# Efficient Vertex Cover Algorithms

Weiwei Kong
Georgia Institute of Technology
North Avenue NW
Atlanta, Georgia 30332
wkong37@gatech.edu

Manasa Kadiri
Georgia Institute of Technology
North Avenue NW
Atlanta, Georgia 30332
manasa.kadiri@gatech.edu

Haowen Zhang
Georgia Institute of Technology
North Avenue NW
Atlanta, Georgia 30332
hwzhang@gatech.edu

**ACM Reference format:**
Weiwei Kong, Manasa Kadiri, and Haowen Zhang. 2017. Efficient Vertex Cover Algorithms. In *Proceedings of CSE 6140, Atlanta, Georgia USA, December 2017,* 8 pages.
DOI: 10.475/777_7

## 1 INTRODUCTION

The minimum vertex cover (MVC) problem is an $\mathcal{NP}$-complete problem that involves covering every edge in a graph using the fewest number of vertices possible. Since it is $\mathcal{NP}$-complete, an efficient algorithm does not exist for solving the MVC problem. In order to solve the MVC problem in practice, we must consider several strategies. These strategies present us with a trade-off between run-time and solution quality. The algorithm design problem that we must consider is how to best balance these two criteria.

In this paper, we analyze four strategies for solving the MVC problem. First, a branch and bound algorithm was implemented to compute a nearly optimal vertex cover for each data set. Since computing exact solutions is computationally expensive, a constructive heuristic was implemented to generate quick solutions. This implementation guarantees a solution that is at worst, twice the size of the optimal solution. Finally, two local search algorithms were implemented that start from a valid vertex cover and progressively move to better vertex covers. Following the implementation, we conduct empirical analysis of algorithm performances on dataset provided and discuss the trade offs between accuracy,speed etc, across different algorithms. The evaluation metric used was the relative error between the size of the final vertex cover produced, $S$, and the size of the minimal vertex cover, OPT, which is calculated as ($S$ - OPT)/OPT.

After running the algorithms on several graphs, we determined that branch and bound is only appropriate for the smallest instances of the MVC problem. We have also presented a DFS-based approximation algorithm which has the best run times, but provides poorer solutions compared to local search algorithms. Of the local search algorithms, the Fast VC local search provided better solutions than Simulated Annealing.

## 2 PROBLEM DEFINITION

An undirected graph $G = (V, E)$ consists of a set of vertices $V$ and a set of edges $E$. Each edge in E consists of a pair of vertices $u$ and $v \in V$. A pair of vertices are neighbors if they share an edge. $N(v)$ is the set of all vertices that are neighbors of v. Given an undirected, unweighted graph $G = (V, E)$, a vertex cover is a set $C \subset V$ such that $\forall (u, v) \in E : (u \in C) \vee (v \in C)$. The MVC is the set C such that $|C|$ is minimized.

*CSE 6140, Atlanta, Georgia USA*
2017. 123-4567-24-567/77/77…$15.00
DOI: 10.475/777_7

## 3 RELATED WORK

The MVC problem is a well known $\mathcal{NP}$-complete problem that has been studied extensively in computer science. Minimum Vertex Cover was proved to be NP-complete by Karp [1]. Therefore, much effort has been trying to develop approximation algorithms. The simple greedy algorithm can lead to an approximation ratio as bad as $\log(n)$ [2]. However, there are several algorithms with an approximation ratio of two [3]. Delbot and Laforest [4] analyze the average-case instead of worst-case performance for six 2-approximation algorithms and conduct extensive experiments on common graphs. Experimental studies on heuristic approaches on the vertex cover problem include[5].

Whether there exists a better than 2-factor approximation for the minimum vertex cover problem is one of the major open problems in the research area of approximation algorithms. It is conjectured that it is impossible to achieve an approximation ration lower than 2 [14]. Currently, there does not exist a better than 1.1666-approximation algorithm for this problem.

In this report, we empirically compare the several algorithms, including branch-and-bound, approximation algorithms, and two local search algorithms.

## 4 ALGORITHMS

### 4.1 Branch and Bound

**Description**:

The implemented branch and bound algorithm first leverages an approximation algorithm to create an initial incumbent solution and to produce a very crude upper bound. At each node in the branch and bound scheme, we are branching on whether or not a candidate vertex should be included into the candidate vertex cover. The order of candidate vertices that we consider is based on their descending degree, e.g. a vertex with 10 connections will be considered before a vertex with 1 connection. In addition rounding to an LP relaxation of the MVC problem with previously branched constraints to get a candidate vertex cover.

In particular, at each step of the branch and bound algorithm, we are solving a linear program (LP) of the form

$$
\begin{aligned}
(P_C) \qquad & \min \sum_{v \in V} x_v \\
& \text{s.t. } x_u + x_v \geq 1, \quad \forall \{u, v\} \in E \\
& \quad 0 \leq x_v \leq 1, \qquad \forall v \in V \\
& \quad x \in C
\end{aligned}
$$

where $C$ is the constraint set on $x$ given by whether or not we include a particular vertex in our candidate solution, e.g. $C = \{x : x_{v_1} = 1, x_{v_2} = 0, x_{v_3} = 1\}$. If $(P_C)$ is infeasible, we use the convention that $\text{OPT}(P_C) = |V| + 1$. In our implementation, Gurobi was used as our choice of an LP solver.

The choice of whether or not we are considering including a vertex $v$ into the candidate solution is dependent on the LP optimal value associated with that choice. For example, if $\text{OPT}(P_{C_1}) < \text{OPT}(P_{C_0})$ where $C_1$ has the constraint that we include vertex $v$, i.e. $x_v = 1$, and $C_0$ has the constraint that we exclude vertex $v$, i.e. $x_v = 0$, then we first branch on the choice of $x_v = 1$ before the choice of $x_v = 0$.

If a candidate solution is found to be feasible, i.e it is a vertex cover, and is smaller than our incumbent solution, we update our incumbent solution to be the feasible candidate solution and go up one level in the tree. Otherwise, we prune that node and go up one level in the tree. Candidate solutions may be found from the full constraint set itself or from rounding the LP solution vector that is generated as each node to form a list of vertices for the vertex cover.

The branch and bound algorithm returns the minimum vertex cover as the incumbent solution when all branches have been searched or appropriately pruned.

**Pseudocode**:

**Algorithm 1** branchAndBound

**Input:** Graph $G = (V, E)$, cutoff time $T$
**Output:** Vertex Cover of G

```
Construct incumbent solution as IS from an approximation
Set LB = 0, UB = |V|, C = ∅ and a candidate solution as CS = ∅
Sort V in descending order of vertex degree
Call the subroutine BnBSub(LB,UB,IS,CS,V,E,C)
return IS
```

**Algorithm 2** BnBSub(LB,UB,IS,CS,$V$,$E$,$C$)

```
if CS is a vertex cover and |CS| < LB then
    Set IS = CS and LB = |CS|
    return
else if CS is a vertex cover and |CS| ≥ LB then
    return
else
    Initialize CS⁻ = CS and CS⁺ = CS
    Let C_0 be C ∪ {x_v = 0}
    Let C_1 be C ∪ {x_v = 1}
    Set C⁻ = argmin_{c∈{C_0,C_1}} OPT(P_c)
    Set C⁺ = argmax_{c∈{C_0,C_1}} OPT(P_c)
    Set CS⁻ to be CS∪{v} if C⁻ = C_1
    Set CS⁺ to be CS∪{v} if C⁺ = C_1
    Round the LP solutions in P_{C_0} and P_{C_1}
    Use these approximations if they are smaller than IS
    Call BnBSub(LB, UB, IS, CS⁺, V\{v}, E, C⁺)
    Call BnBSub(LB, UB, IS, CS⁻, V\{v}, E, C⁻)
    return
return IS
```

Here, we use the convention that if $(P_C)$ is infeasible, then $\text{OPT}(P_C)$ is equal to the number of vertices in the original graph plus one.

**Approximation Guarantee**:
As the branch and bound algorithm, in the worst case scenario, will do a complete enumeration of all of the possible sets of vertices, given enough time the algorithm will always find the optimal solution.

**Time and Space Complexity**:
In our implementation of the branch and bound algorithm, we need to keep a persistent record of the edges $E$ of input graph $G(V, E)$ as well as candidate and incumbent solutions which are of size at most $|V|$. As long as we do not continually copy information every time we branch, the space complexity of this algorithm is

$$S = O(|V| + |E|).$$

For the time complexity of branch and bound, the only known lower bound for this method is the bound generated by full enumeration of the possible vertex sets, which is $O(2^{|V|})$. Given that solving a linear optimization problem is polynomial in the size of the inputs, i.e. $|E|, |V|$, the runtime of this branch and bound method can be reduced to

$$T = O(2^{|V|}c(|V|, |E|))$$

where $\mathcal{P}(\cdot, \cdot)$ is some polynomial function of its inputs that is dependent on the linear programming solver used. As the actual LP solver used in our branch and bound method is controlled through Gurobi, the complexity can be anywhere between polynomial and exponential time.

**Strengths and Weaknesses**:
The main strength of the branch and bound method is that it is guaranteed to find a minimal vertex cover at termination, unlike the approximation and local search algorithms. However, this guarantee will come at the cost of runtime which is exponential in the worst case and is highly dependent on the approximation used at each node in the branch and bound tree. It is also very dependent on the pivoting rule, which can adversely affect the practical use of the branch and bound method.

## 4.2 Approximation Algorithm

We choose the *depth first search (DFS) algorithm* in [6] as our approximation algorithm. It first finds a DFS spanning tree on the graph. Then leaf nodes are removed and the starting vertex is included to generate a vertex cover. In [7], this algorithm was proven to have a worst-case approximation ratio of 2. We have slightly modified the algorithm to make it adaptable to a disconnected graph. The pseudocode is as follows.

**Algorithm 3** Approximation algorithm

**Input:** graph $G = (V, E)$, cutoff time
**Output:** Vertex Cover $VC$ of G

```
for all connected components in G do
    Find a DFS spanning tree, T, from a start vertex s
    Let I(T) be the set of nonleaves vertices of T
    VC ← VC ∪ I(T) ∪ {s}
return VC
```

The time complexity is the same as it of DFS, which is $O(|V|+|E|)$. The space complexity is $O(|V|)$ for a stack in DFS. The advantage of this approximation algorithm is that it can return a solution in a very short time. But the solution is usually not so good.

## 4.3 Local Search 1

This local search algorithm, using simulated annealing, has its own unique methodology in finding the optimal solution for the minimum vertex cover problem. The whole concept of simulated annealing revolves around using a probabilistic technique to find an optimal solution for the problem, in which a cost-decreasing or cost-increasing solution will be chosen based on a function.

For the minimum vertex cover problem it is well know that a vertex which has a larger degree that the other vertices will be put into the cover with higher probability because such a vertex can cover more edges. First, the cost function for a given candidate is defined as as follows:

$$Cost = \Sigma_{i=1}^{n} v_i + \Sigma_{i=1}^{n}\Sigma_{j=1}^{n} v_i \wedge v_j$$

where the state of the $i$th vertex can be determined by

$$v_i = \begin{cases} 1 \ if \ the \ vertex \ i \ is \ in \ the \ cover \\ 0 \ otherwise \end{cases}$$

Notice that the cost of a given candidate gives the vertex cover of size equal to the cost. This is because we can always construct a vertex cover based on the current cover set by include a vertex from each of the uncovered edges

We define the acceptance criteria function as the following :

$$p = \begin{cases} e^{-\frac{\Delta(1-Deg(v_i)}{T}} & v_i = 1 \\ e^{-\frac{\Delta(1+Deg(v_i)}{T}} & v_i = 0 \end{cases}$$

$$Deg(v_i) = \frac{Degree(i)}{EdgeNum}$$

where $Degree(i)$ is the degree of vertex i, its value is equal to the number of edges linking to the ith vertex, and EdgeNum is a constant, equal to total number of edges in a given graph.

For the minimum vertex cover problem, vertices with large degrees will be put into the cover with higher probability compared to vertices with small degrees because such vertices can cover more edges. First we begin with an initial solution $C$. A neighbour $N$ to this solution is generated by changing the state of random vertex. If a reduction in cost is found,the current solution is replaced by the generated neighbour. Otherwise, we use Eq.1, to determine whether a solution is replaced by its neighbour. If $v_i = 1$, this means $v_i$ is originally excluded from the cover set. $p$ takes a larger value if the degree of the i$^{th}$ vertex is larger, which means $N$ will be accepted as a new solution with higher probability. That is to say, a vertex with larger degree will be selected into the cover set with higher probability. If $v_i = 0$, this means $v_i$ is originally included in the cover set. $p$ takes a smaller value if the degree of the i$^{th}$ vertex is larger, which means $N$ will be accepted as a new solution with lower probability. That is to say, a vertex with a larger degree will be removed from the cover set with lower probability.

**Algorithm 4** Local Search : Simulated Annealing

**Input:**graph $G = (V, E)$, cutoff time
**Output**Vertex Cover of G

```
Construct initial vertex cover current
current = approx-vc(Graph)
while elapsed < cutoff do
    T = schedule(t);
    nextSol = Random-Successor;
    Δ = cost(nextSol) - cost(current)
    if Δ < 0 then
        current = nextSol
    else if (v_i = 1) then
        current = nextSol with probability in Eq.1
    else
        current = nextSol with probability Eq 1
    end
end
return C
```

**Time and Space Complexity Analysis**

This algorithm gets an initial solution through approx-vc algorithm which goes through all the edges and vertices of the graph because we need to know the degree of each vertex. Hence the time complexity of this is O($|V| + |E|$).

For Simulated Annealing, we iterate through the while loop as long as the temperature is cooling which is a set value so it will be O($n$) iterations. Then we remove a random vertex from the current solution to find the next solution. So each time we are picking a random vertex, we will be going through every vertex which will be O($|V|$) time. In every temperature step we are going through the adjacent edges of one vertex which is O($|E|$) time. So a possible time complexity can be O($n|E|$).

The space complexity for this algorithm is determined by approx-vc algorithm which uses O($|E| + |V|$) space for storing and removing the vertices. The main simulated annealing uses only O($|V|$) space for saving the current and next solutions of the vertex cover. Overall the space complexity of the entire algorithm would be O($|V| + |E|$).

**Strengths and Weakness**

The strengths of Simulated Annealing is the ability to deal with non-linear models and noisy data. It can achieve global optimality in a short span of time and it is robust. The problems with Simulated Annealing are that where there are many local minima, the algorithm might not give great results. In addition to that, the algorithm cannot determine on its own if it has found an optimal solution.

### 4.4 Local Search 2

We will implement another local search algorithm based on [8]. Algorithm 5 is the top-level algorithm. We show how to construct our initial solution in Algorithm 6 and how to choose the best neighborhood in Algorithm 7.

The main idea of *FastVC* is as follows. First, we construct a vertex cover $C$ by *ConstructVC*. Each iteration we remove a vertex $u \in C$ by *ChooseRmVertex*. Then a random uncovered edge is picked and the endpoint with the greatest *gain* is added to $C$.

The two phases in *ConstructVC* are as follows:

(1) Extending phase: if an edge is uncovered, the endpoint with higher degree of the edge is added into $C$.
(2) Shrinking phase: if a vertex has a *loss* value of 0, we remove it and update the *loss* values of its neighbors.

In [8], the author proved that the time complexity of *ConstructVC* is $O(|E|)$. The space complexity of *ConstructVC* is $O(|V|)$.

**Algorithm 5** Local Search: FastVC

**Input:**graph $G = (V, E)$ and cutoff time
**Output:** Vertex Cover of G

$C \leftarrow ConstructVC()$;
$gain(v) \leftarrow 0$ for each vertex $v \notin C$;
**while** elapsed time < cutoff **do**
  **if** $C$ covers all edges **then**
    $C^* \leftarrow C$;
    remove a vertex with minimum *loss* from $C$;
    continue;
  **end**
  $u \leftarrow ChooseRmVertex(C)$;
  $C \leftarrow C \setminus \{u\}$;
  $e \leftarrow$ a random uncovered edge;
  $v \leftarrow$ the endpoint of $e$ with greater *gain*, breaking ties in favor of the older one;
  $C \leftarrow C \cup \{v\}$;
**end**
**return** $C^*$

**Algorithm 6** ConstructVC

**Input:**graph $G = (V, E)$ and cutoff time
**Output:** Vertex Cover of G

$C \leftarrow \emptyset$;
//extend $C$ to cover all edges
**for** $e \in E$ **do**;
  **if** $e$ is not covered **then**
    add the endpoint of $e$ with higher degree into $C$;
  **end**
//calculate *loss* of vertices in $C$
$loss(v) \leftarrow 0$ for each $v \in C$;
**for** $e \in E$ **do**;
  **if** only one endpoint of $e$ belongs to C **then**
    for the endpoint $v \in C$, $loss(v)++$;
  **end**
//remove redundant vertices
**for** $v \in C$ **do**;
  **if** $loss(v) = 0$ **then**
    $C \leftarrow C \setminus \{v\}$, update *loss* of vertices in $Neighbor(v)$;
  **end**
**return** $C$

**Algorithm 7** ChooseRmVertex

**Input:**A set $S$, a parameter $k$, a comparison function $f$
**Output:** an element of $S$

$best \leftarrow$ a random element from $S$;
**for** $i \leftarrow 1$ to $k - 1$ **do**;
  $r \leftarrow$ a random element from $S$;
  **if** $f(r) < f(best)$ **then**
    $best \leftarrow r$;
  **end**
**return** $best$

In *ChooseRmVertex*, the comparison function $f$ is simply the comparison on *loss* value. Since $k$ is a constant, the time comlexity of *ChooseRmVertex* is $O(1)$. The space complexity of *ChooseRmVertex* is also $O(1)$. Therefore, the time complexity of *FastVC* on each iteration is $O(|E|)$. The space complexity of *FastVC* is $O(|V| + |E|)$ for storing the vertex cover and uncovered edges in each iteration.

In [8], $k$ is set to 50. In our implementation, we also choose 50 as the value of $k$.

**Strengths and Weaknesses**:

*FastVC* takes use of several heuristics and has a low time complexity on each iteration, which enables it to return a near optimal solution and scale well on massive graphs, which is a key strength of the algorithm. On the other hand, because it is a local search algorithm, there is no guarantee of finding a minimal vertex cover after the cutoff period, unlike the branch and bound method.

## 5 EMPIRICAL EVALUATION

### 5.1 Branch and Bound

All binaries were run on the Georgia Tech ISyE compute cluster and on machines with at least 8 GB of RAM. Processor speeds varied between the computers used in the cluster with the most common

processor being an Intel Xeon E5520 processor (2.26 GHz). The language used was C++ and the compiler used was Clang++.

The branch and bound method for each instance was run on the cluster with a cutoff time of 10 hours. Below is a table that summarizes the computational results for our implementation.

| | Branch and Bound | | | |
|---|---|---|---|---|
| Dataset | Time (s) | VC | OPT | RelErr |
| karate.graph | 0.01 | 14 | 14 | 0.0000 |
| football.graph (*) | 27132.28 | 94 | 94 | 0.0000 |
| jazz.graph (**) | 19765.22 | 159 | 158 | 0.0063 |
| netscience.graph | 18.37 | 899 | 899 | 0.0000 |
| delaunay_n10.graph | 15.43 | 726 | 703 | 0.0327 |
| email.graph | 14.64 | 595 | 594 | 0.0017 |
| power.graph | 70.21 | 2205 | 2203 | 0.0009 |
| hep-th.graph | 515.70 | 3928 | 3926 | 0.0005 |
| as-22july06.graph | 2337.25 | 3303 | 3303 | 0.0000 |
| star.graph | 15414.26 | 7507 | 6902 | 0.0877 |
| star2.graph | 2098.37 | 4542 | 4542 | 0.0000 |

(*) Solution sizes of 96 and 95 were respectively found after 19 and 407 seconds.
(**) A solution size of 160 was found after 2 seconds.

## 5.2 Approximation algorithm

We tested our approximation algorithm on a machine with 4 Intel Xeon CPUs (E7-8870, 2.10 GHz) and 1 TB RAM. Below is a table that shows the results for our approximation algorithm. The language used was C++ and the compiler used was g++. The cutoff time used was 1 second.

| | Approximation algorithm | | | |
|---|---|---|---|---|
| Dataset | Time (s) | VC | OPT | RelErr |
| karate.graph | 0.00 | 20 | 14 | 0.3000 |
| football.graph | 0.00 | 109 | 94 | 0.1596 |
| jazz.graph | 0.00 | 175 | 158 | 0.1076 |
| netscience.graph | 0.00 | 1019 | 899 | 0.1335 |
| delaunay_n10.graph | 0.00 | 911 | 703 | 0.2877 |
| email.graph | 0.00 | 816 | 594 | 0.3737 |
| power.graph | 0.00 | 3245 | 2203 | 0.4730 |
| hep-th.graph | 0.01 | 5166 | 3926 | 0.3158 |
| as-22july06.graph | 0.00 | 4702 | 3303 | 0.4236 |
| star.graph | 0.00 | 8679 | 6902 | 0.2574 |
| star2.graph | 0.00 | 7184 | 4542 | 0.5817 |

## 5.3 Local Search - Simulated Annealing

We tested our Local Search Algorithm on a machine with 3.1GHz Intel Core i5 processor and 16 GB RAM. Table 3.3 shows the results of our Local Search Algorithm. The language used was C++ and the compiler used was g++. I ran all the graphs for 3600 seconds with different seed values and the results are presented in the table below.

| | Local Search 1 | | | |
|---|---|---|---|---|
| Dataset | Time (s) | VC | OPT | RelErr |
| karate.graph | 0.03 | 14 | 14 | 0.0000 |
| football.graph | 1.26 | 94 | 94 | 0.0000 |
| jazz.graph | 2.16 | 158 | 158 | 0.0000 |
| netscience.graph | 209.22 | 899 | 899 | 0.0000 |
| delaunay_n10.graph | 399.12 | 712 | 703 | 0.0256 |
| email.graph | 347.597 | 596 | 595 | 0.0017 |
| power.graph | 3600 | 2328 | 2203 | 0.06 |
| hep-th.graph | 3600 | 4475 | 3926 | 0.14 |
| as-22july06.graph | 3600 | 5450 | 3303 | 0.65 |
| star.graph | 3600 | 9507 | 6902 | 0.38 |
| star2.graph | 3600 | 5858 | 4542 | 0.29 |

## 5.4 Local Search - Fast VC

We tested our local search algorithm, FastVC, on a machine with 4 Intel Xeon CPUs (E7-8870, 2.10 GHz) and 1 TB RAM. The cutoff time is set to 600 seconds and the seed is set to 0. Below is a table that shows the results for our local search algorithm, FastVC. The language used was C++ and the compiler used was g++.

| | Local search 2 | | | |
|---|---|---|---|---|
| Dataset | Time (s) | VC | OPT | RelErr |
| karate.graph | 0.00 | 14 | 14 | 0.0000 |
| football.graph | 0.00 | 94 | 94 | 0.0000 |
| jazz.graph | 0.00 | 158 | 158 | 0.0000 |
| netscience.graph | 0.00 | 899 | 899 | 0.0000 |
| delaunay_n10.graph | 0.00 | 703 | 703 | 0.0000 |
| email.graph | 0.00 | 594 | 594 | 0.0000 |
| power.graph | 0.02 | 2203 | 2203 | 0.0000 |
| hep-th.graph | 0.04 | 3926 | 3926 | 0.0000 |
| as-22july06.graph | 0.00 | 3303 | 3303 | 0.0000 |
| star.graph | 76.28 | 6911 | 6902 | 0.0013 |
| star2.graph | 0.08 | 4543 | 4542 | 0.0002 |

# 6 DISCUSSION

Using the empirical results in the previous section and the various graphs and plots in Appendix A, we can make the following observations:

- The approximation algorithm, while fast, fares poorly at finding minimal or close to minimal vertex covers.
- The branch and bound algorithm is moderately fast at finding nearly minimal vertex covers but struggles to find the minimal cover.
  - This suggests that the LP integrality gap is very persistent in many of the graph instances, even as we reduce the LP relaxation but adding more and more constraints.
- The first local search method, simulated annealing, performs better than the branch and bound method on some of the smaller instances but struggles on the larger instances.
  - The QRTDs, SQDs, and Boxplots in Appendix A tell us that this local search variant makes relatively large jumps in solution qualities over time, but may get stuck at a particular solution quality for long periods of time.

  - The seed given as input appears to have minimal effect on the solution quality over time.
- The second local search, Fast VC, performs the best out of all of the algorithms with its greedy selection method and fast implementation.
  - The QRTDs, SQDs, and Boxplots in Appendix A tell us that the progression in solution quality is influenced greatly by the seed given.

In addition, we can see that the results align very well with the expected complexities for each algorithm. That is, the branch and bound method grows exponentially with the number of vertices while the other methods are generally more time efficient as they are either linear or polynomial in the encoding of the graph files.

## 7 CONCLUSION

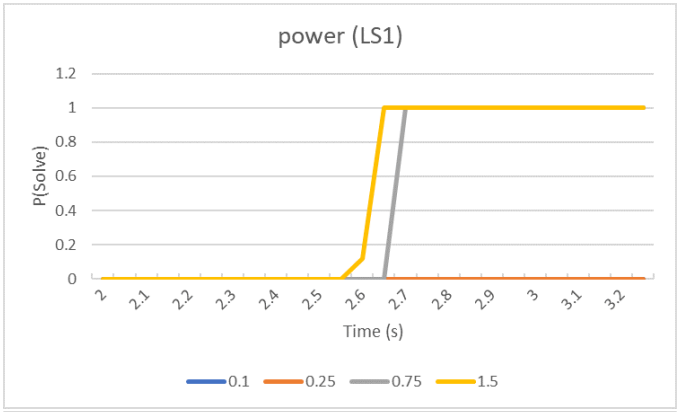
In this report, we have investigated four different algorithms on the MVC problem. First, we have implemented a branch and bound algorithm, which can always find the optimal solution with enough running time. In addition, we have presented a DFS-based approximation algorithm. Though it can return a solution efficiently, the quality of solution is much worst than that of other algorithms. Finally, two local search algorithms are implemented to find a relatively good solution within a short running time. One of them is based on simulated annealing, and the other uses several heuristics to construct the initial solution and to choose the proper neighbor.

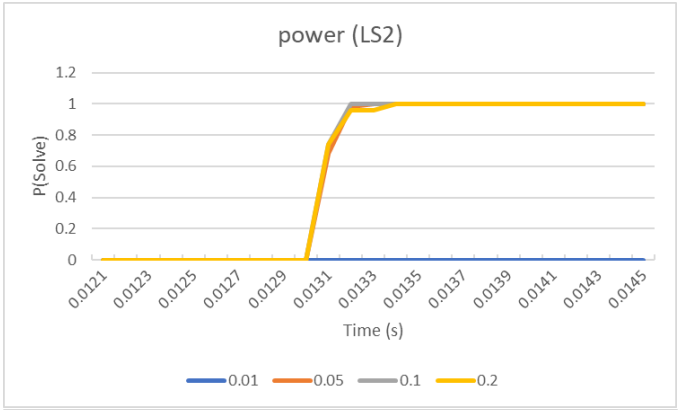
We have exhaustively tested the four algorithms on all of the instance graphs. The results show how the choice of seeds and cutoff times affects the quality of solutions for local search algorithms. Furthermore, we observed that our second local search algorithm, FastVC, finds good solutions for all the instances in a very short time thus outperforms all of the other algorithms.

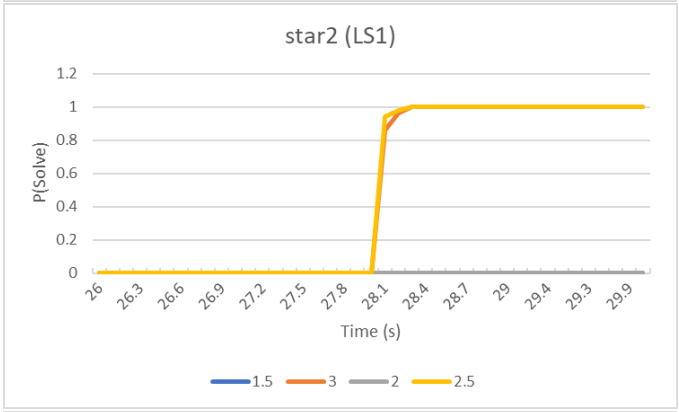
This work compares various algorithms for MVC, and provides insights on how to choose a proper algorithm under different scenarios and types of inputs.

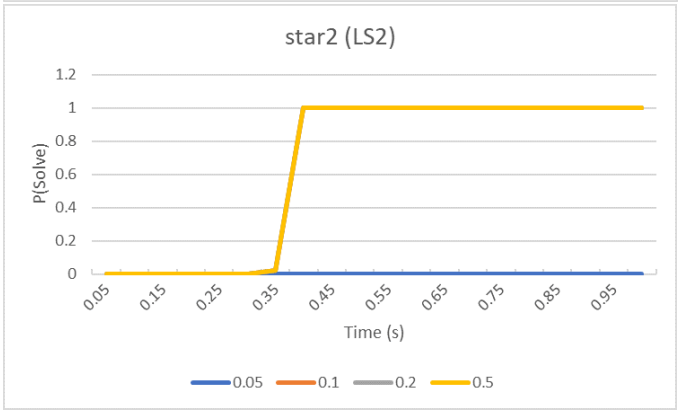
# APPENDIX A

## Qualified Runtime Distributions (QRTD)

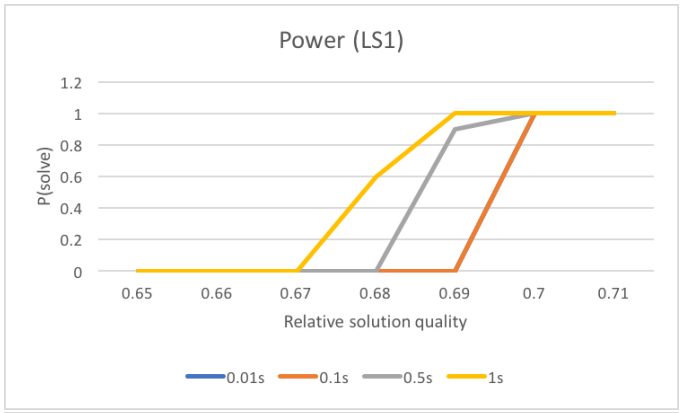


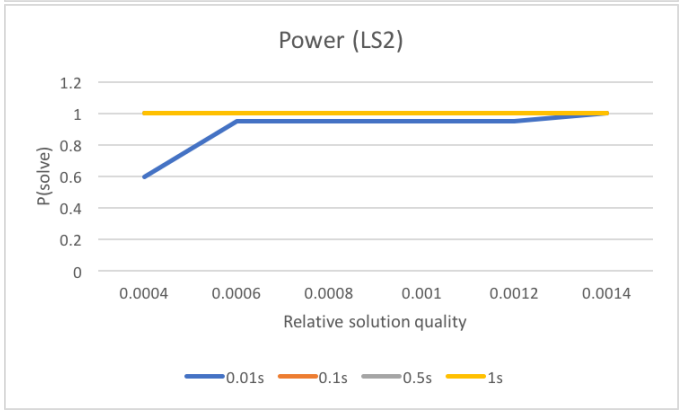



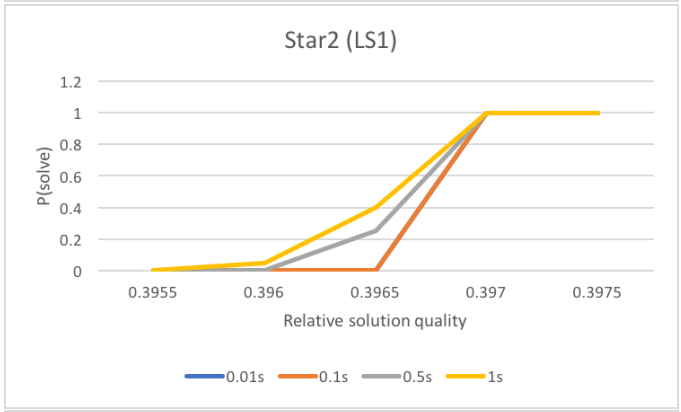



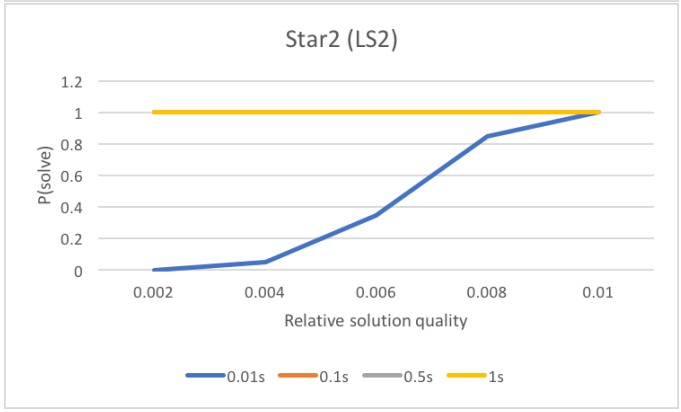




## Solution Quality Distributions (SQD)

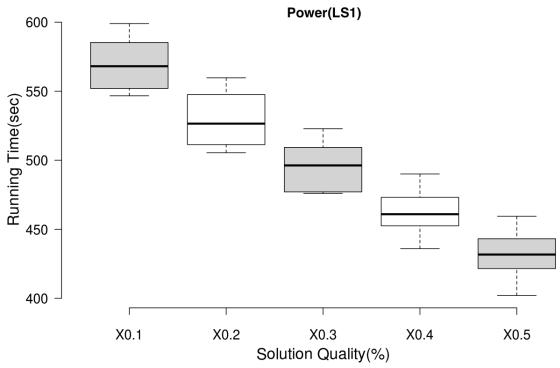


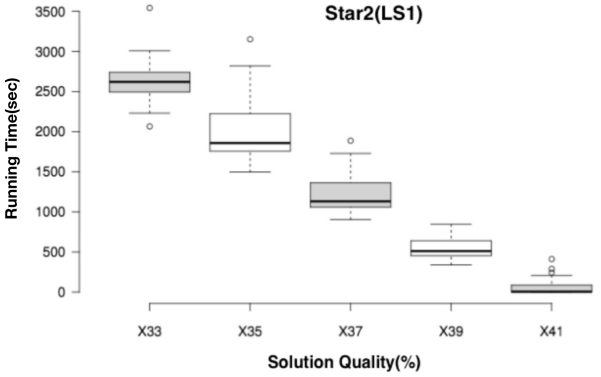



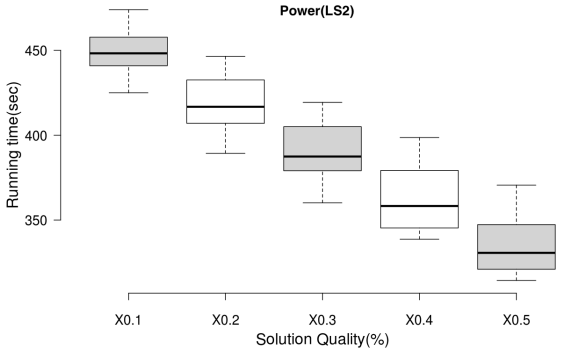



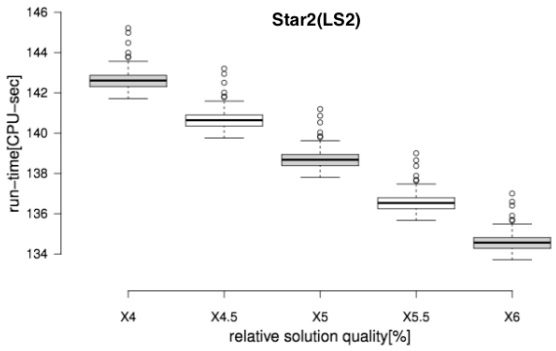



## Boxplots




## REFERENCES


[1] R. M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computation*, pages 85–103, 1972.

[2] D. Johnson. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, (9):256–278, 1974.

[3] D. Hochbaum. Approximation algorithms for combinatorial problems. *J. Computer and System Sciences*, 1996.

[4] F. Delbot and C. Laforest. Analytical and experimental comparison of six algorithms for the vertex cover problem. *Journal of Experimental Algorithmics (JEA)*, (15):1–4, 2010.

[5] P. Pardalos F. Gomes, C. Meneses and G. Viana. Experimental analysis of approximation algorithms for the vertex cover and set covering problems. *Computers and Operations Research*, (33):3520–3534, 2006.

[6] François Delbot and Christian Laforest. Analytical and experimental comparison of six algorithms for the vertex cover problem. *Journal of Experimental Algorithmics (JEA)*, 15:1–4, 2010.

[7] Carla Savage. Depth-first search and the vertex cover problem. *Information Processing Letters*, 14(5):233–235, 1982.

[8] Shaowei Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs.