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 recently 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 $O(1.194^k + n)$ for the parameterized Vertex Cover problem on degree-3 graphs, and a simple algorithm of running time $O(1.1254^n)$ 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 [7, 14]. 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 [10], while theoretically, this line of research may lead to a deeper understanding of the structure of NP-hard problems [4, 9, 11, 13].

Two of the most extensively studied problems in this line of research are the Maximum Independent Set and the Vertex Cover problems. For Maximum Independent Set (given a graph G, find a maximum independent set in G), since the initiation by Tarjan and Trojanowski [22] with an $O(1.259^n)$ time algorithm, there have been continuously improved algorithms for the problem [2, 12, 19, 20]. 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 [2] 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 the Maximum Independent Set problem 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 [9]. This is also due to its applications in fields like computational biochemistry [15]. Since the development of the first parameterized algorithm by Buss (see [3]), which has running time $O(kn + 2^k k^{2k+2})$, there has been an impressive list of improved algorithms for the problem

[*]Supported in part by NSF under the grant CCR-0000206. Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. Email: {gexia,chen}@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.

[1, 5, 6, 8, 17, 21]. Currently, the best parameterized algorithm for Vertex Cover has running time $O(kn + 1.285^k)$ for general graphs [5], and the best parameterized algorithm for Vertex Cover on degree-3 graphs has running time $O(kn + 1.237^k)$ [6].

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 corresponds to an instance of the problem. At a node α in the tree the search process considers a local structure in the problem instance corresponding to 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 corresponds to 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 [1, 5, 19, 22].

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. Obviously, 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 (of few lines) 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 $O(n + 1.194^k)$, improving the previous best algorithm of running time $O(kn + 1.237^k)$ [6], and our algorithm for IS-3 runs in time $O(1.1254^n)$, improving the previous best algorithm of running time $O(1.1259^n)$ [2].

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 [1, 3, 5, 8, 12, 17, 19, 20, 21, 22]. In particular, Vertex Cover and Maximum Independent Set on graphs of degrees 3 and 4 have received a lot of attention recently [2, 5, 6]. 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 a graph with low-degree [5, 17, 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 [4, 11, 13]. More specifically, combining the results in [11], [13], and [4], 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 $O((1+\epsilon)^k p(n))$ (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, 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 $O((1+c_2)^k p(n))$, 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 A simple algorithm for VC-3 and the reduction rules

Let $G = (V, E)$ be an undirected graph. For a subset V' of vertices in G, denote by $G(V')$ the subgraph of G induced by V'. 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. Note that if every vertex in G has degree bounded by 3, and hence, can cover at most 3 edges, it is true that for any induced connected non-tree subgraph H of G with n_H vertices, we have $\tau(H) \geq n_H/3$. Let (G, k) be an instance of the VC-3 problem. This version of the following proposition appears in [6], and is based on a theorem by Nemhauser and Trotter [16].

Proposition 2.1 (Proposition 2.1, [6]) *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), where the graph G_1 contains at most $2k_1$ vertices with $k_1 \leq k$, and such that the graph G has a vertex cover of at most k vertices if and only if the graph G_1 has a vertex cover of at most k_1 vertices.*

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 the vertices 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 whose proof is easy and can be found in [5].

Lemma 2.2 ([5]) *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 $\tau(G) = \tau(G') + 1$. Moreover, a minimum vertex cover for G can be constructed from a minimum vertex cover for G' in constant time.*

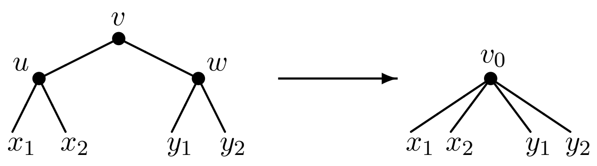

Figure 1: Vertex folding

We introduce some terminologies. A vertex folding on a degree-2 vertex v is *safe* if folding v does not create vertices of degree larger than 3. A cycle of length l in a graph is an *alternating cycle* if it contains exactly $\lfloor l/2 \rfloor$ degree-2 vertices of which no two are adjacent. Finally, a subgraph T in G is an *alternating tree* if: (1) T is induced by a subset of vertices in G; (2) T is a tree; (3) all leaves of T are of degree 3 in G; and (4) 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 algorithm takes as input a graph G and a parameter k and works by growing a partial minimum vertex cover C for G based on a branch-and-search process. If any search path finds a vertex cover of at most k vertices, the algorithm reports a success. Otherwise, the algorithm reports a failure. By *branching on a vertex set S* in which no two vertices are adjacent, we mean branching by either including all vertices in S in the partial cover C, or including all vertices that are not in S but are adjacent to some vertices in S, then recursively working on the remaining graph.

The algorithm is given in Figure 2. The algorithm uses two subroutines **Fold**(v) and **Reducing**. 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 C), and updates the graph G, the partial cover C, and the parameter k accordingly. In particular, we will assume without loss of generality that at beginning of each step, the graph contains no vertices of degree less than 2.

```
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 there exists a degree-2 vertex $v$ such that folding $v$ is safe do Fold($v$);
2.1  if Reducing is applicable then apply Reducing and go to step 1 in VC3-solver;
2.2  else if there is a degree-2 vertex $v$ then branch on the two neighbors of $v$;
2.3  else branch on a degree-3 vertex $v$.

Reducing
0.   if there is a component $H$ of size bounded by 50
     then compute a minimum vertex cover of $H$ by brute force;
1.   else if there are two adjacent triangles $(u, v, w)$ and $(u, v, z)$ then include $v$ in the cover;
2.   else if there is an alternating cycle $K$ in $G$ then include all degree-3 vertices on $K$ in the cover;
3.   else if removing a cut-vertex or a two-edge cut results in a component $H$ with $2 \le |V(H)| \le 50$
     then remove $H$ without any branching as explained in Theorem 2.3;
4.   else 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 $G$.
```

Figure 2: The algorithm VC3-solver

Theorem 2.3 *The algorithm* **VC3-solver** *solves the* VC-3 *problem correctly.*

Proof. To prove the correctness of the algorithm, we show that at least one path in the search tree of the algorithm always has the updated partial cover C entirely contained in a minimum vertex cover of the input graph G. Thus, for a step without branching, we must ensure that the vertices to be included in the partial cover C by this step are entirely contained in a minimum vertex cover of the current graph, while for a branching step, we must show that 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 C.

By Lemma 2.2, folding a degree-2 vertex is always valid. Also, it is easy to see that including the neighbor of a degree-1 vertex into the partial cover C is always safe. Branching on a single vertex v is valid because a minimum vertex cover either contains v or contains all its neighbors. Moreover, it has been proved in [5] that for any degree-2 vertex v, there is a minimum vertex cover that either contains both neighbors of v or contains none of them. Thus, all steps in the main algorithm **VC3-solver**, except the call to the subroutine **Reducing**, are correct. Now consider the subroutine **Reducing**. The steps in **Reducing** can be justified as follows. Step 0 should be clear. Step 1 in **Reducing**: since any minimum vertex cover must contain at least two vertices in any triangle, one of the vertices u and v in the two adjacent triangles (u, v, w) and (u, v, z) must be included in a minimum vertex cover. By symmetry, we can simply include u. Step 2 in **Reducing**: since at least $\lceil l/2 \rceil$ vertices on K are needed to cover the l edges of the cycle K, picking the $\lceil l/2 \rceil$ vertices of degree 3 on K is a safe choice because they not only cover all edges on the cycle K, but also cover all edges incident on K. Step 4 in **Reducing**: we show that there is a minimum vertex cover that either contains all vertices in T that are of degree 3 in G or contains none of them. Pick any vertex v in T of degree 3 in G, and let N_i be the set of vertices in T such that, for every vertex u in N_i, the unique path from v to u along the edges of 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. Suppose that v is in a minimum vertex cover, then 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 T that are of degree 3 in G in the minimum vertex cover. Since v is an arbitrary degree-3 vertex, we have shown that there is a minimum vertex cover that either contains all vertices in T that are of degree 3 in G, or contains none of them. Therefore, branching on the vertices in T that are of degree 3 in G is correct.

Now only step 3 in **Reducing** still needs explanation and justification. Suppose that there is a cut in G, which is either a cut-vertex or a two-edge cut, whose removal results in at least one component H satisfying $2 \leq |V(H)| \leq 50$. Step 3 of **Reducing** removes H as follows.

If the cut is a cut-vertex u, let H_+ be the subgraph induced by H and u. We examine in constant time all minimum vertex covers of H_+. If any minimum vertex cover C_+ of H_+ contains u, we simply include C_+ in the partial cover C. If no minimum vertex cover of H_+ contains u, then if u has exactly one neighbor u' not in H, we include an arbitrary minimum vertex cover of H_+ and u' into the partial cover C; otherwise, u has two neighbors not in H. In this case we remove H and include an arbitrary minimum vertex cover of H_+ into the partial cover (note that u remains in the resulting graph).

If the cut consists of two edges (u, u') and (v, v'), we can assume without loss of generality that u and v are both in H, and that $u \neq v$ (otherwise the case is reduced to the case of a cut-vertex). Suppose first that $u' \neq v'$. We distinguish the following cases.

(1) If there is a minimum vertex cover C_H for H that contains both u and v, then we include C_H in the partial cover C.
(2) If there is a minimum vertex cover C_H for H that includes u (resp. v) and every minimum vertex cover of H excludes v (resp. u), then we include C_H and v' (resp. u').
(3) If there are two minimum vertex covers C_u, C_v of H such that $u \in C_u$, and $v \in C_v$, then remove H, add an edge (u', v') in case it does not already exist, and reduce the parameter k by $\tau(H)$. When the algorithm returns a minimum vertex cover C' of the new graph, a minimum vertex cover of the original graph can be computed as follows. If the minimum vertex cover C' of the new graph contains u', then $C' \cup C_v$ is a minimum vertex cover for the original graph; otherwise, $C' \cup C_u$ is.
(4) If every minimum vertex cover for H excludes both u and v, then we consider two possibilities: (4.1) if the minimum size of a vertex cover for H containing both u and v is $\tau(H) + 2$, then we include a minimum vertex cover for H and u' and v'; (4.2) if the minimum size of a vertex cover for H containing both u and v is $\tau(H) + 1$, then we remove H, add a new vertex w', connect w' to u' and v', and reduce the parameter by $\tau(H)$. If the minimum vertex cover C' for the new graph excludes both u' and v', then C' plus a vertex cover of H including both u and v of minimum size, is a minimum vertex cover of the original graph; otherwise, C' includes both u' and v', and C' plus a minimum vertex cover of H is a minimum vertex cover of the whole graph (note that, as mentioned before, we can always assume that a minimum vertex cover of the new graph includes both u' and v' or excludes both of them).

Suppose now that $u' = v' = w'$. If case (1) above applies, then we do the same as in case (1). Otherwise, we include an arbitrary minimum vertex cover of H and w' in the partial cover.

It is not difficult to verify the correctness of the algorithm in handling the above cases provided that the cases are considered in the listed order. Also, it is easy to see that the cases can be detected and implemented in constant time. We leave the details to the interested reader. □

Remark 2.4 *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. By Proposition 2.1, $k \geq n/2$ where $n = |V(G)|$. We start calling the algorithm on G with $k' = n/2, n/2+1, \ldots, k$. 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. Clearly each call to the algorithm satisfies condition (1). We will show later that the size of the search tree of the algorithm on the instance (G, k) is $O(1.194^k)$. Thus, the size of the search tree in this case is $O(1.194^{k'} + 1.194^{k'+1} + \ldots + 1.194^k) = O(1.194^k)$. Hence, the upper bound on the size of the search tree with the new modification to the algorithm is unchanged. Now to justify (2), suppose that there are $G_1, \ldots, G_r$ components in G with $|V(G_i)| = n_i$. 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 call the algorithm on G_1, with $k_1 = n_1/2, n_1/2+1, \ldots, k$. If the algorithm fails to return a vertex cover in each of these cases, then clearly no vertex cover of size bounded by k exists. Otherwise, the algorithm returns a minimum vertex cover of G_1 of size $k_1 \leq k$. Now we call the algorithm on G_2 with $k_2 = n_2/2, n_2/2+1, \ldots, k - k_1$, and so on. It is now true that on each call to the algorithm on a graph component, conditions (1) and (2) hold true. It is not difficult to verify that the size of the search tree is still bounded by $O(1.194^k)$. We leave the details to the interested reader.

3 Analysis of the algorithm

We analyze the time complexity of the algorithm **VC3-solver** in this section. Denote by $L(k)$ the number of leaves in the search tree of our algorithm looking for a vertex cover of size bounded by k. Let α be a node in the search tree with a corresponding parameter k' (i.e., the resulting parameter at α is k'). If we branch at α by reducing the parameter k' by k'_1, k'_2, $\ldots$, k'_s, in each branch respectively, then such a branch will be called a $(k'_1, k'_2, \ldots, k'_s)$-branch.

Previously, the size of the branching tree (number of leaves) was analyzed by considering the worst-case recurrence relation over all recurrence relations corresponding to the branching cases of the algorithm, and computing the size of the search tree corresponding to this recurrence relation. One can easily see that such analysis is very conservative since we do not always branch with the worst-case recurrence, and hence, the size of the search tree will be much smaller than the size of the search tree obtained in such a conservative analysis.

We present next a novel way of 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 size of the search tree corresponding to the running time of the algorithm on input (G, k) is not larger than the size of a search tree corresponding to (G, k) with all its branches satisfying the recurrence relation $L(k) \leq L(k-3) + L(k-5)$. This will allow us to conclude that the size of the search tree is $O(r^k)$, where $r \leq 1.194$ is 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 1 of the algorithm **VC3-solver**.

2. $(1, 3)$ **branching** operations: the operations performed in step 2.3 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 2.2 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. 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 **Reducing** and those performed by **Clean**.

Let i be an operation[1] 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 in operation i, and k_i the reduction in the parameter after 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 an operation i. For instance, if i is the 6-side in a

[1]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).

$(3,6)$-branch, then $s_i = 6-5 = 1$, whereas if i is the 2-side in a $(2,5)$-branch, then $s_i = 2-3 = -1$. We define the amortized cost m_i of operation i by $m_i = 5e_i - 6v_i + 6s_i - 3k_i$. Note that if an 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 operation is $m_i = 5e_i - 6v_i + 3k_i$.

The amortized cost defined above will serve as a measure to how good an operation is relative to a $(3,5)$-branch. Even though an operation may not be as good as a $(3,5)$-branch, the operation may induce some changes to the graph that are measured by the amortized cost. These changes (like removing many edges relative to the number of vertices removed) will allow us to conclude that if such an operation is performed, there will be some gain somewhere along the way to compensate for this operation (like creating degree-1 vertices, or two adjacent degree-2 vertices). For instance, we will show that the amortized cost of the 2-side of a $(2,5)$-branch is non-negative. This will allow us eventually to conclude that, whenever we have an operation that is a 2-side of a $(2,5)$-branch, we must have some efficient operation (like a safe folding operation) that can be used to compensate for the 2-side of the branch, to make the branch at least as efficient as a $(3,5)$-branch. We start with the following lemma.

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 subgraph C' of C_0 that is removed by **Clean** must have at least one edge connecting it to C_0, 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. It is very easy to see that 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). 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 $k_0 \geq e_0/3$, give $m_0 = 5e_0 - 6v_0 + 3k_0 \geq -6$. □

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 leaves is a chain (i.e., a path between the two leaves).*

Lemma 3.4 *On a nice graph G, an operation i performed in step 3 of* **Reducing** *followed by an invocation to* **Clean***, is not worse than a $(3,5)$-branch and its amortized cost m_i is non-negative.*

PROOF. In step 3 of **Reducing**, the algorithm removes a cut-vertex or two cut-edges. In both cases, by our assumption, this results in a component H satisfying $2 \leq |V(H)| \leq 50$.

Since the operation in this case does not involve any branching, it is not worse than a $(3,5)$-branch. We only need to verify that the amortized cost 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 case (3) in Theorem 2.3 when one edge is added, and case (4.2), when one vertex and two edges are added. We show the statement for case (3). The proof that this statement holds true for case (4.2) is very similar. Following the same notation in Theorem 2.3, let (u, u'), and (v, v') be the two edges joining H to the remaining graph with u, v being in H. The operation in case (3) removes H 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 leaves namely u and v (note that a tree with at least two vertices must have at least two leaves), and by Fact 3.3, H must be a chain. This would imply that there were two adjacent degree-2 vertices in the graph prior to this operation (note that H can consist of only u and v) contradicting the fact that no 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 in H, (u, u'), (v, v') are removed and (u', v') is added), v_H vertices, and reduces the parameter by k_H. By a similar argument to that made in Lemma 3.1, we must have $k_H \geq e_H/3$. Since the operation is a non-branching operation, its amortized cost $m_i = 5(e_H + 1) - 6v_H + 3k_H \geq 6e_H - 6v_H + 5 \geq 5$. Also, since prior to this operation the graph was clean, it is easy to see that 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 $|V(\mathcal{S})| \leq 4l - 5$.*

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

Since the graph is nice, all leaves of T must be joined to $G - T$. Let E_1 be the set of edges that join the leaves of T to $G - T$, and let E_2 be the set of other edges joining T to $G - T$. Let $l_1 = |E_1|$ and $l_2 = |E_2| = l - l_1$. We remove the edges in E_2 and perform the following operation on the internal vertices of T: For any two degree-2 adjacent internal vertices merge them into one vertex. Let T' be the resulting tree. Since G was nice before removing the edges in E_2, it is easy to verify that $|V(T')| \geq |V(T)| - 2l_2$. Now T' satisfies the following conditions: (1) T' has at most l_1 leaves; (2) no two internal degree-2 vertices in T' are adjacent; and (3) no internal vertex in T' is connected to $G - T'$. A simple inductive argument shows that $|V(T')| \leq 4l_1 - 5$. It follows that $|V(T)| \leq |V(T')| + 2l_2 \leq 4l_1 - 5 + 2l_2 \leq 4l - 5$. This completes the proof. □

Lemma 3.6 *On a nice graph G, an operation i performed in step 4 of* **Reducing** *followed by an invocation to* **Clean***, is not worse than a $(3,5)$-branch, and its amortized cost m_i is non-negative.*

PROOF. Let T be a maximal alternating tree with $|V(T)| \geq 4$. 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 $|Y| = y$. Since T is an alternating tree containing $x \geq 1$ degree-3 vertices, it can be easily verified that: (1) $|D_2| = x - 1$ and hence $|V(T)| = 2x - 1$; and (2) there are exactly $(x + 2)$ edges between T and Y. Since the number of vertices in an alternating tree is $2x - 1$ which is an odd number, we have $|V(T)| \geq 5$, and hence, $x \geq 3$. Part (2), together with 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 0 in **Reducing**, there must exist a cut-vertex in Y that separates H from the rest of the graph. Since $2 \leq |H| \leq 50$, this is again not possible at this point of the algorithm by step 3 of **Reducing**. It follows that $y \geq 3$, and branching in step 4 of **Reducing** 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 ≥ 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 exactly those in T whose number is $v_i = 2x-1$. The edges removed are those in T plus the edges between T and Y. It is easy to see that 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 exactly $3x$. Moreover, the reduction k_i in the parameter is exactly 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 $x + 2 = 7$ 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 23, and hence, the total number of vertices in the graph component induced by $V(T) \cup V(\mathcal{S})$ is bounded by 32. This is not possible at this point of the algorithm due to the fact that step 0 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 24 vertices, and is connected to C_0 by exactly two edges. This is again not possible by step 3 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 α be the number of edges between the vertices of Y, and z that 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 $3x + z + \alpha$, 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

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

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

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

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

$$m_i \geq 5x + 3\alpha + 4z - 6t_i - 29 \tag{4}$$
$$\geq 5x + z + 3\alpha - 29. \tag{5}$$

If $x \geq 6$, then from Inequality (5) we have $m_i \geq 0$. If $x = 5$, then from: Inequality (5), the fact that $z \geq 3$, 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 $x = 3$. In the case when $x = 3$, it is easy to check that $z \leq 10$, 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 $|V(\mathcal{S})| \leq 35$ 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

$$m_i \geq 5x + 3\alpha + 4z - 6\lfloor (z-3)/2 \rfloor - 29 \tag{6}$$
$$\geq 5x + z + 3\alpha - 20. \tag{7}$$

Since $x \geq 3$ and $z \geq 3$, if $x \geq 4$, $z \geq 5$, or $\alpha \geq 2$, by (7) we get $m_i \geq 0$. Assume now that $x = 3$, $z \in \{3, 4\}$, and $\alpha \in \{0, 1\}$. Because $x,\ y,\ z,\ \alpha$ are all integers, it is easy to see from (3), that the only possible case is when $x = 3,\ y = 3,\ z = 4,\ \alpha = 0$. 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 by* **Reducing** *followed by an invocation to* **Clean**. *Then i is not worse than a $(3, 5)$-branch and the amortized cost m_i of i is non-negative.*

Proof. Steps 0, 1, and 2 of **Reducing** followed by invocations to **Clean** do not involve any branching operations, and hence, they are more efficient than $(3, 5)$ branches. Moreover, by Lemma 3.2, the amortized cost corresponding to any non-branching operation is non-negative. Lemma 3.4 shows that Step 3 of **Reducing** followed by an invocation to **Clean** is not worse than a $(3, 5)$-branch and has a non-negative amortized cost, and Lemma 3.6 establishes the same fact for Step 4 of **Reducing**. □

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 can either remove two edges and two vertices or three edges and 2 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, 6 edges and 3 vertices are removed, and no degree-1 vertices are created since all the neighbors of the two vertices that we removed must be of degree-3 (otherwise we would have an alternating tree of size at least 5, which is not possible since **Reducing** is not applicable at this point). 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 0 and 3 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, and let v and w be its neighbors. 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. 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. 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 that is performed in **Reducing** or one that is performed in **Clean**. If O is an operation that is performed in **Reducing**, 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 **Reducing**, by the above discussion, O must be the 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 3 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. In all cases, either the amortized cost or a lower bound on it, is given in 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	$m_i \geq 0$

Figure 3: The parameters of the operations

Definition 3.9 *Let α be a node in the search tree $\mathcal{T}$ of the algorithm corresponding to (G, k). We define the label of α to be the reduction in the parameter along the branch in the tree from the parent of α to α. If α is the root of $\mathcal{T}$, then the label of α is k, where k is the original parameter.*

Definition 3.10 *Let $P = (\alpha_i, \ldots, \alpha_j)$ be a path in the search tree $\mathcal{T}$ corresponding to the execution of the algorithm. Let x_1 be the number of nodes on P of label 1 corresponding to the 1-side of the $(1,3)$ branches, x_3 the number of nodes of label 3 corresponding to the 3-side of the $(1,3)$ branches, and x_2 the number of nodes of label 2 corresponding to the 2-side of the $(2,5)$ branches. Let d be the sum of the labels of all the nodes on P that correspond to folding operations plus the sum of all the surplus s_i over every category 4 operation i on P. The path P is said to be compressible if $d \geq 2x_1 + 2x_3 + x_2$.*

Informally speaking, d corresponds to the "extra" (or bonus) reduction in the parameter resulting from efficient operations along the path. Therefore, if a path P is compressible, then along P we can find enough "bonus" reduction in the parameter that can be used to render all inefficient operations (1-side, 3-side of $(1,3)$ branches, and 2-side of $(2,5)$ branches) along P at least as efficient as $(3,5)$ branches.

Proposition 3.11 *Let $\mathcal{T}$ be the search tree corresponding to the execution of the algorithm on input (G,k). Suppose that all the branching operations performed by the algorithm can be classified as $(1,3)$, $(2,5)$, and other branching operations (a,b), where (a,b) is at least as efficient as a $(3,5)$-branch. If every root-leaf path in $\mathcal{T}$ is compressible, then the number of leaves $L(k)$ in $\mathcal{T}$ is bounded by $O(r^k)$ where r is the root of the polynomial $x^5 - x^2 - 1$.*

Proof. Since every root-leaf path in $\mathcal{T}$ is compressible, the reduction in the parameter along any root-leaf path corresponding to folding operations and the surplus of category 4 operations d satisfies the inequality $d \geq 2x_1 + 2x_3 + x_2$. This can be interpreted as follows. For every 1-side of a $(1,3)$-branch we have an "extra" reduction in the parameter of value 2, for every 3-side of a $(1,3)$-branch we have an "extra" reduction in the parameter of value 2, and for every 2-side of a $(2,5)$-branch we have an "extra" reduction in the parameter of value 1. Thus, the nodes labeled 1 and 3 that correspond to the 1-side and 3-side of the $(1,3)$ branches, and the nodes labeled 2 that correspond to the 2-side of the $(2,5)$ branches, can be re-labeled (combined with the nodes corresponding to the efficient operations that reduce the parameter on the path) to yield nodes labeled 3, 5, and 3 that correspond to the 3-side, 5-side, and 3-side branches respectively. All other nodes correspond to operations that are at least as efficient as $(3,5)$ branching. If we compress every root-leaf path in the tree, we transform the tree into a tree that corresponds to an algorithm whose branches are at least as efficient as $(3,5)$ branches. This completes the proof. $\square$

To prove that the number of leaves in the search $\mathcal{T}$ is bounded by $O(r^k)$, where r is the positive root of the polynomial $x^5 - x^2 - 1$, by the above proposition, it suffices to show two things. The first is that every branching operation in category 4 is not worse than a $(3,5)$-branch which was done in Theorem 3.7, and the second is that every root-leaf path in $\mathcal{T}$ is compressible. We show next that every root-to-leaf path in $\mathcal{T}$ is compressible.

Proposition 3.12 *Let $S = (\alpha_i, \alpha_{i+1}, \ldots, \alpha_{i+l-1}, \alpha_{i+l})$, $l > 0$, be a subpath of a path P in $\mathcal{T}$. Suppose that none of the nodes α_j, $i < j < i+l$, corresponds to a 3-regular graph. If α_{i+l} corresponds to a 3-regular graph then the subpath $S = (\alpha_i, \ldots, \alpha_{i+l-1})$ is compressible.*

Proof. Let m_i, n_i be the number of edges and vertices of the graph at α_i, and m_{i+l}, n_{i+l} those at α_{i+l}. Since the graph at α_{i+l} is 3-regular, we have $m_{i+l}/n_{i+l} = 3/2$. Let $m' = m_i - m_{i+l}$, $n' = n_i - n_{i+l}$. Since $m_i/n_i \leq 3/2$ (the graph has degree bounded by 3), it is easy to see that $m'/n' \leq 3/2$.

Let x_f be the number of folding operations on S, 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 S. 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 S. Since $m'/n' \leq 3/2$, we can write

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

Arranging (8), we get

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

By the linearity of the amortized cost, 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 group of operations λ must cover all the E_λ edges removed by that type, and since each vertex can cover at most three edges, $K_\lambda \geq \lceil E_\lambda/3 \rceil$. Hence, $2E_\lambda - 3V_\lambda \geq -3S_\lambda + M_\lambda/2$. Using this inequality and the parameters of the operations given in Figure 3, we get: $3V_f - 2E_f \leq \frac{5}{2}x_f$, $2E_1 - 3V_1 \geq 3x_1$, $2E_3 - 3V_3 \geq 3x_3$, $2E_2 - 3V_2 \geq 3x_2$, $2E_5 - 3V_5 \geq \frac{1}{2}x_5$, $2E_r - 3V_r \geq -3S_r + M_r/2$. Substituting these bounds in Inequality (9) and arranging it we get:

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

By Proposition 3.8, a category 4 operation has non-negative amortized cost. Hence $M_r \geq 0$. Let $d = x_f + S_r$ be the reduction in the parameter along S caused by folding and the surplus of category 4 operations (as defined in Definition 3.10). From Equation (10), we have $d \geq x_2 + (x_1 + x_3) + x_f/6 + x_5/6 + M_r/6$. Note that the graph becomes 3-regular at most once along S (this can only happen at node α_i), and hence, $x_1 + x_3 \leq 1$. Also, since the graph becomes 3-regular at node α_{i+l}, it is not difficult to verify the following: Either we must have at least one folding operation along S, or at least one operation of those described in case (3) of Theorem 2.3. This is true since these are the only operations that could make the graph become 3-regular (note that by part (2) of Remark 2.4, we can assume that initially G is connected. The only way to create a 3-regular component during the execution of the algorithm is either by a folding operation or by an operation in case (3) of Theorem 2.3, which adds an edge to the resulting graph). Since the amortized cost of the operation in case (3) of Theorem 2.3 was proved in Lemma 3.4 to be ≥ 5, it follows that if the graph becomes 3-regular along S then either $x_f \geq 1$ or $M_r \geq 5$. Thus, if $x_1 + x_3 = 1$, since d is an integer, we have $d \geq x_2 + 2 = x_2 + 2(x_1 + x_3)$. If $x_1 + x_3 = 0$, again we have $d \geq x_2 + 2(x_1 + x_3)$. It follows that $d \geq x_2 + 2(x_1 + x_3)$, and the subpath S is compressible. □

Proposition 3.13 *Let G be a nice graph and let m and n be the number of edges and vertices in G, respectively. Then $m/n \geq 6/5$.*

PROOF. Since G is nice, no folding operation is safe. It follows that no two degree-2 vertices are adjacent. Let n_2 and n_3 be the number of degree-2 and degree-3 vertices in G, respectively. Then $n = n_2 + n_3$ (G is clean). Since no two degree-2 vertices in G are adjacent, every edge in the graph either joins two degree-3 vertices, or a degree-2 vertex to a degree-3 vertex. It follows that $m = 2n_2 + (3n_3 - 2n_2)/2$. Moreover, from the same hypothesis it follows that $3n_3 \geq 2n_2$. Combining the above three relations we get the desired result. □

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

PROOF. Let P' be an arbitrary root-leaf path in $\mathcal{T}$, and let α be the last node on P' that corresponds to a 3-regular graph if such a node exists, otherwise, let α be the root of P'. If α corresponds to a 3-regular graph, then one can easily show using Proposition 3.12 and a simple inductive argument (on the number of nodes that correspond to 3-regular graphs), that the subpath of P' from the root to the node preceding α is compressible. To show that P' is compressible, it remains to show that the subpath from α to the leaf of P', henceforth denoted by P, is compressible. Let n, m, and k be the number of vertices, edges, and parameter, respectively, in the graph at the root node of P (i.e., α). Then the following inequalities hold:

$$m/n \leq 3/2 \tag{11}$$

$$k \geq n/2 \tag{12}$$

Inequality (12) is true for any 3-regular graph, and if node α does not correspond to a 3-regular graph, then α is the root node of P', and Inequality (12) holds by Proposition 2.1.

Let α_0 be the last node on P corresponding to a branching operation (i.e., all nodes after α_0 correspond to folding or non-branching operations). 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 on P from the root to the node preceding α_0. Let e', n', k', and s', be the number of edges eliminated, number of vertices eliminated, reduction in the parameter, and the surplus, respectively, in the remaining subpath of P starting at α_0. Each operation performed by the algorithm reduces the parameter k by the coefficient listed in the table in Figure 3 for that operation. For instance, each folding operation reduces the parameter by 1. Since x_f is the number of folding operations performed by the algorithm, the total reduction in the parameter attributed to the folding operations is x_f. Similarly for the other operations. By part (1) in Remark 2.4, we can assume that the parameter k is not larger than the size of a minimum vertex cover of G. Thus, at the leaf node of the path P we can write the following inequality:

$$x_f + x_1 + 2x_2 + 3x_3 + 5x_5 + K_r + k' \geq k \tag{13}$$

One point in the above inequality needs clarification. As it was shown before, the partial cover grown at each step of the algorithm is always contained in a minimum vertex cover. Thus, when the algorithm terminates along any path in the search tree, either the computed cover is a minimum vertex cover, and hence, the reduction in the parameter along the path is exactly equal to the size of the minimum vertex cover, or the size of the resulting cover has exceeded the parameter k, and hence, the reduction in the parameter along the path is greater than k. This justifies writing the above inequality.

According to our algorithm, before it performs a branching operation the graph must be nice, i.e., there are no degree-0 and degree-1 vertices, and no safe folding is applicable. It follows that before the last branch at node α_0 the graph is nice. By Proposition 3.13, at the node preceding α_0 the ratio of the number of edges to the vertices in the graph is not smaller than 6/5. Thus, we can write the following inequality:

$$\frac{m - E_f - E_1 - E_3 - E_2 - E_5 - E_r}{n - V_f - V_1 - V_3 - V_2 - V_5 - V_r} \geq \frac{6}{5} \tag{14}$$

In a similar way to that in Proposition 3.12, we can define the amortized cost for each type of operations λ ($\lambda = 1, 2, 3, 5, r, f$), by: $M_\lambda = 5E_\lambda - 6V_\lambda + 6S_\lambda - 3K_\lambda$. 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 3 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 the previous inequalities with Inequalities (11), (12), (13), (14), and arranging the terms we get:

$$5x_f \quad \geq \quad 6x_1 + 6x_2 + 6x_3 + x_5 + M_r - 6S_r - 3k' \tag{15}$$

Hence:

$$x_f + S_r \quad \geq \quad x_2 + (x_1 + x_3) + x_5/6 + M_r/6 + x_f/6 - k'/2 \tag{16}$$

Accommodating for the surplus s' in Inequality (16), we get:

$$\begin{aligned} x_f + S_r + s' \quad \geq \quad & x_2 + (x_1 + x_3) + x_5/6 + M_r/6 \\ & + x_f/6 + s' - k'/2 \end{aligned} \tag{17}$$

Let d be the reduction in the parameter along the path P corresponding to folding and category 4 operations (as defined in Definition 3.10), then $d = f + S_r + s'$. By Proposition 3.8, the amortized cost of category 4 operations $M_r \geq 0$. It follows that

$$d \quad \geq \quad x_2 + (x_1 + x_3) + x_f/6 + s' - k'/2 \tag{18}$$

We can assume that the size of a minimum vertex cover in the resulting graph at α_0 is larger than 16 (i.e., $k' \geq 17$); otherwise, either the size of the graph would be bounded by 50 (since the graph is nice and has degree bounded by 3) and no branching is needed since step 0 of **Reducing** would be applicable, or the algorithm can reject the instance with no branching if the size of the graph is larger than 50 (since covering such a graph with fewer than 17 vertices is not possible). If the branch at α_0 is a side of a branch that is not worse than a $(3, 5)$-branch (like a branch that occurs in **Reducing**), then the branch at α_0 along P does not contribute to the parameters x_1, x_2, x_3 involved in the path compression. To show that P is compressible in this case, we need to show that $d \geq x_2 + 2(x_1 + x_3)$. Since $s' \geq k' - 5$ (from the definition of the surplus), and $k' \geq 17$, we have $d \geq x_2 + (x_1 + x_3) + x_f/6 + 7/2$. Since d is an integer, $d \geq x_2 + (x_1 + x_3) + 4$. Note that at most one node on P (node α) can be a side of a $(1, 3)$-branch. This is true since only the root node on P can correspond to a 3-regular graph. Thus, $x_1 + x_3 \leq 1$. Hence, $d \geq x_2 + 2(x_1 + x_3)$, and the path P is compressible. Now suppose that α_0 contributes to the parameters x_1, x_2, or x_3. We assume that α_0 is the 2-side of a $(2, 5)$-branch (the other cases when α_0 is the 1-side or 3-side of a $(1, 3)$-branch are easier to handle). Then $s' = k' - 2$ (all the reduction in the parameter along P after node α_0 is a surplus). To show that P is compressible, we need to show in this case that $d \geq (x_2 + 1) + 2(x_1 + x_3)$ (α_0 contributes by 1 to the number of 2-side branches x_2 on the path on P up to the node before α_0). Since $k' \geq 17$ and d is an integer, we get $d \geq x_2 + (x_1 + x_3) + 7 = (x_2 + 1) + (x_1 + x_3) + 6$. By the same token, $x_1 + x_3 \leq 1$, and $d \geq (x_2 + 1) + 2(x_1 + x_3)$. Thus, P is compressible. This shows that in all cases P is compressible. It follows that the path P' is compressible, and hence, every root-leaf path in $\mathcal{T}$ is compressible. This completes the proof. □

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

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 corresponding to the execution of 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, it follows from Proposition 3.11 that the number of leaves in $\mathcal{T}$ is $O(r^k)$, where $r \leq 1.194$ is the positive root of the polynomial $x^5 - x^2 - 1$. Now we claim that along every root-leaf path in $\mathcal{T}$ the time spent by the algorithm is linear in the size of the graph (which is $O(k)$). 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. Also, it is not difficult to search for the vertices that need cleaning after each operation, since the search can be localized. It follows that along the whole path in the tree the time taken by **Clean** is $O(k)$. By the same argument, the total time taken by **Fold** along a root-leaf path in $\mathcal{T}$ is $O(k)$. Also the time taken by a branching operation is constant, which means that the total time taken by branching operations along a path is $O(k)$. The only thing that remains to be analyzed is the time taken by **Reducing**. Observe the following. Even though the statements in the subroutine **Reducing** ask for checking whether a certain structure exists in the graph, which normally would take $O(k)$ to do, we only need to check the existence of such a structure in the vicinity of a vertex that we need to branch at. This is true since the analysis only assumes that no such structure exists in the vicinity of the vertex and not in the whole graph. Thus, the search can be localized to a specific vertex. For instance, if the graph is 3-regular, then according to the algorithm, in such a case we pick any vertex and branch on it. Now we can pick a vertex v, and apply the subroutine **Reducing** to the vicinity of v. If the algorithm does not find any of the structures mentioned in **Reducing** in the vicinity of v, then the algorithm branches at v (the size of the vicinity of a vertex can be upper bounded by a constant). If the algorithm finds one of the structures specified in **Reducing**, then this structure is removed. The time spent by the subroutine now is proportional to the size of the structure removed. This means that if we remove a structure with a certain number of vertices, then those vertices will never contribute to the search time in later operations along the path. It should not be difficult to see that with this implementation of the algorithm, and using suitable data structures, the time spent by **Reducing** along a path in $\mathcal{T}$ is $O(k)$. Now the running time of the algorithm is $O(n + k\sqrt{k} + 1.194^k k) = O(1.194^k k + n)$, where $O(k\sqrt{k} + n)$ is the pre-processing time. Niedermeier and Rossmanith showed how to get rid of the size of the kernel in the running time [18]. Applying their techniques, we conclude that the running time of the algorithm **VC3-solver** is $O(1.194^k + n)$. □

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 used in [5]. The algorithm for IS-3 runs in time $O(1.1254^n)$, and slightly beats Beigel's $O(1.1259^n)$ time algorithm [2].

Lemma 4.1 (Lemma 6.1, [5]) *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 $O(1.1254^n)$.*

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 $k_i \leq (2n_i + 1)/3$. Thus, by Theorem 3.15, a maximum independent set for the component C_i can be constructed in time $O(1.194^{(2n_i+1)/3} + n_i)$, which is $O(1.1254^{n_i})$. In conclusion, a maximum independent set in the graph G can be constructed in time $O(1.1254^{n_1} + \cdots + 1.1254^{n_k})$. Now it is fairly straightforward to verify that $O(1.1254^{n_1} + \cdots + 1.1254^{n_k}) = O(1.1254^n)$. $\square$

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 $O(1.194^k + n)$ and improves Chen et al.'s $O(1.237^k + kn)$ time algorithm [6]. Our algorithm for IS-3 runs in time $O(1.125^n)$ and improves Beigel's $O(1.126^n)$ time algorithm [2].

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 later 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 optimization problems.

References

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

[2] 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.

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

[4] L. Cai and D. Juedes, On the existence of subexponential-time parameterized algorithms, available at `http://www.cs.uga.edu/ ~ cai/`.

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

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

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

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

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

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

[11] R. Impagliazzo, R. Paturi, and F. Zane, Which Problems Have Strongly Exponential Complexity?, *Journal of Computer and System Sciences (JCSS)* **63-4**, (2001), pp. 512-530.

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

[13] D. Johnson and M. Szegedy, What are the least tractable instances of max. independent set?, *Proceedings of the (SODA'99)*, (1999), pp. 927-928.

[14] 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).

[15] 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).

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

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

[18] R. Niedermeier and P. Rossmanith, A general method to speed up fixed-parameter-tractable algorithms, *Information Processing Letters* **73**, (2000), pp. 125-129.

[19] J. M. Robson, Algorithms for maximum independent set, *J. of Alg.* **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] R. E. Tarjan and A. E. Trojanowski, Finding a maximum independent set, *SIAM Journal on Computing* **7**, (1986), pp. 537-546.