CSE6140 Fall 2022 Project: Minimum Vertex Cover (Group 40)

ANAND RADHAKRISHNAN, Georgia Institute of Technology, USA
SEONGMIN LEE, Georgia Institute of Technology, USA
JESUS EMMANUEL ARIAS, Georgia Institute of Technology, USA
BENJAMIN ALEXANDER WILFONG, Georgia Institute of Technology, USA

ACM Reference Format:
Anand Radhakrishnan, Seongmin Lee, Jesus Emmanuel Arias, and Benjamin Alexander Wilfong. 2022. CSE6140 Fall 2022 Project: Minimum Vertex Cover (Group 40). 1, 1 (December 2022), 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

`Minimum Vertex Cover` is an NP-Hard problem for covering edges in an unweighted graph. Efficient calculation of exact and approximate solutions is valuable because it can be used to solve other hard problems with real world applications, such as the use of minimum set cover to select a subset of tools each with different capabilities to achieve a larger set of capabilities overall. `Minimum Vertex Cover` has also been used more directly for solving multi-document summarization and finding phylogenetic trees based on protein domains [1, 12].

In this report we present four different approaches to solving the `Minimum Vertex Cover` problem. This first is an exact algorithm that has the potential to take up to exponential time, but with careful bounds can run much quicker. The second is an approximate algorithm that yields a solution with a known quality. These two algorithms give the same result every time they are ran. Unfortunately, the exact algorithm, despite its bounds, can still take a very long time to find solutions and the approximate algorithm often provides solutions with relatively low qualities when compared to the two other local search algorithms we present. The first of these is a k-vertex cover finding algorithm that implements a linear time complexity exchange process and edge weighting with a forgetting mechanism. The second solves the `Minimum Vertex Cover` using a random restart hill climbing approach using a two-improvement.

To compare these methods we first perform time trials and record the best solution found and the time it took to find. Because the local search algorithms are less deterministic and their run-time and solution quality, we compare them via box-plots of their run-times at different qualities, qualified run-time distributions, and solution quality distributions. These allow use to better analyze and compare these non-deterministic approaches. We ultimately conclude that unless an exact solution for a relatively small graph is needed, local search should be used to find approximate vertex covers with acceptable quality in a reasonable amount of time.

Authors' addresses: Anand Radhakrishnan, aradhakr34@gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA; Seongmin Lee, slee3473@gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA; Jesus Emmanuel Arias, jesus.arias@gtri.gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA; Benjamin Alexander Wilfong, bwilfong3@gatech.edu, Georgia Institute of Technology, Atlanta, Georgia, USA.

XXXX-XXXX/2022/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PROBLEM DEFINITION

The `Minimum Vertex Cover` problem can be formulated as in the course project description: for given an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, `Minimum Vertex Cover` problem aims to find out a subset $C \subseteq V$ with the minimum $|C|$ such that $\forall (u, v) \in E$: $u \in C \vee v \in C$. In the rest of this reports, the notations in the problem definition will be repeatedly used with the same meaning unless we specify some other meanings.

3 RELATED WORK

Even though it is a well-known fact that the `Minimum Vertex Cover` problem is NP complete [15], there have been many efforts to efficiently solve the `Minimum Vertex Cover` problem as it has numerous applications [10, 21, 24].

One direction to get the exact solutions for the `Minimum Vertex Cover` problem is the branch-and-bound approach, which iterates all the possible solutions while pruning unpromising branches based on the lower and upper bounds of the solutions. Wang et al. [25] propose a branch-and-bound algorithm to efficiently get the exact solution for the `Minimum Vertex Cover` problem with two lower bounds, each of which is based on the degree of vertices and MaxSAT reasoning. Akiba et al. [2] empirically examine the algorithms that yield the exact solutions for the `Minimum Vertex Cover` problem to identify the gap between theoretical and practical performances. Fixed-parameter tractable algorithms have also been leveraged to get the exact solutions for the `Minimum Vertex Cover` problem [11, 17, 20], but most of them are primarily focused on theoretical derivation rather than the practical applications. Also, as the vertex cover is a NP complete solution, there is no algorithm that gives the exact solutions for the `Minimum Vertex Cover` problem in polynomial time to our knowledge.

For better scalability, some literature has proposed the algorithms to reduce the time complexity by sacrificing the quality of the solution: approximation algorithms and local search. The approximation (ratio 2) Multiple approximation algorithms that output a valid vertex cover of a graph, which is not necessarily *minimum* vertex cover, have been presented [6, 9]. However, the approximation ratio of the simple approximation algorithms is 2; in the worst case, the number of vertices in the returned solution is twice as many as the optimal number. To resolve this limitation of the approximation ratio, multiple variations of the approximation algorithm has

been presented [7, 14, 18], but the improvement of the algorithms proposed so far is marginal.

Another branch of scalable algorithms that give a vertex cover of a graph is local search algorithms that have no guarantees but are usually near optimal [3, 5, 8, 13, 19, 22, 23, 26]. However, even though local search algorithms are likely to yield a good solution, it is hard to use these algorithms for the applications that require guarantees for the quality of the solution due to the absence of the guarantees.

There are trade-offs in the algorithms for the `Minimum Vertex Cover` problem that have been proposed so far. Therefore, in order to choose an algorithm to solve `Minimum Vertex Cover` problem in practice, a comprehensive understanding of each algorithm is essential. Among many approaches to the `Minimum Vertex Cover` problem, we narrow down our focus into branch-and-bound algorithms for the exact solutions, approximation algorithms with a heuristic, and two local search algorithms. The details of our algorithms will be elaborated in the Section 4.

4 ALGORITHMS

4.1 Branch-and-Bound

The branch-and-bound algorithm is an exact method for generating a solution to the `MVC` problem. The algorithm implemented is based on the work of Akiba and Iwata [2] and Wang et al. [25].

This involves creating a frontier set F, that contains the current set of subproblems. At each step the algorithm explores the latest subproblem in F with each subproblem defined as:

$$\texttt{F(i)} = (\texttt{current\_vertex}, \texttt{included}, (\texttt{parent\_vertex}, \texttt{parent\_included}))$$

For each subproblem, the subgraph G' is reduced depending on whether the current vertex is included or not before picking a new maximum degree vertex. As in Akiba [2], if a vertex is included, then it is added to the vertex cover and likewise removed from the subgraph. Conversely, if a vertex is excluded, then its neighbors (denoted by $N(v)$) are added to the vertex cover and removed from the subgraph. Concurrently, sets for the so-far explored vertices and included vertices are also maintained.

The partial vertex cover C is then checked to determine if it is a valid vertex cover to G. If so the algorithm is set to backtrack, and if the solution is better, B is overwritten with C. If not a solution, the current subgraph is checked to determine if it is promising by comparing the lower bound to the current best vertex cover ($LB < |B|$). If promising, the algorithm takes the highest degree node of the current subgraph G' and creates two new subproblems, one including said vertex, and another excluding it, and adds them to the frontier set. Otherwise, if not promising, the algorithm is set to backtrack. Furthermore, the method is biased such that the subproblem including the new vertex is evaluated first. As such, backtracking only takes place if a solution is found, or a path is determined to be unfeasible ($LB > |B|$). This results in a quick evaluation to a solution that can then be used as an upper bound for the remaining subproblems.

As a lower bound, several options are explored. The first is given by Kleinberg and Tardos § 10.2 [16] with lower bound being defined as $lb = E/V$. Another lower bound option defined in Wang et al. [25] as $lb = E/\max(deg(u))$ was also implemented. In practice, the performance of the algorithm was largely determined by the exponential scaling of the frontier set. Additionally, the approximation algorithm of § 4.2 was also used as a lower bound defined as $lb = \frac{1}{2}A(G')$. While this served as a sharper lower bound, the cost to evaluate the approximation algorithm reduced the overall efficiency of the algorithm. Ultimately the original lower bound formulation was used due to its cheap cost of evaluation.

Operations on the subgraph are done in-place, in an effort to reduce the amount of time spent on generating subgraphs for each subproblem, and likewise reduce the amount of memory in caching subproblems, similarly to Zhong [27]. In the case that the algorithm backtracks along the frontier set, the explored set of vertices is backtracked until it matches the parent vertex given by the queued subproblem in the frontier set. With each step, a vertex is removed from the explored vertex set and added back to the subgraph G', along with its associated edges, until the subgraph is returned to the parent state.

Lastly, due to the exponential nature of branch-and-bound, a cutoff time is used to terminate the program early. If the cutoff time is reached, then the algorithm returns the current best vertex cover calculated by branch-and-bound.

Algorithm 1 is the pseudo-code for our branch-and-bound algorithm. The overall time complexity is given by $O(2^{|V|})$, as there are two possible subproblems per vertex in G (included vs not included). Then for each subproblem we must perform operations on a graph of size (V, E), where the lowest possible time complexity is $O(|V + E|)$. This gives us a final time complexity of $O(|V + E| \cdot 2^{|V|})$. The space complexity is ultimately determined by the number of subproblems that can possibly be in the frontier set at once plus the cost of storing the graph itself. Each subproblem is stored in the frontier set as `F(i) = (current_vertex, included, (parent_vertex, parent_included))`. As such the space complexity is given by the number of possible subproblems $O(2^{|V|})$, plus $O(|V+E|)$ for storing the graph itself for an overall space complexity of $O(2^{|V|} + |V + E|)$.

4.2 Approximation

Our approximation algorithm for the `Minimum Vertex Cover` problem is based on the *edge deletion* [9] method, which iterates the three steps to

(1) select one of the remaining edges,
(2) add the two endpoints of the selected edge to the vertex cover set C, and
(3) remove all the edges with at least one endpoint in the set C

until there is no remaining edge. As the heuristic to select an edge in the step (1), we evaluated the degree of each edge by summing up the degree of the two vertices consisting of the edge. The formal definition of the *degree of edge* is in the Definition 1.

Definition 1 (Degree of Edge). *For an edge $e = (u, v)$, we define the degree of edge $deg(e) = deg(u) + deg(v)$. Here, $deg(u)$ and $deg(v)$ are the conventional degree of the nodes.*

Algorithm 1: Branch-and-Bound algorithm

Data: Graph $G = (V, E)$
Result: Vertex Cover $C \subseteq V$

$F \leftarrow \emptyset$ // Each entry in F = [current vertex, included, (parent_vertex, parent_included)]
$C \leftarrow \emptyset$, $B = V$, $G' = (V', E') \leftarrow G$.copy()
explored_set $\leftarrow \emptyset$, include_set $\leftarrow \emptyset$
$v_{max} \leftarrow \max(deg(V'))$, parent = (-1, False)
$F \leftarrow F \cup [(v_{max}, False, parent), (v_{max}, True, parent)]$
while $F \neq \emptyset$ *and time* < *cutoff* **do**
  v, *include*, *parent* = F.pop()
  if *include == True* **then**
    $G' \leftarrow G' \backslash v$
    add v to explored_set, add True to include_set
  else if *include == False* **then**
    $G' \leftarrow G' \backslash N(v)$ // $N(v)$ are neighbors
    for *n in* $N(v)$ **do**
      add n to explored_set, add True to include_set
    end
  C = all explored vertices that are included
  if C *is a vertex cover* **then**
    backtrack = True
    if $|C| < |B|$ **then**
      $B = C$
  else
    if $|C|$+ *compute_lb*(G') < $|B|$ **then**
      $parent = (v, include)$
      $v_{max} \leftarrow \max(deg(V'))$
      $F \leftarrow F \cup [(v_{max}, False, parent), (v_{max}, True, parent)]$
    else
      backtrack = True
    end
  end
  if *backtrack == True and* $F \neq \emptyset$ **then**
    ~, ~, (parent_ver, parent_inc) $\leftarrow$ frontier[end]
    if *parent_ver is in explored_set* **then**
      index $\leftarrow$ explored_set.index(parent_ver) + 1
      while *index*<*len(explored_set)* **do**
        v_{rem} = explored_set.pop(), include_set.pop()
        $C \leftarrow C \backslash v_{rem}$, $G' \leftarrow G' \cup v_{rem}$
        $neighbors = G.neighbors(v_{rem})$
        for *n in neighbors* **do**
          if *n is in* G' *and n is not in* C **then**
            $E' \leftarrow E' \cup (n, v_{rem})$
          end
        end
    else if *(parent_ver, parent_inc) = (-1, False)* **then**
      explored_set = $\emptyset$, include_set = $\emptyset$
      $G' = (V', E') \leftarrow G$.copy()
end
C = B
return C

We select the edge with the maximum degree as the two vertices of the edge are likely to cover many edges and therefore reduce the number of vertices in the Vertex Cover set.

Algorithm 2 is the pseudo-code for our approximation algorithm.

Algorithm 2: Approximation algorithm

Data: Graph $G = (V, E)$
Result: Vertex Cover $C \subseteq V$

1 $G1 = (V1, E1) \leftarrow G$.copy()
2 $C \leftarrow \emptyset$
3 **while** $E1 \neq \emptyset$ **do**
4 Compute $deg(v)$ for $v \in V$
5 $(u, v) \leftarrow$ MaxDegreeEdge(E)
6 $C \leftarrow C \cup \{u, v\}$
7 $G1$.remove(u)
8 $G1$.remove(v)
9 **end**
10 **return** C

Let's look into the time complexity of our approximation algorithm. First, the loop in the line 3 is iterated $O(\min(|V|, |E|))$ times; the loop will be quit once all the edges are deleted from the graph $G1$ or all the vertices are appended to the set C. Degree computation in the Line 4 takes $O(|E|)$ as we iterate over all the edges in the graph and increase the degree of the endpoints of each edge. The line 6 to iterate all the edges and retrieve the edge with the maximum degree also takes $O(|E|)$. Appending the two nodes to the set C is done in constant time while removing the two nodes from the graph takes $O(|E|)$ as well. Combining all the information, each iteration takes $O(|E|)$, while the number of iterations is $O(\min(|V|, |E|))$. Hence, the time complexity of our approximation algorithm is $O(|E| \min(|V|, |E|))$. In case of space complexity, copying the graph and storing the degrees for each edge and vertex are the only parts that require additional memory. This would give the space complexity of $O(|V| + |E|)$.

The correctness of our approximation algorithm can be proven by showing that the C returned by the algorithm can always cover the graph G. If there exists a case that the returned $C \subseteq V$ does not cover the graph G, it means that there exists an edge $e = (u, v) \in E$ such that $u \notin C$ and $v \notin C$. In our algorithm, the edge $e = (u, v)$ can be removed from the graph $G1$ when at least one of the vertices u and v are added to C. Hence, $e = (u, v)$ will not be removed from the graph $G1$ and remains in its edge set $E1$; $e \in E1$. However, for C to be returned, the set $E1$ should be empty, which contradicts to $e \in E1$. Therefore, we can conclude that the returned set $C \subseteq V$ covers the input graph G.

To derive the approximation guarantee of our algorithm,

Claim 1. *Let's define E' as the set of edges that are selected in the line 5 of the Algorithm 2; if the loop iterates m times, $E' = \{e_1, e_2, \ldots, e_m\}$. Then, there are no two edges in E' that share an endpoint; i.e., if $e_i, e_j \in E'$ and $i \neq j$, e_i and e_j do not share any endpoints.*

The claim 1 is true as each vertex is chosen at most once, and all the edges having the vertex as an endpoint are removed from

the graph after the vertex is added to C. Then, we can get the approximation ratio of 2 from the Claim 2.

Claim 2. *Let the number of vertices in the minimum vertex cover of the graph G be $OPT(G)$. Then, for the vertex cover C of the graph G returned by the Algorithm 2, $OPT(G) \leq |C| \leq 2 \times OPT(G)$.*

Proof. As we know that C is a vertex cover of the graph G, $|C| \geq OPT(G)$ by the definition of the $OPT(G)$. Let $E' \subseteq E$ be the set of edges selected in the line 5 of the Algorithm 2. As one edge is selected at every iteration, $|E'|$ is the number of iterations, while $|C| = 2|E'|$ as two different vertices, which are endpoints of the selected edges, are added at every iteration. Since $E' \subseteq E$, the minimum vertex cover covers all the edges in E'; at least one endpoint of each edge in E' should be in the minimum vertex cover. As there are no two edges in E' sharing an endpoint, at least $|E'|$ vertices should be in the minimum vertex cover, which gives the inequality $|E'| \leq OPT(G)$. This gives $|C|/2 = |E'| \leq OPT(G)$ and therefore $OPT(G) \leq |C| \leq 2 \times OPT(G)$.

4.3 Local Search 1

The first local search algorithm we implemented is *NuMVC* as developed by Cai et. al. [8]. This approach is unique because it utilizes a two stage exchange operation and edge weighting with forgetting to improve performance. The comparative results to other state of the art heuristics for minimum vertex cover show that *NuMVC* is competitive and complementary with *PLS* and outperforms *COVER* and *EWCC* in 11 test cases.

At its heart, *NuMVC* is a k-vertex cover finding algorithm for an undirected graph $G = (V, E)$ where k is the number of verticies in the vertex cover. When a k-vertex cover is found, a vertex is removed and the algorithm begins a search for a $(k-1)$-vertex cover. The vertices $\{u, v\}$ of an edge e are called the endpoints of e. $N(v)$ denotes the neighboring vertices of vertex v. C is used to denote a candidate solution and C' is used to denote $C/\{v\}$, a candidate solution with one vertex removed. The state of vertex v is denoted by S_v where $S_v = 1$ implies vertex v is in C and $S_v = 0$ implies that vertex v is not in C. The weight of an edge e is denoted by $w(e)$. The cost of a candidate solution C with weights w on graph G is given by

$$cost(G, C) = \sum_{e \text{ not covered by } C} w(e).$$

The *dscore* of a vertex v is defined by

$$dscore(G, v) = cost(G, C) - cost(G, C')$$

where $C' = C/\{v\}$.

The use of a simpler two stage exchange yields an exchange with complexity $|R| + |A|$ rather than the commonly used simultaneous exchange approaches with complexity $|R| \cdot |A|$ where R and A are the set of candidate vertices for removing and adding. The first stage of this approach is to select the vertex in the graph with the highest *dscore* and removing it from the candidate solution C. The second stage is to select a random uncovered edge e with uniform probability and add the endpoint v with the higher *dscore* to the current solution C under the constraint that at least one of its neighbors has changed its state since the last time v was removed from the candidate solution. This approach may miss some more optimal exchanges, but balances complexity and quality, a common trade-off when implementing local search algorithms.

The use of edge weighting to improve performance is not a new development, but the inclusion of a forgetting mechanism is unique to the approach developed by Cai et. al. and their implementation. Edge weighting by itself involves initializing each edge with an integer weight of one, and adding one to the weight of each uncovered edge at the end of each iteration. Forgetting is implemented through two parameters γ and ρ. At each iteration, the average edge weight of all edges $\bar{w}$ is calculated. If $\bar{w} > \gamma$, then the weight of each edge is reduced to $\lfloor \rho w(e) \rfloor$. The forgetting mechanism ensures that weighting decisions from early in execution do not affect results later in execution when they are no longer relevant. Parameters ρ and γ were selected to be 0.3 and $0.5|V|$ as used by Cai et. al.

Pseudo-code for this approach is given in Algorithm 3. The space complexity is $O(|V| + |E|)$. The time complexity of the contents of the while loop is also $O(|V| + |E|)$. The total number of iterations performed given a cuttoff T is $O(T/(|V| + |E|))$.

Algorithm 3: Local search 1 algorithm

Data: Graph $G = (V, E)$, Cutoff T
Result: Vertex Cover $C \subseteq V$

Initialize $w(e) = 1$ for each edge e
Initialize $dscores(v)$ for each vertex v
Use aproximation algorithm to find an initial vertex cover C
$confChange \leftarrow \{1, ..., 1\}$
while $time < T$ **do**
  if C *is a vertex cover* **then**
    $C^* = C$
    Select vertex u with highest $dscore$
    $C \leftarrow C/\{u\}$
    $confChange(u) \leftarrow 0$
    $confChange(z) \leftarrow 1$ for each $z \in N(u)$
  end

  Select vertex u with highest $dscore$
  $C \leftarrow C/\{u\}$
  $confChange(u) \leftarrow 0$
  $confChange(z) \leftarrow 1$ for each $z \in N(u)$

  Randomly select and uncovered edge e
  Chose endpoint v of e with $confChange(v) = 1$ and higher $dscore$
  $C \leftarrow C \cup \{v\}$
  $confChange(z) \leftarrow 1$ for each $z \in N(v)$

  $w(e) \leftarrow w(e) + 1$ for each uncovered edge e
  if $\bar{w} > \gamma$ **then**
    $w(e) \leftarrow \lfloor \rho w(e) \rfloor$
  end
end
return C^*

4.4 Local Search 2

The second local search algorithm computes the minimum vertex cover as the complement of the maximum independent set on the original graph $G = (V, E)$. To prove this claim, we assume a vertex cover K that covers all the edges E in the graph. Then, $I = V \setminus K$ is an independent set. Suppose two vertices (u, v) in $V \setminus K$ share an edge e. Since they belong in $V \setminus K$, this means the edge e was not covered as both of its endpoints are not in K. Thus, K is not a valid vertex cover, which is a contradiction. Hence, if K is a minimum vertex cover, this means that its complement $I = V \setminus K$ is a maximum independent set.

The implementation follows the work of [3]. An independent set I is maximal if it has no vertices in $V \setminus I$ with neighbors exclusively in $V \setminus I$. The algorithm initializes a maximal independent set I by picking vertices at random from V and excluding its neighbors until it runs out of vertices to pick from. Next, it tries to find a vertex u in I with a valid 2-improvement. A 2-improvement exists for a vertex u if it has neighbors v and w that do not share an edge between them and have no other neighbors in I except u. The vertex u is selected by iterating over all possible valid candidates. The set of valid candidates is initially set to be equal to I and a 2-improvement is found by iterating over its neighbors to find v and w.

The algorithm repeatedly finds 2-improvements until no such candidate vertex is available. f no valid vertex is found for all candidate vertices, the algorithm ensures that the independent set I is maximal by including vertices in $V \setminus I$ that have no neighbors in I. It then returns $K = V \setminus I$ as the minimum vertex cover.

Initially, the list of candidate solutions is set to include all vertices in the maximal independent set I. However, once a vertex is found to have no 2-improvements, it is excluded from this list. If a 2-improvement is found for the vertex u, it is removed from I and its two neighbors are included in I. In the next iteration, both v and w are included and u is excluded from the candidate list. Additionally the neighbors of u, v and w are updated to indicate the shared edges to I, which are used to determine a valid 2-improvement in the next iteration. Finally, vertices in I that share a neighbor with u are included back in the candidate list.

Each iteration of the 2-improvement algorithm takes at most $O(|E|)$ time as it iterates over all possible edges to find a valid 2-improvement. Once a valid 2-improvement is found, it updates the candidate list and neighbors in $O(\Delta)$ time, where Δ is the maximum degree in the graph. The algorithm also minimizes the computational cost of subsequent iterations by minimizing the number of vertices in the candidate list. This is done by only excluding vertices in I from the candidate list and only including them back if they share a neighbor with the pruned vertex u.

The aforementioned algorithm repeatedly computes maximal independent sets until no 2-improvement is found. However, such an algorithm has a tendency to get stuck at a local maxima. To overcome this, we use an iterated local search algorithm by perturbing the local maxima to include one additional vertex u' at random from $V \setminus I$. Tabu search is used to pick the vertex u' by keeping track of the last iteration in which a given vertex was included in I. Next, the neighbors of u' that are part of I are pruned and their neighbors are updated to indicate shared edges in the independent set. Finally, I is made maximal by including vertices in $V \setminus I$ that share no edges with I, and the 2-improvement algorithm is restarted. Thus, the computational complexity of the local search algorithm is equal to $O(k|E|)$, where k is the number of restarts, which is set to 1000 for our purpose. Since the algorithm is essentially done in place, the space complexity is $O(|V| + |E|)$

5 EMPIRICAL EVALUATION

All execution times were measured using an M2 Apple Silicon processor paired with 16 GB of ram. We begin our analysis by presenting comprehensive run time results for each algorithm on 11 different graphs of various sizes and features from the 10th DIMACS challenge [4]. These results are given based on data from 10 trial runs on each instance. We then further analyze the non-deterministic local search algorithms on two specific graphs, power and star2. Results for more detailed runtime statistics, qualified runtime distributions, and solution quality distributions are provided to better understand the performance of these algorithms. These results are given based on data from 100 trial runs on each instance.

5.1 Comprehensive Time Trials

Time trials are averaged over 10 runs for each algorithm and instance. For the deterministic branch and bound and approximation algorithms, the results presented correspond to the repeatable final solution found in a given cutoff time. For the local search algorithms, the results presented correspond to the time for each trial to reach the worst quality final solution found over the 10 trials. For each algorithm and instance the final vertex cover size is presented alongside with the average time it took to obtain it and the relative error of it as defined by

$$\mathrm{e} = \frac{\mathrm{N_{alg}} - \mathrm{N_{OPT}}}{\mathrm{N_{OPT}}} \cdot 100\%$$

Table 1 presents the results from the different algorithms. First we address the results of the Branch-and-Bound algorithm. In general, the BnB algorithm does not run to completion for a cutoff time of 600 seconds, except for the *karate* graph. This is directly due to the exponential scaling of the problem with number of vertices. That being said, however, the quality of the solution is generally high, even when the algorithm terminates early. BnB was also able to find the optimal solution for two of the ten graphs in a relatively small amount of time. That being said, the only way to ensure that the optimal solution is found using branch-and-bound is to allow it to run to completion. Investigating the trace files for BnB, it is apparent that the algorithm typically finds an initial solution very quickly, then takes much longer to find a second solution or runs out of time entirely. This is likely due to the frontier growing more and more laterally as the algorithm progresses. Overall the performance of BnB is prohibitively expensive, and is only recommended for use when the absolute precision of an exact algorithm is required along with the luxury of unlimited time.

The performance of the remaining algorithms are substantially different, given that they are polynomial time algorithms for approximately solving the MVC problem. The approximation algorithm given in §4.2 is based on a heuristic with a provable bound on the

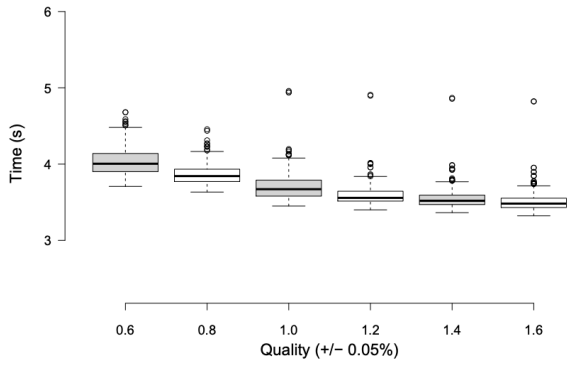

Fig. 1. Local Search 1 on power.graph

optimal solution. From Table 1 it is evident that the runtime is substantially improved from the exact algorithm. However, the quality of the solution suffers with the approximation algorithm generally having the highest error of the different algorithms considered.

The two local search algorithms also feature substantially improved runtime performance compared to the BnB algorithm. In general, LS1 features better accuracy than the approximation algorithm, even being able to find the optimal solution in four of the graphs. However, LS1 generally also carries a worse runtime performance than the approximation algorithm.

LS2 has the best runtime performance of all the algorithms, consistently running faster than both Approx and LS1. LS2 also has comparable error to LS1 for the non-exact algorithms. On 100 averaged runs, LS2 was able to find the optimal solution to the same four graphs as LS1 while generally having a lower error than Approx.

5.2 Local Search Runtime Statistics

Box plots allow us to better understand the runtime performance of the two local search algorithms. These plots show the average runtime, the inner quartile range of the runtime, and upper and lower bounds indicating if outliers exist. These plots are created by binning intermediate runtimes based on the relative error of intermediate solutions. The bin centers shown correspond to the contours shown later in section 5.2. Each bin is centered at the indicated values and has a width of 0.05% relative error.

Figures 1 and 2 show the results of local search 1 and 2 when applied to the graph power, a network representing the topology of the western states power grid. While local search 1 takes longer to execute than local search 2 at the same quality, local search 1 is able to reliably find solutions of slightly higher quality than local search 2. Table 2 provides specific values at a relative error of 1% for comparision. Figures 3 and 4 show the results of local search 1 and 2 when aplied to the graph star2 wich is described as star-like structures of different graphs with different types. For this graph, local search 2 outperforms local search 1 and finds solutions of higher quality in far less time. Table 3 provides specific values at a relative error of 4.0% and 3.8% for local search 1 and local search 2 respectively for comparision.

Table 1. **Algorithm Performance.** Time, VC size (VC value), and relative error of BnB, Approximation, and two local search algorithms with 11 graph instances. We set two time entries for the BnB algorithm as the BnB algorithm generally does not run to completion for a cutoff time of 600 seconds, at which point it returns the best solution found so far. If that is the case, we report the final time, and VC value, and relative error values output after the cutoff time on the column *BnB (By 600s)*. In parentheses, we report the time in the trace at which the current best solution was found.

Algorithm	BnB (By 600s)	Approx	LS1	LS2
star				
Time(s)	600 (36.3127)	22.88	29.26	0.02
VC Value	7366	10096	7376	7005
RelErr(%)	6.7226	46.2764	6.8676	1.4923
star2				
Time(s)	600 (34.1273)	15.53	29.2	8.36
VC Value	4677	5900	4568	4574
RelErr(%)	2.9722	29.8987	0.5724	0.7045
football				
Time(s)	600 (1.643)	0.01	0.03	0.01
VC Value	95	102	94	94
RelErr(%)	1.0638	8.5106	0.0	0.0
jazz				
Time(s)	600 (0.0313)	0.03	0.02	0.01
VC Value	159	174	158	158
RelErr(%)	0.6329	10.1266	0.0	0.0
email				
Time(s)	600 (0.4156)	0.3	4.11	0.24
VC Value	605	728	595	595
RelErr(%)	1.8518	22.5589	0.1684	0.1684
delaunay_n10				
Time(s)	600 (0.3958)	0.3	6.13	0.22
VC Value	740	858	708	716
RelErr(%)	5.2631	22.0484	0.7112	1.8492
netscience				
Time(s)	600 (0.5733)	0.33	0.31	0.01
VC Value	899	1144	899	899
RelErr(%)	0.0	27.2525	0.0	0.0
karate				
Time(s)	0.0017 (0.0003)	0.0003	0.0000	0.0003
VC Value	14	16	14	14
RelErr(%)	0.0	14.2857	0.0	0.0
as-22july06				
Time(s)	600 (31.8089)	12.44	9.65	1.91
VC Value	3312	4956	3307	3314
RelErr(%)	0.2724	50.0454	0.1211	0.333
hep-th				
Time(s)	600 (12.4653)	6.4	8.82	3.19
VC Value	3947	5200	3930	3935
RelErr(%)	0.5348	32.4503	0.1019	0.2292
power				
Time(s)	600 (5.7627)	2.25	2.45	2.45
VC Value	2273	3112	2214	2227
RelErr(%)	3.1774	41.2619	0.4993	1.0894

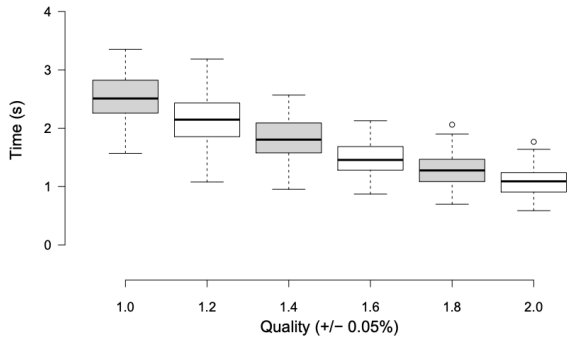

Fig. 2. Local Search 2 on power.graph

Table 2. power.graph Relative Error Comparision (seconds)

Algorithm	Lower Whisker	1st Quartile	Median	3rd Quartile	Upper Whisker
LS1 (1.0%)	3.45	3.58	3.67	3.79	4.08
LS2 (1.0%)	1.57	2.26	2.51	2.82	3.35

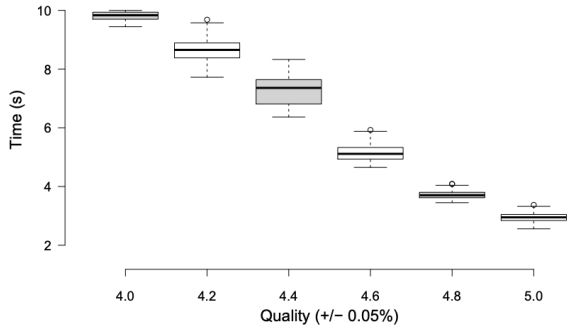

Fig. 3. Local Search 1 on star2.graph

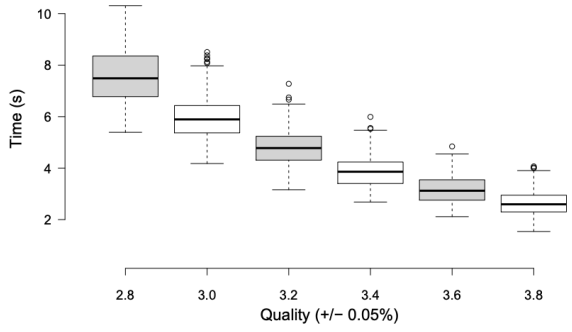

Fig. 4. Local Search 2 on star2.graph

Table 3. star.graph Relative Error Comparision (seconds)

Algorithm	Lower Whisker	1st Quartile	Median	3rd Quartile	Upper Whisker
LS1 (4.0%)	9.44	9.70	9.83	9.94	10.00
LS2 (3.8%)	1.54	2.30	2.59	2.95	3.91

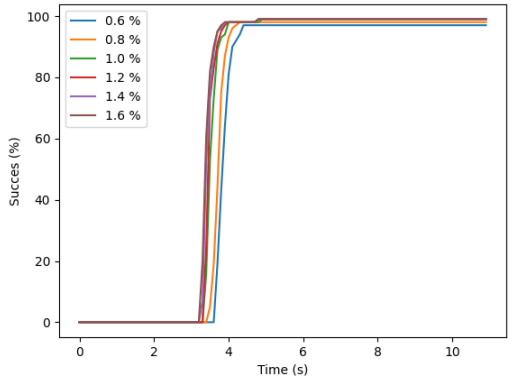

Fig. 5. QRTD for Local Search 1 on power.graph

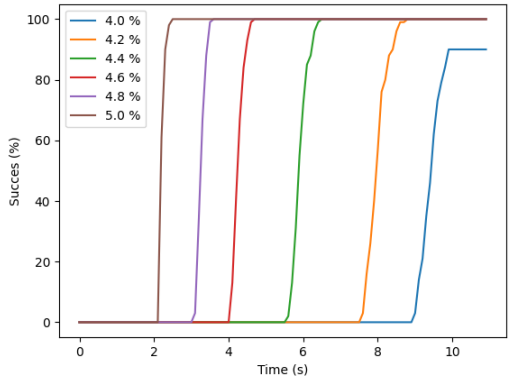

Fig. 6. QRTD Local Search 1 on star2.graph

5.3 Qualified Runtime Distributions

Qualfied runtime distributions help paint a picture of the percentage of runs that achieve a fixed quality q with varying run time for the randomized local search algorithms. The fixed quality q is then changed in order to observe the difference in behavior. We conduct this test for both the LS1 and LS2 algorithms on the *power* and *star2* graphs. The performance for the LS1 algorithm on the *power* and *star2* graphs are provided in figures 9 and 10.

The results for LS1 on power.graph indicate very little variance indicate similar solution time for correct results with varying quality q. The cutoff time to achieve $0.5 - 1\%$ this quality is about 4 seconds. For star2.graph, the cutoff time to achieve a required quality shows a drastic difference. The solution qualities are also not as optimal as power.graph, with about $4 - 5\%$ solution quality in similar time.

The performance for the LS2 algorithm on the *power* and *star2* graphs are provided in figures 11 and 12.

For power.graph, LS2 also shows little variance in solution quality with varying run times. The observed solution quality is in the range of $1 - 2\%$, with a typical runtime of about 1 seconds. LS2 shows

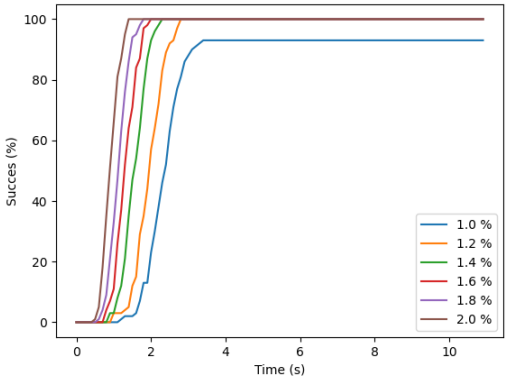

Fig. 7. QRTD for Local Search 2 on power.graph

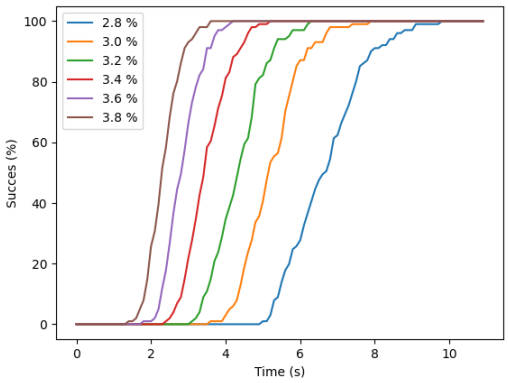

Fig. 8. QRTD Local Search 2 on star2.graph

lesser deviation in solution time with varying qualities as compared to LS1 on star2.graph. It achieves better quality solutions ($3 - 3.5\%$) with a better run time than LS1.

5.4 Solution Quality Distributions

Solution quality distributions help decipher the varying solution quality for a fixed cutoff time t. The fixed cutoff time t is then changed in order to observe the difference in behavior. We conduct this test for both the LS1 and LS2 algorithms on the *power* and *star2* graphs. The performance for the LS1 algorithm on the *power* and *star2* graphs are provided in figures 9 and 10.

The results for LS1 on power.graph indicate a sharp jump in solution quality with varying cutoff time t. The algorithm achieves near perfect solutions in about 4 seconds. For star2.graph, the cutoff time has a marginal effect on the solution quality. A solution of about $5 - 6\%$ is observed for cutoff times in the range of $1 - 5$ seconds.

The performance for the LS2 algorithm on the *power* and *star2* graphs are provided in figures 11 and 12.

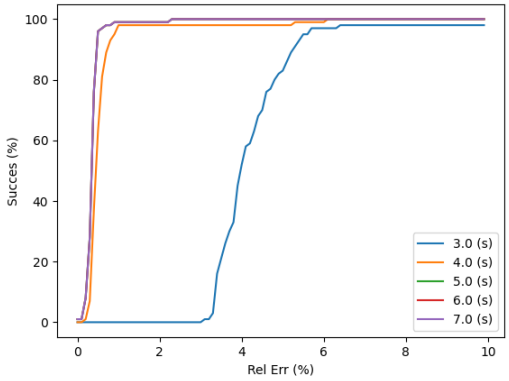

Fig. 9. SQD for Local Search 1 on power.graph

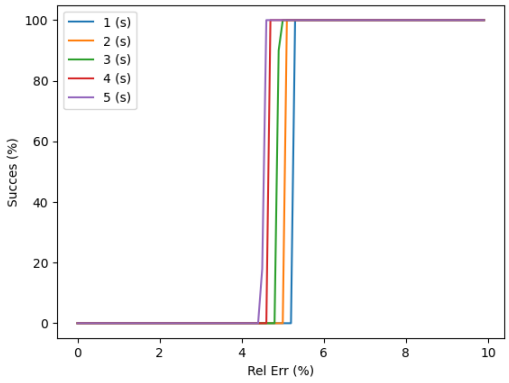

Fig. 10. SQD for Local Search 1 on star2.graph

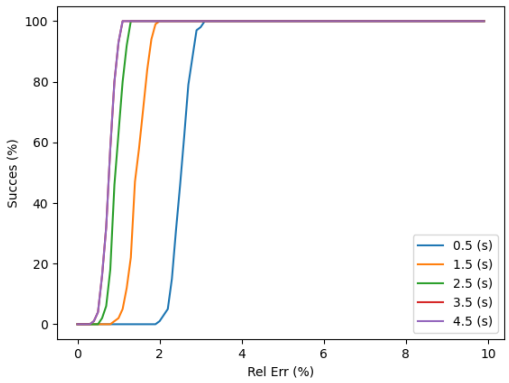

Fig. 11. SQD for Local Search 2 on power.graph

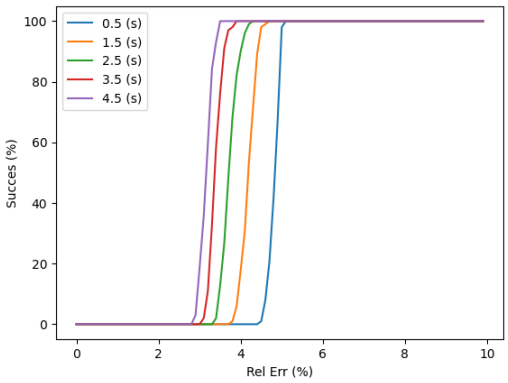

Fig. 12. SQD Local Search 2 on star2.graph

For power.graph, LS2 also shows a jump in solution quality with varying cutoff times. The solution qualities are slightly worse than LS1 (0.5%), however, the cutoff times required are lower ($3-4$ seconds). LS2 shows little deviation in solution quality with varying cutoff times on star2.graph. It achieves slightly better solution qualities($3-4\%$) at similar cutoff times.

6 DISCUSSION

Time complexity. The Table 1 shows a great gap in the time efficiency of the exact BnB algorithm and the other three as the BnB algorithm runs longer than 600 seconds for all graph instances except for the *karate* graph. For the approximation algorithm, which is expected to have the time complexity of $O(|E|\min(|V|,|E|))$, indeed shows the great time scalability. However, our heuristics for selecting the edge to be removed brings the overhead to compute the degree of edges by summing up the degrees of the edge's two endpoints. The overhead is insignificant with the small graph but would be problematic as the graph grows bigger and bigger. Hence, even though the heuristics align with the intuition to get smaller vertex cover, random edge selection that does not have any overhead could be a better choice for time-sensitive situations or large graphs. The Local search algorithms are quicker than the approximate algorithm as their time complexities are essentially $O(|E|)$. Since LS2 uses information from previous iterations to minimize the work involved in current iterations, later iterations take $O(\Delta)$ work instead of $O(E)$ (Δ is the maximum degree of any vertex in the graph) We observe the advantage of using this information as it tends to outperform LS1 for larger graphs.

Solution quality. The BnB algorithm is expected to return the optimal solution; however, due to its unscalable exponential time complexity, its solutions generated in 600 seconds are not always optimal. It gives the optimal solution for the *karate* graph that it ran to completion.

The approximation algorithm's relative error is no more than 50.05% for the given graph instances. This result is consistent with the theoretical approximation ratio of 2; according to the theoretical ratio, the relative error should be no more than 100%. However, the relative error is often greater than 20%, which might be an issue for some quality-sensitive applications. Better heuristics to choose the edge to be removed can be crafted depending on the size or structure of the graph.

Both Local search algorithms tend to provide $1-5\%$ relative error in under 5 seconds. Solution quality for LS1 shows slight improvement over LS2 albeit at a much higher cost in terms of solution time. LS2 solution quality can be further improved by increasing the number of random restarts (currently set at 1000) at the cost of increasing solution time.

7 CONCLUSION

The results of these analyses suggest that local search algorithms outperform approximation algorithms both with respect to run time and solution quality. While the exact branch and bound algorithm can always find an exact solution, it takes orders of magnitude longer than local search to find them. We ultimately conclude that unless an exact solution for a relatively small graph is needed, local search should be used to find approximate vertex covers with acceptable quality in a reasonable amount of time

REFERENCES

[1] Faisal Abu-khzam, Rebecca Collins, Michael Fellows, Michael Langston, W. Suters, and Christopher Symons. 2004. Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments. 62–69.
[2] Takuya Akiba and Yoichi Iwata. 2016. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science* 609 (2016), 211–225.
[3] Diogo V Andrade, Mauricio GC Resende, and Renato F Werneck. 2012. Fast local search for the maximum independent set problem. *Journal of Heuristics* 18, 4 (2012), 525–547.
[4] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. 2018. *Benchmarking for Graph Clustering and Partitioning.* Springer New York, New York, NY, 161–171. https://doi.org/10.1007/978-1-4939-7131-2_23
[5] S Balaji, V Swaminathan, and K Kannan. 2010. An effective algorithm for minimum weighted vertex cover problem. *Int. J. Comput. Math. Sci* 4, 1 (2010), 34–38.
[6] Reuven Bar-Yehuda and Shimon Even. 1981. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms* 2, 2 (1981), 198–203.
[7] Reuven Bar-Yehuda and Shimon Even. 1983. *A local-ratio theorm for approximating the weighted vertex cover problem.* Technical Report. Computer Science Department, Technion.
[8] S. Cai, K. Su, C. Luo, and A. Sattar. 2013. NuMVC: An Efficient Local Search Algorithm for Minimum Vertex Cover. *Journal of Artificial Intelligence Research* 46 (apr 2013), 687–716. https://doi.org/10.1613/jair.3907
[9] François Delbot and Christian Laforest. 2010. Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover Problem. *ACM J. Exp. Algorithmics* 15, Article 1.4 (nov 2010), 27 pages. https://doi.org/10.1145/1671970.1865971
[10] Ayaan Hossain, Eriberto Lopez, Sean M Halper, Daniel P Cetnar, Alexander C Reis, Devin Strickland, Eric Klavins, and Howard M Salis. 2020. Automated design of thousands of nonrepetitive parts for engineering stable genetic systems. *Nature biotechnology* 38, 12 (2020), 1466–1475.
[11] Yoichi Iwata, Keigo Oka, and Yuichi Yoshida. 2014. *Linear-Time FPT Algorithms via Network Flow.* 1749–1761. https://doi.org/10.1137/1.9781611973402.127 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611973402.127
[12] Ansamma John and M. Wilscy. 2015. Vertex Cover Algorithm Based Multi-document Summarization Using Information Content of Sentences. *Procedia Computer Science* 46 (2015), 285–291. https://doi.org/10.1016/j.procs.2015.02.022 Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace Island Resort, Kochi, India.
[13] Raka Jovanovic and Milan Tuba. 2011. An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem. *Applied Soft Computing* 11, 8 (2011), 5360–5366.
[14] George Karakostas. 2005. A better approximation ratio for the vertex cover problem. In *International Colloquium on Automata, Languages, and Programming.* Springer, 1043–1050.

[15] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. *Complexity of Computer Computations* (1972), 85–103.
[16] Jon Kleinberg and Eva Tardos. 2006. *Algorithm design.* Pearson Education India.
[17] Stefan Kratsch and Frank Neumann. 2009. Fixed-Parameter Evolutionary Algorithms and the Vertex Cover Problem *(GECCO '09)*. Association for Computing Machinery, New York, NY, USA, 293–300. https://doi.org/10.1145/1569901.1569943
[18] Burkhard Monien and Ewald Speckenmeyer. 1985. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica* 22, 1 (1985), 115–123.
[19] Masaru Nakajima, Hong Xu, and Sven Koenig TK Satish Kumar. 2018. Towards understanding the min-sum message passing algorithm for the minimum weighted vertex cover problem: An analytical approach. In *the International Symposium on Artificial Intelligence and Mathematics*.
[20] Rolf Niedermeier and Peter Rossmanith. 2003. On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms* 47, 2 (2003), 63–77. https://doi.org/10.1016/S0196-6774(03)00005-1
[21] Alexander C Reis, Sean M Halper, Grace E Vezeau, Daniel P Cetnar, Ayaan Hossain, Phillip R Clauer, and Howard M Salis. 2019. Simultaneous repression of multiple bacterial genes using nonrepetitive extra-long sgRNA arrays. *Nature biotechnology* 37, 11 (2019), 1294–1301.
[22] Shyong Jian Shyu, Peng-Yeng Yin, and Bertrand MT Lin. 2004. An ant colony optimization algorithm for the minimum weight vertex cover problem. *Annals of Operations Research* 131, 1 (2004), 283–304.
[23] Changbing Tang, Ang Li, and Xiang Li. 2017. Asymmetric game: A silver bullet to weighted vertex cover of networks. *IEEE transactions on cybernetics* 48, 10 (2017), 2994–3005.
[24] Limin Wang, Wenxue Du, Zhao Zhang, and Xiaoyan Zhang. 2017. A PTAS for minimum weighted connected vertex cover P_3 problem in 3-dimensional wireless sensor networks. *Journal of Combinatorial Optimization* 33, 1 (2017), 106–122.
[25] Luzhi Wang, Shuli Hu, Mingyang Li, and Junping Zhou. 2019. An exact algorithm for minimum vertex cover problem. *Mathematics* 7, 7 (2019), 603.
[26] Xinshun Xu and Jun Ma. 2006. An efficient simulated annealing algorithm for the minimum vertex cover problem. *Neurocomputing* 69, 7-9 (2006), 913–916.
[27] Haonan Zhong. 2017. Engineering an Efficient Branch-and-Reduce Algorithm for the Minimum Vertex Cover Problem.