degrees of the vertices in Y, we get

$$3y = x + 2 + z + 2e_Y. \tag{3}$$

Combining (2) and (3) and noting that $t_i \leq \lfloor z/2 \rfloor$, we get

$$\begin{aligned} m_i &\geq 5x + 3e_Y + 4z - 6t_i - 29 &(4)\\ &\geq 5x + z + 3e_Y - 29. &(5) \end{aligned}$$

If $x \geq 6$, then from Inequality (5) we have $m_i \geq 0$. If $x = 5$, then from Inequality (5), the fact that $z \geq 3$ (note that if $z \leq 2$ then the graph induced by $V(T) \cup Y$ has size bounded by 50 and a binding set of size at most 2), Equality (3), and the fact that y is an integer, we have $m_i \geq 0$. If $x = 4$ and $z \geq 9$, then again by Inequality (5), $m_i \geq 0$. We are left with the cases $x = 4$ and $z < 9$, or $x = 3$. If $x = 3$, then $z \leq 10$, because there cannot be more than 5 vertices in Y each of which has to be joined by at least one edge to T. It follows that in both cases $z \leq 10$ and $|V(T) \cup Y \cup V(\mathcal{S})| \leq 50$ (since $|\mathcal{S}| \leq 33$ by Proposition 3.5). By an argument similar to the above, we must have at least two non-tree components in $G - (V(T) \cup Y)$, or a non-tree component that is joined to Y by at least three edges. It follows that $t_i \leq \lfloor (z-3)/2 \rfloor$. Combining this with Inequality (4), we get

$$\begin{aligned} m_i &\geq 5x + 3e_Y + 4z - 6\lfloor (z-3)/2 \rfloor - 29 &(6)\\ &\geq 5x + z + 3e_Y - 20. &(7) \end{aligned}$$

Since $x \geq 3$ and $z \geq 3$, if $x = 4$, $z \geq 5$, or $e_Y \geq 2$, by (7) we get $m_i \geq 0$. Assume now that $x = 3$, $z \in \{3, 4\}$, and $e_Y \in \{0, 1\}$. Because $x,\ y,\ z,\ e_Y$ are all integers, it is easy to see from (3), that the only possible case is when $x = 3,\ y = 3,\ z = 4,\ e_Y = 0$. Substituting these values in (6), we get $m_i \geq 2$.

It follows that branch i is not worse than a $(3, 5)$-branch, and the amortized cost of i including the invocation to **Clean** is non-negative. This completes the proof. □

Theorem 3.7 *Let i be an operation performed in one of steps B-E in* **Reducing***, or step 2 in* **VC3-solver** *followed by an invocation to* **Clean***. Then the amortized cost m_i of i is non-negative.*

Proof. By Lemma 3.2, the amortized cost corresponding to any non-branching operation is non-negative. In particular, the amortized cost corresponding to an operation performed in any of steps B-D of **Reducing** is non-negative. Lemma 3.4 shows that step E of **Reducing** followed by an invocation to **Clean** has a non-negative amortized cost (note that Lemma 3.2 cannot be applied to an operation in step E since such an operation may add edges and vertices to the graph). Lemma 3.6 establishes the same facts for step 2 of **VC3-solver**. □

Proposition 3.8 *Let O be an operation that removes e_0 edges, v_0 vertices, reduces the parameter by k_0, and has surplus s_0. Let $m_0 = 5e_0 - 6v_0 + 6s_0 - 3k_0$ be the amortized cost of operation O.*

(i) *If O is a category-1 operation then $m_0 \geq 1$.*
(ii) *If O is the 1-side branch in a category-2 operation then $m_0 = -6$.*
(iii) *If O is the 3-side branch in a category-2 operation then $m_0 \geq -6$.*
(iv) *If O is the 2-side branch in a category-3 operation then $m_0 = 0$.*
(v) *If O is the 5-side branch in a category-3 operation then $m_0 \geq 1$.*
(vi) *If O is a category-4 operation, then $m_0 \geq 0$.*

Proof. A folding operation removes at least two edges and two vertices. Hence, $e_0 \geq 2$ and $v_0 = 2$. In both cases we have $s_0 = k_0 = 1$ (since there is no branching). It follows that $m_0 \geq 1$. Now in the 1-side of the $(1, 3)$-branch it is always the case that exactly one vertex and three edges are removed. Since $s_0 = -2$ and $k_0 = 1$, we have $m_0 = -6$. Also, the remaining graph is clean, and **Clean** is not applicable. Similarly for the 2-side of the $(2, 5)$-branch, when we branch on the two neighbors w_1 and w_2 of a degree-2 vertex w, 6 edges and 3 vertices are removed, and no degree-1 vertices are created since all the other neighbors of the two vertices w_1 and w_2 must be of degree 3 (otherwise we would have an alternating tree of size at least 5, which is not possible since step 2 of **VC3-solver** was not applicable). Since $s_0 = -1$ and $k_0 = 2$, we have $m_0 = 0$. In all the above cases, the subroutine **Clean** is not applicable since all the remaining vertices have degrees larger than one. This proves parts $(i), (ii), (iv)$.

To prove part (iii), note first that in the 3-side of the $(1, 3)$ branching we have $s_0 = -2$ and $k_0 = 3$. Also, we know that before this operation the graph G is 3-regular. Let u be the degree-3 vertex that we branch on, and let v, w, z be its neighbors. Let H be the graph induced by $\{u, v, w, z\}$. Since **Reducing** does not apply at this point, there cannot be more than one edge among v, w, z (otherwise, we would have two adjacent triangles). Suppose that there exists one edge among v, w, z. This means that there are exactly four edges connecting H to $G - H$. Note that in this case no component in $G - H$ can be a tree, otherwise, using Proposition 3.5, the graph induced by the vertices of the tree component plus the vertices of H has size bounded by 50, and is connected to the remaining graph by at most two edges (since the tree component has to be connected to $\{v, w, z\}$ by at least two edges), which is not possible at this stage of the algorithm since steps B-E of **Reducing** do not apply. Thus, we can assume that no component in $G - H$ is a tree, and hence by Lemma 3.1, the amortized cost of **Clean** in case it is invoked is non-negative. The number of edges and vertices removed in this case is 8 and 4, respectively, giving $m_0 \geq 5e_0 - 6v_0 - 21 = -5$.

Now suppose that no edge exists among v, w, z, and hence, there are exactly six edges connecting H to $G - H$. By a similar argument to the above, we cannot have two different components in $G - H$ that are trees. Thus, in the worst case, the amortized cost of **Clean** is at least -6 by Lemma 3.1. The branch itself removes 9 edges and 4 vertices from the graph. Since the total amortized cost is the sum of the amortized cost of the branch and that of **Clean**, it follows that $m_0 \geq 5e_0 - 6v_0 - 27 = -6$.

Now we look at part (v) which is the 5-side of the $(2, 5)$-branch. Note that in this case we have $s_0 = 0$ and $k_0 = 5$. Let u be the degree-2 vertex that we branch on its two neighbors v and w. Let v_1 and v_2 be the neighbors of v other than u, and w_1 and w_2 be those of w. Observe that since folding is not applicable, v and w must be of degree 3 and they do not share any neighbors except u. Also, since no alternating tree of size ≥ 5 exists at this point, v_1, v_2, w_1, w_2 must be all of degree 3. Let H be the graph induced by $\{u, v, w, v_1, v_2, w_1, w_2\}$. If there are more than two edges among the vertices $\{v_1, v_2, w_1, w_2\}$, the graph H, which has size bounded by 50, would be connected to $G - H$ by at most two edges, which is not possible at this stage of the algorithm (because no induced subgraph with a binding set of size at most two exists). If the number of edges between $\{v_1, v_2, w_1, w_2\}$ is two, then there are exactly four edges connecting H to $G - H$. By a similar argument to the above, there cannot be any tree component in $G - H$, otherwise, there will be at most two edges connecting H and the tree (having size bounded by 50), to the remaining graph. The number of edges and vertices removed in this case is 12 and 7 giving $m_0 \geq 3$, and the amortized cost of **Clean** is positive (since there is no tree component). Now suppose there is exactly one edge between $\{v_1, v_2, w_1, w_2\}$. In this case the number of edges between H and $G - H$ is exactly six, and the number of edges and vertices removed is 13 and 7. By the same token, there cannot be two tree components in $G - H$, and hence the amortized cost of **Clean** is at least -6 by Lemma 3.1. This gives $m_0 \geq 5e_0 - 6v_0 - 21 = 2$. If there are no edges among $\{v_1, v_2, w_1, w_2\}$, then there are exactly eight edges connecting H to $G - H$, and the number of edges and vertices removed is 14 and 7. Again, we cannot have more than two tree components in $G - H$ giving an amortized cost of at least -12 for **Clean**. This gives $m_0 \geq 5e_0 - 6v_0 - 27 = 1$. It follows that in all cases of the branch $m_0 \geq 1$.

To prove part (vi), note that a category-4 operation is either an operation performed in steps B-E of **Reducing** followed by an invocation to **Clean**, an operation performed in step 2 of **VC3-solver** followed by an invocation to **Clean**, or one that is performed in **Clean**. If O is an operation that is performed in steps B-E of **Reducing** or in step 2 of **VC3-solver**, then by Theorem 3.7, the amortized cost of O including the call to **Clean** is non-negative. Now if O is an operation in **Clean** that does not follow an operation in steps B-E of **Reducing** or step 2 of **VC3-solver**, by the above discussion, O must be an operation following a 3-side of a $(1, 3)$-branch, or a 5-side of a $(2, 5)$-branch (these cover all the cases in which **Clean** is called). By parts (iii) and (v) above, the negative part of the amortized cost of **Clean** was combined with the amortized cost of the operation itself, and the remaining part is positive. This completes the proof. □

Based on Proposition 3.8, we give in Figure 5 the parameters for any operation i in the four categories. If operation i is a category-4 operation (or one side of a category-4 operation), then we denote its surplus by s_i, reduction in the parameter by k_i, and amortized cost by m_i. For every operation, a lower bound on its amortized cost is given in the last column of the table.

Operations		reduction in k	surplus	amortized cost
Folding		1	1	1
$(1, 3)$ branching	1-side	1	-2	-6
	3-side	3	-2	-6
$(2, 5)$ branching	2-side	2	-1	0
	5-side	5	0	1
A category-4 operation i		k_i	s_i	0

Figure 5: The parameters of the operations

Each non-root node α in a search tree $\mathcal{T}$ for the algorithm **VC3-solver** uniquely *specifies* the operation in the algorithm from the parent of α to α. Therefore, each operation in the algorithm can be uniquely referred to by the corresponding node in the tree $\mathcal{T}$. To simplify the description, we also assume that the root of $\mathcal{T}$ has a "virtual" parent associated with the input (G, k) to the algorithm, and that the root of $\mathcal{T}$ specifies a "dummy" operation whose parameter reduction, surplus, and amortized cost, are all equal to 0. Thus, every node in the search tree (including the root) has a parent. By saying *the operations on a path P* in the search tree $\mathcal{T}$, we will be referring to the operations specified by the nodes on P. The reader should note the distinction between *the operation specified by a node* and *the instance (G', k') associated with the node* (i.e, the resulting graph G' and the parameter value k' at the node). In particular, the operation specified by a node is actually the operation applied to the instance associated with the parent of the node.

Definition 3.9 In a search tree $\mathcal{T}$ of the algorithm **VC3-solver**, we assign to each node α a *label* whose value is equal to the parameter reduction of the operation specified by the node α. More precisely, if the operation from the parent of a node α in $\mathcal{T}$ to α is the a-side (resp. the b-side) of an (a, b) branch, then the label of α is a (resp. b); if the operation from the parent of α to α is a non-branching operation that reduces the parameter value by c, then the label of α is c. As discussed above, the root of $\mathcal{T}$ specifies a dummy operation whose parameter reduction is 0, and hence, the label of the root is 0.

Let P be a path in a search tree $\mathcal{T}$. Denote by $x_1(P)$ the number of nodes on P with label 1, specifying the 1-side operations of (1, 3) branches. Similarly, denote by $x_3(P)$ and $x_2(P)$ the number of nodes on P with labels 3 and 2, specifying the 3-side operations of (1, 3) branches and the 2-side operation of (2, 5) branches, respectively. Finally, denote by $d(P)$ the sum of the surplus of all other operations (i.e., the operations in categories 1 and 4) on the path.

Definition 3.10 Let P be a path in a search tree $\mathcal{T}$ of the algorithm **VC3-solver**. The *surplus* of the path P, denoted by $\text{Surp}(P)$, is equal to the sum of the surplus of all the operations on P: $\text{Surp}(P) = d(P) - (2x_1(P) + 2x_3(P) + x_2(P))$. The path P is said to be *compressible* if $\text{Surp}(P) \geq 0$.

To justify the formula given in the definition of $\text{Surp}(P)$, note that $d(P)$ is the sum of the surplus of the category-1 and category-4 operations on P. Each side of a $(1, 3)$ branch has surplus -2, and the total surplus of the category-2 operations on P is $-2x_1(P) - 2x_3(P)$. The 2-side (resp. the 5-side) of a $(2, 5)$ branch has surplus -1 (resp. 0), and the total surplus of the category-3 operations on P is $-x_2(P)$. This justifies why the given formula for $\text{Surp}(P)$ captures the total value of the surplus on the whole path P. Intuitively speaking, in comparison to a $(3, 5)$ branch, the 1-side (resp. the 3-side) of a $(1, 3)$ branch "loses" a value 2 in the parameter reduction when compared with the 3-side (resp. the 5-side) of the $(3, 5)$ branch, and the 2-side of a $(2, 5)$ branch "loses" a value 1 in the parameter reduction when compared with the 3-side of the $(3, 5)$ branch. On the other hand, the value $d(P)$ corresponds to the "extra" parameter reduction we gain in comparison to $(3, 5)$ branches. Therefore, the surplus $\text{Surp}(P)$ of a path P measures how much the "extra" gain in the parameter value can make up for the losses along the path.

Proposition 3.11 *Let $\mathcal{T}$ be the search tree for the algorithm* **VC3-solver** *on input (G, k). If every root-leaf path in $\mathcal{T}$ is compressible, then the number of leaves in $\mathcal{T}$ is bounded by r_0^k, where $r_0 \leq 1.194$ is the unique positive root of the polynomial $x^5 - x^2 - 1$.*

PROOF. First note that according to the algorithm **VC3-solver**, each branch node in $\mathcal{T}$ is either a $(1, 3)$ branch, a $(2, 5)$ branch, or an (a, b) branch that is not worse than a $(3, 5)$ branch (see Lemma 3.6). We say that a search tree $\mathcal{T}_0$ is *normalized* if: (1) for every 1-child node α in $\mathcal{T}_0$, the child of α is a leaf; and (2) every branch node in $\mathcal{T}_0$ is either a $(1, 3)$, a $(2, 5)$, or a $(3, 5)$ branch. We can use the following procedure to convert a general search tree $\mathcal{T}$ into a normalized search tree $\mathcal{T}_0$, with a one-to-one correspondence between the root-leaf paths in the two trees, and such that the corresponding root-leaf paths in the two trees have the same surplus. Let the leaves of the original search tree $\mathcal{T}$ be $\alpha_1, \ldots, \alpha_t$. We first construct, based on the tree $\mathcal{T}$, a search tree $\mathcal{T}'$ with leaves $\alpha_1', \ldots, \alpha_t'$, as follows. For each i, let the path from the root to the leaf α_i in $\mathcal{T}$ be P_i. If $d(P_i) = 0$, then leave the path P_i unchanged and let α_i' in $\mathcal{T}'$ be α_i. If $d(P_i) > 0$, then add to P_i a new leaf α_i' with label $d(P_i)$ and make α_i' the unique child of α_i (thus α_i becomes a 1-child non-leaf node in $\mathcal{T}'$). To obtain the normalized tree $\mathcal{T}_0$, we further perform the following two operations on $\mathcal{T}'$: (1) convert each (a, b) branch node α that is not worse than a $(3, 5)$ branch into a $(3, 5)$ branch by giving the label 3 (resp. the label 5) to the child of α corresponding to the a-side (resp. b-side) of α; (2) remove all non-branching nodes: for each 1-child node α in the tree with a child β, where β is not a new leaf created in $\mathcal{T}'$, remove the edge $[\alpha, \beta]$, merge the two nodes α and β, and assign a label to the resulting (new) node equal to the label of α (this corresponds to removing the non-branching operation specified by β). The resulting tree $\mathcal{T}_0$, with leaves $\alpha'_1, \ldots, \alpha'_t$, is a normalized search tree.

Let P_i be the path from the root to the leaf α_i in $\mathcal{T}$ and let P'_i be the path from the root to the leaf α'_i in $\mathcal{T}_0$. Since no $(1,3)$ branch nodes or $(2,5)$ branch nodes are changed or re-labeled in the above procedure, we have $x_1(P_i) = x_1(P'_i)$, $x_3(P_i) = x_3(P'_i)$, and $x_2(P_i) = x_2(P'_i)$. Moreover, if $d(P_i) = 0$, then the operations in steps (1) and (2) above are not applicable to P_i. Therefore, the path P'_i is the same as P_i, and $d(P'_i) = 0$. On the other hand, if $d(P_i) > 0$, then by our construction, the only node on P'_i that is not a $(1,3)$, a $(2,5)$, or a $(3,5)$ branch is the 1-child node whose child is the leaf α'_i with a label $d(P_i)$. Thus, $d(P'_i) = d(P_i)$, and the paths P_i and P'_i have the same surplus.

From the above discussion, for each general search tree satisfying the condition in the proposition, there is a normalized search tree with the same number of leaves that also satisfies the condition in the proposition. Thus, it suffices to prove the proposition for normalized search trees. We do this by induction on the number of nodes in a normalized search tree $\mathcal{T}$. The proposition certainly holds true if the tree $\mathcal{T}$ consists of a single node or has only one leaf. Now assume that $|\mathcal{T}| > 1$ and that $\mathcal{T}$ has more than one leaf. Since $\mathcal{T}$ is normalized, the root α of $\mathcal{T}$ must be a branch node, which is either a $(1,3)$, a $(2,5)$, or a $(3,5)$ branch node.

Suppose the root α of $\mathcal{T}$ is a $(1,3)$ branch. Let β_1 and β_3 be the children of α labeled 1 and 3, respectively. Let $\mathcal{T}_1$ be the subtree rooted at β_1 in $\mathcal{T}$. Every path P_i from the root α to a leaf α_i in $\mathcal{T}_1$ contains the node β_1, and hence $x_1(P_i) \geq 1$. Since the path P_i is compressible, we have $\mathrm{Surp}(P_i) = d(P_i) - (2x_1(P_i) + 2x_3(P_i) + x_2(P_i)) \geq 0$. It follows that $d(P_i) \geq 2$, and the label of the leaf α_i is at least 2. Therefore, in the tree $\mathcal{T}$ we can "shift" 2 units from the label of each leaf in the subtree $\mathcal{T}_1$ to the node β_1, by adding 2 units to the label of β_1 and subtracting 2 units from the label of every leaf in $\mathcal{T}_1$. Now the label of β_1 becomes 3. Similarly, we can add 2 units to the label of the node β_3 and subtract 2 units from the label of every leaf in the subtree rooted at β_3. This makes the label of β_3 become 5. Note that the resulting search tree is still normalized, with the difference that the root α now becomes a $(3,5)$ branch node, and that the label of each leaf in $\mathcal{T}$ is decreased by 2. In particular, each root-leaf path P_i in the resulting tree is still compressible (with the value $x_1(P_i)$ or $x_3(P_i)$ decreased by 1 and the value $d(P_i)$ decreased by 2).

Similarly, if the root α of the tree $\mathcal{T}$ is a $(2,5)$ branch with its label-2 child β_2 corresponding to the 2-side of the branch, then we can decrease the label of each leaf in the subtree rooted at β_2 by 1, add 1 to the label of β_2, and make the root α a $(3,5)$ branch. All root-leaf paths remain compressible.

Therefore, we can always end up with a normalized search tree $\mathcal{T}$ whose root is a $(3,5)$ branch in which all root-leaf paths are compressible. Let γ_3 be the child of α labeled by 3 and γ_5 be the child of α labeled by 5. Consider the subtree $\mathcal{T}_3$ rooted at γ_3. By re-setting the label of γ_3 to 0, the subtree $\mathcal{T}_3$ becomes a valid normalized search tree for the algorithm **VC3-solver** on input $(G', k-3)$, where G' is the graph resulting from G by the operation specified by γ_3. Moreover, each root-leaf path in $\mathcal{T}_3$ is compressible since the node γ_3 in $\mathcal{T}$ is not a child of a $(1,3)$ branch or a $(2,5)$ branch node. Now by the inductive hypothesis, the number of leaves in $\mathcal{T}_3$ is bounded by r_0^{k-3}, where r_0 is the unique positive root of the polynomial $x^5 - x^2 - 1$. Similarly, re-setting the label of γ_5 to 0 makes the subtree rooted at γ_5 a valid normalized search tree with no more than r_0^{k-5} leaves. Adding the number of the leaves in the two subtrees, we get that the number of leaves in the search tree $\mathcal{T}$ is bounded by $r_0^{k-3} + r_0^{k-5}$. Since the polynomial $x^k - x^{k-3} - x^{k-5}$ and the polynomial $x^5 - x^2 - 1$ have the same positive root r_0, we get $r_0^k = r_0^{k-3} + r_0^{k-5}$, which proves that the number of leaves in the search tree $\mathcal{T}$ is bounded by r_0^k. This completes the inductive proof and the proof of the proposition. $\square$

By Proposition 3.11, what remains to show is that every root-leaf path in a search tree for the algorithm **VC3-solver** is compressible. We start with the following proposition.

Proposition 3.12 *Let $P = (\alpha_i, \alpha_{i+1}, \dots, \alpha_{i+l})$, $l > 0$, be a subpath of a root-leaf path in a search tree $\mathcal{T}$ for the algorithm* **VC3-solver**. *If α_{i+l} is the only node on the path P whose associated graph is 3-regular, then the path P is compressible.*

Proof. Let (G_{i-1}, k_{i-1}) be the instance associated with the parent node α_{i-1} of α_i in $\mathcal{T}$, where the graph G_{i-1} has n_{i-1} vertices and m_{i-1} edges (recall that the root of $\mathcal{T}$ also has a virtual parent associated with the input instance to the algorithm). Let G_{i+l} be the graph associated with the node α_{i+l} where G_{i+l} has n_{i+l} vertices and m_{i+l} edges. Since the graph G_{i+l} is 3-regular, we have $m_{i+l}/n_{i+l} = 3/2$. Let $m' = m_{i-1} - m_{i+l}$, $n' = n_{i-1} - n_{i+l}$. Since $m_{i-1}/n_{i-1} \le 3/2$ (the graph G_{i-1} has degree bounded by 3), we have $m'/n' \le 3/2$.

Let x_f be the number of folding operations on P, E_f the number of edges removed, V_f the number of vertices removed, S_f the surplus, and K_f the reduction of the parameter, in all folding operations on P. In a similar way, define x_1, E_1, V_1, S_1, K_1, for the 1-side of the $(1,3)$ branches; x_3, E_3, V_3, S_3, K_3, for the 3-side of the $(1,3)$ branches; x_2, E_2, V_2, S_2, K_2 for the 2-side of the $(2,5)$ branches; x_5, E_5, V_5, S_5, K_5, for the 5-side of the $(2,5)$ branches; and x_r, E_r, V_r, S_r, K_r, for the category-4 operations on P. Since $m'/n' \le 3/2$, we can write

$$\frac{E_f + E_1 + E_3 + E_2 + E_5 + E_r}{V_f + V_1 + V_3 + V_2 + V_5 + V_r} \le \frac{3}{2}. \tag{8}$$

Arranging (8), we get

$$3V_f - 2E_f \ge (2E_1 - 3V_1) + (2E_3 - 3V_3) + (2E_2 - 3V_2) + (2E_5 - 3V_5) + (2E_r - 3V_r). \tag{9}$$

From the definition of the amortized cost, and by the monotonicity of addition, we can define the amortized cost for each type of operations, λ ($\lambda = 1, 2, 3, 5, r, f$), by: $M_\lambda = 5E_\lambda - 6V_\lambda + 6S_\lambda - 3K_\lambda$. Since the total K_λ vertices included in the partial cover for any type of operations λ must cover all the E_λ edges removed by that type, and since each vertex can cover at most three edges, $K_\lambda \ge E_\lambda/3$. Hence, $2E_\lambda - 3V_\lambda \ge -3S_\lambda + M_\lambda/2$. Using this inequality and the parameters of the operations given in Figure 5, we get: $3V_f - 2E_f \le \frac{5}{2}x_f$, $2E_1 - 3V_1 \ge 3x_1$, $2E_3 - 3V_3 \ge 3x_3$, $2E_2 - 3V_2 \ge 3x_2$, $2E_5 - 3V_5 \ge \frac{1}{2}x_5$, $2E_r - 3V_r \ge -3S_r + M_r/2$. Substituting these bounds in Inequality (9) and arranging it we get:

$$x_f + S_r \ge x_2 + (x_1 + x_3) + x_5/6 + x_f/6 + M_r/6. \tag{10}$$

Since the graph G_{i+l} associated with the node α_{i+l} is 3-regular, it is not difficult to verify the following: Either we must have at least one folding operation along P, or at least one operation of those described in step 4 of **BindingSet2()**. This is true since these are the only operations that could make the graph become 3-regular. (The only way to create a 3-regular graph during the execution of the algorithm is either by a folding operation or by an operation in step 4 of **BindingSet2()**, which adds an edge to the resulting graph. All the other operations remove some vertices from the graph, which has degree bounded by 3, and hence cannot result in a 3-regular graph.) Since every category-4 operation has a non-negative amortized cost by Proposition 3.8, and since the amortized cost of the operation in step 4 of **BindingSet2()** was proved to be at least 6 in Lemma 3.4, it follows that if the operation in step 4 of **BindingSet2()** is performed, then we must have $M_r \ge 6$. Therefore, we either have $x_f \ge 1$, or $M_r \ge 6$. Since $x_f + S_r$ is an integer, from Inequality (10), we get

$$x_f + S_r \ge x_2 + (x_1 + x_3) + 1. \tag{11}$$

Note that if a node α specifies an operation corresponding to the 1-side or the 3-side of a $(1, 3)$ branch, then the graph associated with the parent of α must be 3-regular (see step 4 of the algorithm **VC3-solver**). Since α_{i+l} is the only node on the path P whose associated graph is 3-regular, and since α_{i+l} is the last node on the path, there is at most one node (i.e., node α_i) on the path P that may specify the 1-side or the 3-side operation of a $(1, 3)$ branch, and hence, $x_1 + x_3 \leq 1$. Combining this observation with Inequality (11), we get

$$x_f + S_r \geq x_2 + 2(x_1 + x_3). \tag{12}$$

Using the same notations given before Definition 3.10, we have $d(P) = x_f + S_r$, $x_2 = x_2(P)$, $x_1 = x_1(P)$, $x_3 = x_3(P)$, and $(x_f + S_r) - (x_2 + 2(x_1 + x_3))$ is the surplus of the path P. Thus, Inequality (12) gives that $\mathrm{Surp}(P) \geq 0$, and hence the path P is compressible. $\square$

Proposition 3.13 *Let G be a nice graph with n vertices and m edges. Then $m/n \geq 6/5$.*

Proof. The nice graph G contains no vertices of degree less than 2. Let n_2 and n_3 be the number of degree-2 and degree-3 vertices in G, respectively. Then $2m = 2n_2 + 3n_3 = 2n + n_3$. Since the nice graph G contains no adjacent degree-2 vertices, we have $3n_3 \geq 2n_2$. Combining these two relations we get the desired result. $\square$

Lemma 3.14 *Every root-leaf path in a search tree $\mathcal{T}$ of the algorithm* **VC3-solver** *is compressible.*

Proof. For an input (G, k) to the algorithm **VC3-solver**, if the graph G is 3-regular, then we subdivide an edge of G by two degree-2 vertices. Let the resulting graph be G'. Since the graph G can be obtained from G' by folding a degree-2 vertex in G', by Lemma 2.2, G has a vertex cover of size k if and only if G' has a vertex cover of size $k + 1$. Therefore, we can instead apply the algorithm to the instance $(G', k' = k + 1)$, where G' is not a 3-regular graph. Note that after subdividing an edge in G to obtain G', G' is connected, and $\tau(G') = \tau(G) + 1$. Since the parameter k in the instance (G, k) is assumed to be not larger than $\tau(G)$ by condition (1) in Assumption 2.6, the parameter k' is also not larger than $\tau(G')$. Therefore conditions (1) and (2) in Assumption 2.6 still hold on the graph G'. Moreover, since G' has two more vertices than G, and the parameter $k' = k + 1$, G' satisfies the assumption given by Proposition 2.1, namely that the number of vertices in G' is bounded by $2k'$. By doing this operation, the order of the running time of the algorithm is not affected. Thus, we can always assume that the graph associated with the root of the search tree $\mathcal{T}$ is not a 3-regular graph.

For any root-leaf path $P' = (\alpha'_1, \alpha'_2, \ldots, \alpha'_t)$ in the search tree $\mathcal{T}$, let α'_{i_1}, α'_{i_2}, $\ldots$, α'_{i_r} be the nodes on P' whose associated graphs are 3-regular. Note that it is impossible for two graphs associated with two consecutive nodes on P' to be both 3-regular — the only operation applicable to a 3-regular graph is step 4 in the algorithm **VC3-solver** that does not result in a 3-regular graph. Thus, each of the subpaths $(\alpha'_{i_{j-1}+1}, \ldots, \alpha'_{i_j})$ on P', $j = 1, \ldots, r$ (here we let α'_{i_0+1} be α'_1), satisfies the condition in Proposition 3.12 and is compressible, which equivalently means that the path has a non-negative surplus. Therefore, in order to prove the lemma, it suffices to show that the subpath $P = (\alpha'_{i_r+1}, \ldots, \alpha'_t)$ has a non-negative surplus, and hence is compressible. To simplify the notations, we rename the nodes on P and let $P = (\alpha_1, \alpha_2, \ldots, \alpha_s)$ (if the root-leaf path P' contains no node associated with a 3-regular graph, we let $P = P'$).

Let (G_0, k_0) be the instance associated with the parent of α_1 (note that the root of $\mathcal{T}$ also has a virtual parent whose associated instance is the original input to the algorithm), where the graph G_0 has n_0 vertices and m_0 edges. Since the degree of G_0 is bounded by 3, we have

$$m_0/n_0 \leq 3/2. \tag{13}$$

If α_1 is the root of $\mathcal{T}$, then (G_0, k_0) is the original input instance to the algorithm. In this case, by Proposition 2.1, we have $k_0 \geq n_0/2$. On the other hand, If α_1 is not the root of $\mathcal{T}$, then the graph G_0 associated with the parent node of α_1 is 3-regular, and $2m_0 = 3n_0$. Since each vertex can cover at most 3 edges, we must have $3k_0 \geq m_0$ (otherwise the answer to the instance (G_0, k_0) is negative). This also gives us $k_0 \geq n_0/2$. Therefore, in all cases, we have

$$k_0 \geq n_0/2. \tag{14}$$

Case 1. All the nodes on the path P are non-branching nodes.

Suppose that the parameter reduction and the surplus of the operation specified by the first node α_1 on P are k_1 and s_1, respectively. By the definition of the surplus, we have $s_1 \geq k_1 - 5$. The instance associated with α_1 is $(G_1, k_0 - k_1)$ for some graph G_1. By condition (1) in Assumption 2.6, the original parameter k is not larger than the size of a minimum vertex cover of G. Therefore the parameter value $k_0 - k_1$ associated with α_1 is not larger than the size of a minimum vertex cover of G_1, otherwise the original parameter k would be larger than the size of a minimum vertex cover of G. It follows that when the algorithm terminates at node α_s along the path P in the search tree, either the computed cover is a minimum vertex cover, and hence, the reduction in the parameter along the path $\alpha_2, \ldots, \alpha_s$ is exactly equal to $k_0 - k_1$; or the size of the resulting cover for G_1 has exceeded the parameter $k_0 - k_1$, and hence, the reduction in the parameter along the path $\alpha_2, \ldots, \alpha_s$ is greater than $k_0 - k_1$. Note that all the nodes on P, except α_1, specify non-branching operations whose parameter reduction and surplus are equal. Therefore, the reduction in the parameter, or equivalently the sum of the surplus, of the nodes $\alpha_2, \ldots, \alpha_s$ is at least $k_0 - k_1$. Adding the surplus of the node α_1, we get

$$\text{Surp}(P) \geq s_1 + (k_0 - k_1) \geq (k_1 - 5) + (k_0 - k_1) = k_0 - 5. \tag{15}$$

Observe that we have $k_0 \geq 25$. In fact, if $k_0 \leq 24$, then by Inequality (14), $n_0 \leq 2k_0 < 50$. In such case step B of **Reducing** would be applicable to (G_0, k_0), and the parent node of α_1 would not be a branch node. Combining the fact that $k_0 \geq 25$ with Inequality (15), we conclude that in this case $\text{Surp}(P) \geq 20$.

Case 2. The path P contains branch nodes. Let α_h be the last branch node on P.

Consider the two subpaths $P_1 = (\alpha_1, \ldots, \alpha_h)$ and $P_2 = (\alpha_{h+1}, \ldots, \alpha_s)$ of P. Let (G_h, k_h) be the instance associated with the node α_h. Since all nodes in the subpath P_2 are non-branching nodes, as shown in Case 1 (see Inequality (15)), we have

$$\text{Surp}(P_2) \geq k_h - 5. \tag{16}$$

Now consider the value $\text{Surp}(P_1)$. Let x_f, E_f, V_f, K_f, S_f, x_1, E_1, V_1, K_1, S_1, x_3, E_3, V_3, K_3, S_3, x_2, E_2, V_2, K_2, S_2, x_5, E_5, V_5, K_5, S_5, x_r, E_r, V_r, K_r, S_r, denote the same entities as in Proposition 3.12 along the subpath $P_1 = (\alpha_1, \ldots, \alpha_h)$. We have

$$x_f + x_1 + 2x_2 + 3x_3 + 5x_5 + K_r + k_h = k_0. \tag{17}$$

This is because the operation specified by the first node α_1 on the subpath P_1 is applied to the instance (G_0, k_0) (which is associated with the parent of α_1), and the last node α_h on P_1 is associated with the instance (G_h, k_h), and $x_f + x_1 + 2x_2 + 3x_3 + 5x_5 + K_r$ is the total parameter reduction of the operations on P_1.

According to our algorithm, the graph G_h associated with the branch node α_h in the search tree $\mathcal{T}$ is nice. Thus, if we let n_h and m_h be the number of vertices and edges in G_h, respectively, then by

Proposition 3.13, $m_h/n_h \geq 6/5$. Using our notations, we have $m_h = m_0 - E_f - E_1 - E_3 - E_2 - E_5 - E_r$ and $n_h = n_0 - V_f - V_1 - V_3 - V_2 - V_5 - V_r$. Therefore,

$$\frac{m_0 - E_f - E_1 - E_3 - E_2 - E_5 - E_r}{n_0 - V_f - V_1 - V_3 - V_2 - V_5 - V_r} \geq \frac{6}{5}. \tag{18}$$

In a similar way to that in Proposition 3.12, define the amortized cost for each type of operations λ ($\lambda = 1, 2, 3, 5, r, f$), by: $M_\lambda = 5E_\lambda - 6V_\lambda + 6S_\lambda - 3K_\lambda$. Hence, we have $5E_\lambda - 6V_\lambda = M_\lambda + 3K_\lambda - 6S_\lambda$. Using this equality and the parameters of the operations given in Figure 5, we get: $6V_f - 5E_f \leq 2x_f$, $5E_1 - 6V_1 \geq 9x_1$, $5E_3 - 6V_3 \geq 15x_3$, $5E_2 - 6V_2 \geq 12x_2$, $5E_5 - 6V_5 \geq 16x_5$, $5E_r - 6V_r \geq M_r + 3K_r - 6S_r$. Combining these inequalities with Inequalities (13), (14), (17), (18), and arranging the terms we get:

$$5x_f \geq 6x_1 + 6x_2 + 6x_3 + x_5 + M_r - 6S_r - 3k_h. \tag{19}$$

Hence:

$$x_f + S_r \geq x_2 + (x_1 + x_3) + x_5/6 + M_r/6 + x_f/6 - k_h/2 \geq x_2 + (x_1 + x_3) - k_h/2. \tag{20}$$

Here we have used Proposition 3.8 which gives that $M_r \geq 0$. Since $d(P_1) = x_f + S_r$, $x_1 = x_1(P_1)$, $x_2 = x_2(P_1)$, and $x_3 = x_3(P_1)$ (see Definition 3.10), From (20), we get

$$\mathrm{Surp}(P_1) = (x_f + S_r) - (x_2 + 2(x_1 + x_3)) \geq -(x_1 + x_3) - k_h/2. \tag{21}$$

Combining this inequality with Inequality (16),

$$\mathrm{Surp}(P) = \mathrm{Surp}(P_1) + \mathrm{Surp}(P_2) \geq k_h/2 - (x_1 + x_3) - 5. \tag{22}$$

As explained in Case 1, since the node α_h is a branch node, the graph G_h associated with α_h must be nice and have at least 50 vertices. Since G_h is nice (and hence is clean), the number of edges in G_h is more than 50. Since each vertex can cover at most 3 edges, we have $k_h \geq 17$ (otherwise the answer to the instance (G_h, k_h) is negative). Moreover, no graph associated with a node on the path P_1 is 3-regular. Since a 1-side or a 3-side operation of a $(1,3)$ branch can only be applied on a 3-regular graph, there is at most one node on P_1 (the node α_1) that may specify such an operation. Therefore, $x_1 + x_3 \leq 1$. Combining the last two inequalities with Inequality (22), we get $\mathrm{Surp}(P) \geq 0$ and the path P is compressible. This completes the proof of the lemma. $\square$

Theorem 3.15 *The algorithm* **VC3-solver** *runs in time* $O(1.194^k k^2 + n)$.

PROOF. First observe that by spending $O(n)$ time pre-processing the input instance, we can remove vertices of degree 0 and 1. After that, it must be true that every component in the graph is a non-tree component, and hence, at least one third of the number of vertices in each component must be included in any vertex cover of the component. This means that the resulting parameter k satisfies $k \geq n/3$, where n is the number of vertices in the resulting graph (otherwise the answer to the instance is negative). Then the algorithm mentioned in Proposition 2.1 is applied. This algorithm runs in $O(k\sqrt{k})$ time. Finally the algorithm **VC3-solver** is invoked. Let $\mathcal{T}$ be the search tree for the algorithm **VC3-solver** on the input instance. By Lemma 3.14, every root-leaf path in $\mathcal{T}$ is compressible. Since every branching operation in $\mathcal{T}$ can be classified as a $(1,3)$, $(2,5)$, or (a,b), with (a,b) not worse than a $(3,5)$-branch, from Proposition 3.11 we get that the number of leaves in $\mathcal{T}$ is $O(r^k)$, where $r \leq 1.194$ is the positive root of the polynomial $x^5 - x^2 - 1$. It follows that the number of root-leaf paths in $\mathcal{T}$ is also $O(1.194^k)$. Since every non-root node on a given path corresponds to a reduction in the parameter value, and since the parameter to the input instance has value k, it follows that any root-leaf path in $\mathcal{T}$ contains at most $k+1$ nodes. Therefore, the total number of nodes in $\mathcal{T}$ is $O(1.194^k k)$. At every node in the search tree $\mathcal{T}$ the time spent by the algorithm is linear in the size of the graph, which is $O(k)$. To verify that, let us look at the operations performed by the algorithm. First, whenever **Clean** is invoked, the time spent is proportional to the size of the subgraph removed, and hence is $O(k)$. Also the time taken by a branching operation is $O(k)$. We explained in Section 2 how step 2 of **VC3-solver** can be carried out in time $O(k)$. We also explained in Section 2 how each step in **Reducing** can be performed in time $O(k)$. This shows that the time spent at every node in the search tree is $O(k)$. The running time of the algorithm is then $O(n + k\sqrt{k} + 1.194^k k^2) = O(1.194^k k^2 + n)$, where $O(k\sqrt{k} + n)$ is the pre-processing time. It follows that the running time of the algorithm **VC3-solver** is $O(1.194^k k^2 + n)$. □

4 An algorithm for IS-3

In this section we show how the algorithm for VC-3 implies an algorithm for IS-3. The approach is exactly the same as that employed in [6], which used a less efficient algorithm for VC-3 than the one given in this paper, to derive an algorithm for IS-3 running in time $O(1.174^n)$. The algorithm for IS-3 presented here runs in time $O(1.1255^n)$, and slightly beats the previously most efficient $O(1.1259^n)$-time algorithm by Beigel [3].

Lemma 4.1 (Lemma 6.1, [6]) *Let G be a connected graph of n vertices and degree bounded by 3. Then a minimum vertex cover of G contains at most $(2n+1)/3$ vertices.*

Theorem 4.2 *The* IS-3 *problem can be solved in time $O(1.1255^n)$.*

Proof. Let G be a graph of degree bounded by 3. The graph G may not necessarily be connected. Let $C_1, \ldots, C_k$ be the connected components of G of sizes $n_1, \ldots, n_k$, respectively. It is clear that a maximum independent set of G is the union of maximum independent sets of the components $C_1, \cdots, C_k$. For each component C_i of G, instead of finding a maximum independent set for C_i, we try to construct a vertex cover of k_i vertices, for $k_i = 1, 2, \ldots$. At the first k_i for which we are able to construct a vertex cover of k_i vertices for C_i, we know this vertex cover is a minimum vertex cover. Thus, the complement of this vertex cover is a maximum independent set for C_i. By Lemma 4.1, we must have $k_i \le (2n_i+1)/3$. Thus, by Theorem 3.15, a maximum independent set for the component C_i can be constructed in time $O(1.194^{(2n_i+1)/3}(2n_i+1)^2/3 + n_i)$, which is $O(1.1255^{n_i})$. In conclusion, a maximum independent set in the graph G can be constructed in time $O(1.1255^{n_1} + \cdots + 1.1255^{n_k})$. By an argument similar to that given in the proof of Assumption 2.6 at the end of Section 2, it follows that $O(1.1255^{n_1} + \cdots + 1.1255^{n_k}) = O(1.1255^n)$. □

5 Conclusion

In this paper we presented algorithms for the parameterized Vertex Cover and the Maximum Independent Set problems on degree-3 graphs. Our algorithm for VC-3 runs in time $O(1.194^k k^2 + n)$ and improves Chen et al.'s $O(1.237^k + kn)$ time algorithm [7]. Our algorithm for IS-3 runs in time $O(1.1255^n)$ and improves Beigel's $O(1.1259^n)$ time algorithm [3].

We emphasize that the importance of our results lies in the techniques that we use to analyze the size of the search tree. Despite the fact that the analysis of the algorithm is lengthy, the algorithm itself is very simple and uniform. The algorithm distinguishes few cases to eliminate cut-vertices and bridges from the graph. However, all these cases are solved easily and without any branching. As a matter of fact, these cases use very simple and elegant graph-theoretic operations that can be generalized in a straightforward manner to the Vertex Cover problem on general graphs. If one looks carefully at the algorithm itself, the algorithm is very intuitive. Basically the overall behavior of the algorithm can be described as follows. As long as the case can be solved without any branching, solve it (folding, reducing, and cleaning). If none of the above applies, then either we can do an efficient and uniform branch (alternating tree), which is a single branch that does not distinguish any cases, or we branch arbitrarily at any vertex, and the amortized analysis shows that this operation will be balanced by non-branching operations. The analysis of the algorithm might be lengthy, but the techniques involved are elementary combinatorial techniques.

Finally, we indicate that our approach opens a new direction in the analysis of the running time of exact algorithms for NP-hard problems that use the search tree method. Instead of looking at sophisticated algorithms and deriving an easy but conservative upper bound on the size of the search tree, we can consider instead very simple and intuitive algorithms, and perform an amortized analysis that reflects more closely the actual size of the search tree. We believe that this method of analysis is applicable to a variety of NP-hard problems.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, MA, (1974).

[2] R. Balasubramanian, M. R. Fellows, and V. Raman, An improved fixed parameter algorithm for vertex cover, *Information Processing Letters* **65**, (1998), pp. 163-168.

[3] R. Beigel, Finding maximum independent sets in sparse and general graphs, in *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), (1999), pp. 856-857.

[4] J. F. Buss and J. Goldsmith, Nondeterminism within P, *SIAM Journal on Computing* **22**, (1993), pp. 560-572.

[5] L. Cai and D. Juedes, On the existence of subexponential-time parameterized algorithms, *Journal of Computer and System Sciences* **67-4**, (2003), pp. 789-807.

[6] J. Chen, I. A. Kanj, and W. Jia, Vertex cover: further observations and further improvements, *Journal of Algorithms* **41**, (2001), pp. 280-301.

[7] J. Chen, L. Liu, and W. Jia, Improvement on Vertex Cover for low-degree graphs, *Networks* **35**, (2000), pp. 253-259.

[8] *DIMACS Workshop on Faster Exact Algorithms for NP-hard problems*, Princeton, NJ, (2000).

[9] R. Downey and M. Fellows, Parameterized computational feasibility, in *Feasible Mathematics II*, P. Clote and J. Remmel, eds., Boston, Birkhauser (1995), pp. 219-244.

[10] R. Downey and M. Fellows, *Parameterized Complexity*, New York, Springer, (1999).

[11] P. Hansen and B. Jaumard, Algorithms for the maximum satisfiability problem, *Computing* **44**, (1990) pp. 279-303.

[12] R. Impagliazzo, R. Paturi, and F. Zane, Which problems have strongly exponential complexity?, *Journal of Computer and System Sciences* **63-4**, (2001), pp. 512-530.

[13] T. Jian, An $O(2^{0.304n})$ algorithm for solving the maximum independent set problem, *IEEE Transactions on Computers* **35**, (1986) pp. 847-851.

[14] D. Johnson and M. Szegedy, What are the least tractable instances of max. independent set?, in *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), (1999), pp. 927-928.

[15] D. S. Johnson and M. A. Tricks, Eds., *"Cliques, Coloring and Satisfiability, Second DIMACS Implementation Challenges"*, DIMACS Series on Discrete Mathematics and Theoretical Computer Science **26**, American Mathematical Society, Providence, RI, (1996).

[16] I. A. Kanj, Vertex Cover: Exact and Approximate Algorithms and Applications, *Ph.D. Dissertation*, Dept. Computer Science, Texas A&M University, College Station, Texas, (2001).

[17] G. L. Nemhauser and L. E. Trotter, Vertex packing: structural properties and algorithms, *Mathematical Programming* **8**, (1975), pp. 232-248.

[18] R. Niedermeier and P. Rossmanith, Upper bounds for vertex cover further improved, *Lecture Notes in Computer Science* **1563**, (1999), pp. 561-570.

[19] J. M. Robson, Algorithms for maximum independent set, *Journal of Algorithms* **6**, (1977), pp. 425-440.

[20] M. Shindo and E. Tomita, A simple algorithm for finding a maximum clique and its worst-case time complexity, *Sys. and Comp. in Japan* **21**, (1990), pp. 1-13.

[21] U. Stege and M. Fellows, An improved fixed-parameter-tractable algorithm for vertex cover, *Technical Report* **318**, Department of Computer Science, ETH Zurich, April 1999.

[22] J. E. Hopcroft and R. E. Tarjan, Dividing a graph into triconnected components, *SIAM Journal on Computing* **2**, (1973), pp. 135-158.

[23] R. E. Tarjan and A. E. Trojanowski, Finding a maximum independent set, *SIAM Journal on Computing* **7**, (1986), pp. 537-546.

[24] W. T. Tutte, *Graph Theory*, Addison-Wesley Publishing Company, Menlo Park, California, (1984).