# CSE 6140 Final 2022 Project Minimum Vertex Cover

Jin Guo
Georgia Institute of Technology
jguo409@gatech.edu

Tianjun Hou
Georgia Institute of Technology
thou40@gatech.edu

Ziyan Liu
Georgia Institute of Technology
zliu873@gatech.edu

Wenbin Li
Georgia Institute of Technology
wli643@gatech.edu

## 1 Introduction

This project is about the Minimum Vertex Cover (MVC) problem, which is a classical NP-complete problem serving a lot of real-world applications. In this paper, four algorithms are designed to solve the MVC problem, which are Branch-and-Bound, Approximation, Local Search-Simulated Annealing, and Local Search-Edge Weighting with Configuration Change. We apply all these models on 13 graphs and report their solutions with the time cost, as well as the theoretically optimal solutions of these graphs. Further analysis and comparison of these algorithms are also presented in this paper.

## 2 Problem Definition

For an undirected graph $G = (V, E)$ with a set of vertices $V$ and a set of edges $E$, a vertex cover is a subset of vertices $C \subseteq V$ that for each edge in graph $G$, at least one of its endpoints is in $C$. The Minimum Vertex Cover problem is to minimize the size of $C$ and find the corresponding vertex cover.

## 3 Related Work

Minimum Vertex Cover problem, as a well-known NP-hard optimization problem with great application importance in the real world, has been the object of many researches. There are mainly two ways of solving MVC problems, one is Exact Algorithms, including Branch and Bound algorithm, and the other is Heuristic Algorithms, including Approximation and Local Search Algorithms. Exact Algorithm guarantees optimal solution but may take significant long time, on the other hand, Heuristic Algorithms don't guarantee optimal solution but can provide satisfying solutions within reasonable time.

For the Branch and Bound algorithm, the key is to find a suitable lower bound. Wang's work [6] shows that a lower bound based on vertices' degree is a feasible solution, and can help to choose the most promising vertex.

For Approximation algorithm, Francois's work Analytical and experimental comparison of six algorithms for the vertex cover problem [4] argued that measurement of approximation ratio is too macroscopic and by analyzing and comparing algorithms running path graphs. We construct a greedy algorithm removing vertex and its corresponding edges from the graph until all the edges have been removed.

For Local Search Algorithm, there exist many subsidiaries, including Hill Climbing, Simulated Annealing, Iterated method and Stochastic Method; and there is a difference between First-improvement and Best-improvement. We referred to Černý's work *Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm* [7] and Kirkpatrick's work *Optimization by Simulated Annealing* [5] to build Simulated Annealing(SA) algorithm which is proved to have high efficiency. Cai and his co-authors proposed a series of iterated methods, including Edge Weighting Local Search(EWLS) and Edge Weighting Local Search with Configuration Changing(EWCC) in *Local search with edge weighting and configuration checking heuristics for minimum vertex cover* [2], and two new strategies developed thereafter: Two-stage Exchange and Edge Weighting with Forgetting in *NuMVC: An Efficient Local Search Algorithm for Minimum Vertex Cover*[1].

## 4 Algorithms

### 4.1 Branch and Bound

#### 4.1.1 Description.

Branch-and-Bound is an algorithm that guarantees the optimal solution using a search tree and decision bounds. For a Minimum Vertex Cover problem, the lower bound can be various. Here our algorithm is developed based on calculating the lower bound of each instance by using the sum of edges divided by the max vertex degree in the remaining vertex set. We will show how this works.

First, we sort all vertices in our graph in terms of degree in descending order. The vertex with a higher degree is more promising to be included in the MVC. Second, for each vertex considered, we expand the search tree with binary branches, which are including or excluding this vertex respectively. If the vertex is included, we remove all edges linked to it and recalculate the degrees of remaining nodes and resort all vertices.

Suppose we have an undirected graph $G$ and $i$ nodes have been included in the vertex set, then the lower bound for the sub-problem is:

$$LB(G) = \lceil i + \frac{|E'|}{d(v_{i+1})} \rceil \qquad (1)$$

Here, $v_{i+1}$ is the vertex with the maximum degree in the graph excluding the first $i$ nodes and edges linked to them. And $E'$ represents remaining edges that $G - \{v_1, v_2, \cdots v_i\} = (V', E')$. Then for the remaining graph, we need at least $\frac{|E'|}{d(v_{i+1})}$ vertex to cover all edges, assuming that all remaining nodes have the same degree. That's how this lower bound works.

Each time we include a node in our solution set, we will recalculate the degree of vertices for the remaining sub-graph and find the one with the highest degree. This node will be the most promising one to follow with the lower bound function above.

According to the Branch and Bound algorithm, there will be three possible results after expanding. First, if we have removed all edges after including some vertices, the vertex set will be a solution and can update the upper bound if the vertices we used are less than the current best solution. Second, if the lower bound after expanding is greater than the current upper bound, then we prune the current branch and return to the parent node. Third, if the lower bound is smaller than the upper bound, it means that there could be a better solution and thus we continue to expand the next level of branches.

#### 4.1.2 Pseudo-Code.

The Pseudo-Code is shown in Algorithm 1.

#### 4.1.3 Time and Space Complexity.

In the worst case, all vertices can be included in the decision tree, so there could be at most $2^{|V|}$ decision nodes. To expand and check each node, we need a time of $O(|V|\log(|V|) + |E|)$. So the algorithm takes $O(2^{|V|}(|V|\log(|V|) + |E|))$ in the worst case, which is an exponential time. Similarly, the worst-case space complexity is $O(2^{|V|} + |E|)$ since we need to store all the decision nodes as well as the information of the graph.

#### 4.1.4 Strength and Weakness.

The Branch and Bound algorithm guarantee finding the optimal solution if there's no time limit. It improves the greedy algorithm by avoiding meaningless cases. However, the algorithm takes exponential time in the worst case. For our project, it can only return optimal solutions for small graphs within a limited time, and it often fails for large graphs with a cutoff time.

### 4.2 Approximation

#### 4.2.1 Description.

We implemented the Maximum Degree Greedy which is referred from the paper[4] for Approximation method. Basically, this algorithm sorts the node's degree in descending order, then removes the node with the maximum degree from the initial graph and its corresponding edges, at the same time, adds the node to the set of minimum vertex cover,repeating this process until all the edges in the graph have been removed.

**Algorithm 1** Branch and Bound

```
Input: G =(V,E), cutoff time T
Output: Minimum Vertex Cover of G
  $Frontier \leftarrow (\emptyset, G)$
  $B \leftarrow (\infty, (\emptyset, G))$                    ▷ Best cost and solution
  $G' \leftarrow G$                                       ▷ subgraph
  while $Frontier \neq \emptyset$ & $Runtime \leq T$ do
    $v \leftarrow$ vertex with the maximum degree in $G'$
    $V \leftarrow V - v$
    $E \leftarrow E - e$
    Update $G'$
    Get two configurations: including and excluding $v$
    Each configuration $\leftarrow (VertexCover, G')$
    for new configurations do                          ▷ two branches
      if solution found then
        if $cost < B[0]$ then
          $B \leftarrow (cost, configuration)$         ▷ update UB
        end if
      else
        if $LB$ for $G' + cost \geq B[0]$ then
          prune                                        ▷ dead end
        else
          $Frontier$.append(configuration)
        end if
      end if
    end for
  end while
  $MVC \leftarrow B[1][0]$                             ▷ best solution
  return $MVC$
```

To be more specific, for this algorithm, we created two dictionary which are $dic_1$,$dic_2$ and one solution set $mvc$. $dic_1$ has nodes as keys and its neighbors as values, $dict2$ has nodes as keys and its degrees as values . In each iteration, after finding the maximum degree node through sorting the value for $dic2$,we first check its neighbors through $dic1$, if the neighbor is not in our solution, then we will decrease its neighbor's degrees by 1 since this edge will be removed, so the value for $dic2$ will be updated, then we will remove this node and all the corresponding edges from the initial graph, so this node will be deleted from $dic2$,and added to set $mvc$. When $len(edges) = 0$,our algorithm will stop and we get our final solution for the minimum vertex cover problem.

#### 4.2.2 Pseudo-Code.

The Pseudo-Code is shown in Algorithm 2.

#### 4.2.3 Time and Space Complexity.

This Algorithm will run the iteration at most $|V|$) times. In each iteration,the sorting step consumes at most$O(|V|log(|V|))$, the neighbor checking step consumes at most $O(|V|)$ and updating neighbors takes at most $O(|V|)$,so this checking and updating step consumes $O(|V|)^2)$ and the step to remove the max node consumes $O(1)$. The step to remove corresponding

**Algorithm 2** MaximumDegreeGreedy

```
Input: G =(V,E),where E= $|edges|$
Output: a vertex cover of G
  $MVC \leftarrow \emptyset$
   $dic_1$ /*node: neighbors*/
   $dic_2$ /*node: degree*/
   while $E \neq 0$ do
       $maxnode$= Sort($dic_2$)[0]
       for $n$ in $dic_1$ do
           if $n$ not in $MVC$ then
               Update $dic_2$
           end if
       end for
       $V \leftarrow V - \{maxnode\}$
       $E \leftarrow E - \{e\}$
       $MVC$.append(max node)
   end while
   return $MVC$
```

edges is $O(1)$ since here we actually use length to represent number of edges and it is just minus a number. The step to add the node to the set also consumes $O(1)$. Hence,the time complexity for the algorithm is $O(|V|^3)$. It satisfies the requirement of polynomial time for approximation method. We used two dictionaries and one solution set here. For $dic_2$ and set,their space complexly are $O(|V|)$. For $dic_1$, its space complexity is $O(|V|+|E|)$. So space complexity is $O(|V|+|E|)$.

#### 4.2.4 Approximation guarantee.

This method is a 2-approximation algorithm, because every edge picked by the algorithm contains at least one vertex of the optimal solution,as such, the cover generated is at most twice larger than the optimal. Its worst-case approximation ratio is $H(\delta)$, with $H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$, the harmonic series ($H(n) \approx \ln n + 0.57$ when$n$ is large) and $\delta$ is the maximum degree of the graph. Also, when $n$ tends to infinity, the limit of the expected approximation ratio is $1 + e^{-2}$. The above approximation guarantee analysis is referred from paper[4].

#### 4.2.5 Potential strength and weakness.

The biggest strength for this Maximum Degree Greedy algorithm is that its running time is small so it can quickly find a comparably not bad solution and from the result that we generated, the longest time it takes is around 8 seconds. Another strength is that this method is easy to understand and implement since it is mostly based on greedy algorithm. The weakness is that since it is only an approximation method so it can not guarantee to find a good solution especially when graphs are large and complicated even though it shows very small error for a small graph.

### 4.3 Local Search: Simulated Annealing

#### 4.3.1 Description.

The simulated annealing(SA) local search algorithm is inspired by annealing, a process of heating and cooling metals in order to improve strength[5][7]. The atoms in metals are first excited to move around actively by a high temperature. As the temperature decreases, atoms are less free to move around and fall into a new stable configuration. In the SA optimization algorithm, temperature is equivalent to the probability of accepting worse neighbouring solution. When temperature is high, worse neighboring solutions are more likely to be accepted. As temperature decreases, the probability of accepting worse neighboring solutions shrinks.

We start with an initial solution given by the approximation algorithm. Then we construct neighbors by randomly removing vertices from the solution until there are uncovered edges and randomly adding one vertex to the solution. The full size of one neighborhood should be $(|V| - |sol| - 1)^2$. If the neighboring solution is better than the previous solution, which means the size of the new vertex cover is smaller, it will be the current best solution. Otherwise, we will generate a random number following a uniform distribution between 0 and 1. The probability of accepting worse solutions is

$$p = \exp(-\delta f / T) \tag{2}$$

where $f$ is the objective function, $\delta f$ is the increase in $f$ and $T$ is the "temperature". If the number is smaller than the probability of accepting worse solutions, it will be the current best solution. Note that we decrease T by 5% in each iteration.

$$T = aT, a = 0.95 \tag{3}$$

Therefore the probability of accepting worse solution also decreases.

#### 4.3.2 Pseudo-Code.

The Pseudo-Code is shown in Algorithm 3.

#### 4.3.3 Time and Space Complexity.

The time complexity is $O(|V|)$. The space complexity is $O(|V| + |E|)$ because we use adjacency list to represent the graph.

#### 4.3.4 Potential Strength and Weakness.

The major advantage of SA is that it can avoid being trapped in local optimal which may be far from global optimal by tolerating a limited number of bad moves. It is also easy to implement. If the rate of cooling, $a$ in equation (3) is carefully selected, the SA can find the global optimal solution. However, since SA allocates more search in region of high fitness, it may fail if the true optimal is in a small region surrounded on all sides by regions of low fitness. While SA has interesting properties such as convergence, these are of limited practical relevance. The complexity and non-linearity of multivariate systems require more sophisticated numerical algorithms.

**Algorithm 3** Simulated Annealing

```
Input: Graph $G$, $|V|$, cutoff $t$
Output: Best Vertex Cover $sol_{best}$, $|sol_{best}|$
  $T \leftarrow 0.3$
  $sol \leftarrow$ initial solution from Approximation
  while running time $\geq t$ do
      $T = T * 0.95$
      if $sol$ is a VC then
          $sol_{best} \leftarrow sol$
          Maintain a list of uncovered edges
          Randomly remove one vertex from $sol$
      end if
      Randomly select an exiting vertex $u \in sol$
      Randomly select an entering vertex $v \notin sol$
      if $|sol| > |sol_{best}|$ then
          $p = \exp(\frac{|sol|-|sol_{best}|}{T})$
          $num = random.uniform(0, 1)$
          if $num < p$ then
              $sol_{best} \leftarrow sol$
          end if
      else
          $sol_{best} \leftarrow sol$
      end if
  end while
  return $|sol_{best}|$, $sol_{best}$
```

## 4.4 Local Search: Edge Weighting with Configuration Change

### 4.4.1 Description.

Edge Weighting Local Search with Configuration Change is an iterated local search, which is developed by Cai, Su and Sattar(2011) on the basis of Edge Weighting Local Search using Tabu method. Both method initialize all edge weights to 1 and maintain a set of uncovered edges. The weights of all uncovered edges will be increased by 1 if the algorithm gets stuck in local optimum.

The algorithm starts with a initial Vertex Cover reached by greedy algorithm, i.e., Approximation, which is not necessarily the MVC. Thus, the aim is to reduce the size of candidate vertex sets. Each iteration is a step from current candidate solution $C$ to the neighbourhood, which is defined by swapping two vertices: a vertex $u \in C$ is removed and a vertex $v \notin C$ is added into $C$. By leveraging on uncovered edges, the vertices that are endpoints of uncovered edges will have higher importance and be more likely to be added into the vertex cover.

To decide which pair of vertices $(u, v)$ should be swapped. The cost of current solution $C$ is defined by:

$$cost(G, C) = \sum_{e \in UncoveredEdges} w(e) \tag{4}$$

Hence for each $v \in V$, we have:

$$dscore(v) = \begin{cases} cost(G, C) - cost(G, C \backslash \{v\}), v \in C \\ cost(G, C) - cost(G, C \cup \{v\}), v \notin C \end{cases} \tag{5}$$

where $dscore(v) \geq 0$ if $v \notin C$ and $dscore(v) \leq 0$ if $v \in C$. To avoid cycling problem, which occurs when a pair of $(u, v)$ is repeatedly exchanged out and in, EWCC uses configuration checking strategy to decide whether or not a vertex $v \notin C$ can be added instead of $TabuAdd$ and $TabuRemove$ in EWLS. The configuration checking strategy defined $confChange[v]$ indicating if neighbours of $v$, i.e., vertices linked with $v$ have been added since $v$'s last leaving. It is updated following four rules:

- Initialize $confChange[v] = 1$ for $v \in V$.
- Update $confChange[v] = 0$ for $v$ removed from $C$.
- Update $confChange[v] = 1$ for each $v \in N(u) \backslash C$, when $u$ changes its state.
- If the weight of $e(u, v)$ is updated, both $confChange[u]$ and $confChange[v]$ are set to 1.

### 4.4.2 Pseudo-Code.

The Pseudo-Code is shown in Algorithm 4 and 5.

**Algorithm 4** Edge Weighting with Configuration Change

```
Input: Graph $G = (V, E)$, cutoff, random seed
Output: $VC$: Vertex Cover of $G$
  Initialize all edge weights as 1 and $confChange[v]$ as 1 for all vertices
  Construct $VC$ greedily until it is a vertex cover
  $C \leftarrow VC$
  $UncoveredEdges \leftarrow \emptyset$
  while running time $\geq$ cutoff do
      Compute $dscores[v]$ of all vertices
      if $UncoveredEdges = \emptyset$ and $len(C) < len(VC)$ then
          $VC \leftarrow C$
          Randomly remove a vertex with the highest $dscores$ from $C$
      end if
      if Exists $(u, v) := GetSwapPair(C, L, dscores)$ then
          $C = C \backslash \{u\} \cup \{v\}$
      else
          $u \leftarrow$ vertex with the highest $dscores$ from $C$
          $v \leftarrow$ endpoint with higher $dscore$ of a random edge $e \in UncoveredEdges$
          $C = C \backslash \{u\} \cup \{v\}$
      end if
      Update $UncoveredEdges$ and $confChange[N[v]]$
      $w(e) = w(e) + 1$ and reset $confChange$ of $e_u$ and $e_v$ to 1 for all $e \in UncoveredEdges$
  end while
  return $VC$
```

**Algorithm 5** GetSwapPair

```
Input: $C$: set of vertices; $UncoveredEdges$
Output: $(u, v)$: a pair of vertices to swap
  $u \longleftarrow$ vertex with the highest $dscores$ from $C$
  $v \longleftarrow$ endpoints of $e \in UncoveredEdges$
  if $|dscores[u]| < dscores[v]$ and $confChange[v] = 1$ then
      return $(u, random(v))$
  end if
  return None
```

### 4.4.3 Time and Space Complexity.

The time complexity of EWCC is $\mathbb{O}(|V|^2 + |E|)$ since at maximum it needs to check every possible pair of vertices . The space complexity is larger than plain local search method since it requires to store edge weights, *dscores* and conference change in list form, however, these are all linear increases, thus the space complexity is still $\mathbb{O}(|V| + |E|)$.

### 4.4.4 Potential Strength and Weakness.

The major advantages of EWCC comes from two aspects. Firstly, it inherits the advantage of increasing weights on uncovered edges to avoid being stuck into local optimum; secondly, it uses configuration changes to avoid cycling problems. However, a possible weakness is that EWCC may still falls into local optimum when weights of uncovered edges are increased to extreme high level. Therefore, a possible improvement is adding a forgetting threshold to reset edge weights and starting point. Also, in our application, a draw back of this algorithm is that calculating the cost and *dscores* take significant longer time, thus we refer to a less precise *dscore* calculation procedure and while select $u$ to remove, we also use *dscores* instead of total random to improve performance and hence the configuration process is simplified according to Chintapalli's work[3].

## 5 Empirical Evaluation

### 5.1 Branch and Bound

#### 5.1.1 Platform.

OS: macOS Ventura version 13.0

CPU: Intel Core i9-9880H @ 2.3 GHz (8 cores)

Memory: 16 GB 2667 MHz DDR4

Language: Python 3.8.8

#### 5.1.2 Evaluation Criteria and Procedure.

As the Branch and Bound algorithm takes exponential time, we set the cutoff time as 600 here to check the best solution found at that time for each graph. The time here is measured by CPU time. For each graph, we report its best solution and the time this solution was found. And we also report the relative error for each graph to see the accuracy of the Branch and Bound within a limited time.

The lower bound we used here follows the equation1 shown in part 4.1.1. The evaluation metrics can be the best vertex cover size we generated, and we can compare it to the optimal value with the relative error we provide.

#### 5.1.3 Results.

Table 1 includes the running time, best solution, and relative error for each graph using different algorithms. We can see that though the Branch and Bound algorithm guarantees the best solution for all graphs theoretically, it reaches the optimal solution only for a few graphs in 600 CPU seconds, like *jazz*, *karate*, *netscience* and two small dummy graphs. As the graph gets larger, the algorithm can hardly find the optimal set within our cutoff time.

Compared to other algorithms, the Branch and Bound can only have similar performance for easy graphs like *karate* and two dummy graphs. When the graph gets a little more complex, such as *jazz*, the time cost will increase fast. Though we can get the optimal solution within the cutoff time, the time consumed is much more than other algorithms. And when the graph gets really large, such as two *star* graphs, *as-22july06* and *hep-th*, the Branch and Bound algorithm can generate only one solution within the cutoff time, and the quality highly depends on the first solution found. So the relative error for large graphs is higher than other models and varies from graph to graph.

### 5.2 Approximation

#### 5.2.1 Platform.

Windows: Windows 11 version 21H2

CPU: Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz 2.11 GHz

Memory: 8.00 GB

Language: Python 3.9.13

#### 5.2.2 Evaluation Criteria and Procedure.

The algorithm we implemented for Approximation is Maximum Degree Greedy, which is referred from the paper[4]. We completely followed the logic to construct this algorithm but used dictionary data structure to help to decrease its running time. The time here is measured by CPU time. For each graph, we report the running time to find the complete solution, and the size of this vertex cover, also the relative error for each graph.

#### 5.2.3 Results.

Table 1 includes the running time, our best solution, and relative error for each graph using different algorithms. We can see that for Approximation method, the average running time is much smaller than other methods, the longest time it takes is only about 7 seconds which is for graph *as-22july06*, and it also shows that when the graph is more complicated, it takes longer time but it still runs much faster than other.

With respect to accuracy and relative error, from table 1 we can see that when the graph is easy, Approximation can find a good solution as other methods, but when a graph is more complex, its relative error is larger than other methods, such as graph *star,star2,power*,their error can reach maximum at 0.0684, which means that if we want to guarantee an optimal solution, Approximation can only work when the input or the graph is small. Otherwise it will get evident larger error than other methods.

## 5.3 Local Search: Simulated Annealing

### 5.3.1 Platform.

OS: macOS Monterey version 12.2
CPU: Apple M1
Memory: 8GB unified memory, configurable to 16GB
Language: Python 3.9.7

### 5.3.2 Evaluation Criteria and Procedure.

The evaluation criterion is solution probability when the running time hits cutoff $t_{max}$. The running time is measured by CPU time.

The protocol for obtaining the empirical Run-Time Distribution (RTD) for an SA $A$ applied to a given instance $\pi$ is as follows:

1. Generate 20 random seeds and perform 20 independent runs of $A$ on $\pi$ with cutoff time 600.
2. For each run, record its final solution in a solution file; record its run time and size of every updated solution in a list in a trace file.
3. Average the total running time, size of final solutions, and relative error of each run.
4. Select a range of solution qualities $q$ to plot qualified run-time distribution (QRTD), defined by:
$$qrtd_q(t) = rtd(t,q) = P(RT_{A,\pi} \leq t, SQ_{A,\pi} \leq q) \quad (6)$$
5. Select a range of running time $t$ to plot solution quality distribution (SQD), defined by:
$$sqd_t(q) = rtd(t,q) = P(RT_{A,\pi} \leq t, SQ_{A,\pi} \leq q) \quad (7)$$

### 5.3.3 Results.

Table 1 reports average time, average vertex cover size, and the relative error with respect to the optimum solution quality. SA is able to find the global optimal for 10 out of 13 instances within 160 seconds. Given a cutoff of 600 seconds, the relative errors for *star*, *star2*, *delaunay_n10* are 0.0103, 0.0014, 0.0003, respectively. It is interesting to note that the solution quality is not necessarily negatively related with number of vertex or number of edges. Although the number of vertice in *as-22july06* is twice that in *star*, SA can still achieve 0 relative error for *as-22july06*. Similarly, $|E|$ of *as-22july06* is more than 15 times that of *delaunay_n10*. Surprising, SA can find the global optimal in *as-22july06* but not *delaunay_n10*. This implies that the structure of graphs may have a significant impact on the performance of SA.

Figure 1 and 2 are QRTD plots for instances *power* and *star2*. We set the relative error as 0.5%, 1.0%, 1.5%, 2.0% and 2.5% for *power* and 2.0%, 3.0%, 4.0%, 5.0% and 6.0% for *star2*. For the *power* instance, SA achieves the quality of 2.5% within less than 1 second. For the *star2* instance, it takes less than 25 seconds to achieve the quality of 2.5% error. SA takes more time to solve *star2* than *power* and the quality of the final solutions is worse for star2 than power. As quality requirement increases, it takes more running time to achieve 0.5% improvement in both instances. In practice, if the quality threshold can be reached within seconds such as 2.5% for the *power* instance, SA can be completed within 1 second rather than 600 seconds, which greatly saves time.

Figure 3 and 4 are SQD plots for instances *power* and *star2*. We set the time as 5,10,15,20 and 60 for *power* and 15,20,25,40 and 60 for *star2*. For the *power* instance, when running time hits 5 seconds, roughly 40% of the 20 independent runs achieves quality of 0.6% error. If time limit raises, the percentage of runs achieves the same level quality increases but the speed of improvement decreases. The quality improvement is more significant in the time interval between 5-10 seconds than 15-20 seconds. The speed of improvement decreases is relatively stable in the *star2* instance because the solution is far from optimal.

Figure 9 and 10 are box plots for running time for instances *power* and *star2*. As expected, the performances of SA in both instances are not the same for each independent run.

## 5.4 Local Search: Edge Weighting with Configuration Change

### 5.4.1 Platform.

OS: macOS Monterey version 12.2.1
CPU: Apple M1
Memory: 16GB
Language: Python 3.8.8

### 5.4.2 Evaluation Criteria and Procedure.

The evaluation criterion and protocol for obtaining the empirical Run-Time Distribution (RTD) for EWCC algorithm applied to a given instance are similar SA. Here, we generate 10 random seeds and perform 10 independent runs of each instances with cutoff time 600. The running time is measured by CPU time.

### 5.4.3 Results.

From Table 1, we can see that regarding average running time, best solution, and relative error, Edge Weighting with Configuration Check reaches the optimal solutions only for graph instances *karate*, *netscience*, *dummy1* and *dummy2* within a cutoff of 600 seconds, which is not as satisfying as other algorithms. For large graphs like *star* and *star2*, EWCC performs relatively good giving a vertex cover closest to the optimal. The highest relative error happened with the instance *football*, which is 0.0106. And for a small graph *jazz*, EWCC didn't find the optimal and takes longer time comparing to other algorithm which may due to the lack of forgetting strategy. The EWCC algorithm should not be instance dependent, thus this result may indicates our EWCC algorithm is not as complete as described in the paper.

The QRTD plots of instances *power* and *star2* are shown as Figure 5 and Figure 6.The relative errors for *power* are 0.2%, 0.4%, 0.6%, 0.8% and 1.0%, and EWCC achieves the quality of 1.0% error with in 12.5 seconds, however, it takes much loner, around 30 seconds, to reduce it to 0.2%. The relative errors for *star2* 1.8%, 2.0%, 2.2%, 2.4% and 2.6%. EWCC achieves the quality of 2.6% within 160 seconds and 2.0% within 200 seconds. We can observe that for *power*, the same quality improvement of 0.2% needs more time as the quality becoming higher. And the plot for *star2* proves that for getting a relatively good result, EWCC is fast, but the improvement can be time consuming.

The SQD plots of instances *power* and *star2* are shown as Figure 7 and Figure 8. The time for *power* are 10, 15, 20, 30 and 40 and for *star2* 350, 400, 450, 500, 550. For *power*, when running time hits 10 seconds, 80% of the 10 independent runs achieves quality around 1.1% error. For *star2*, the running times are of a smaller range and similar over the 10 independent runs, given 550 seconds, around 80% runs reach 0.02% error and given 350 seconds only 10% can reach the quality of 0.02% error. Both figures prove that the percentage achieves a given level of quality increases as time limit increases.

Also, from comparing Box plots of instances *power* and *star2* as Figure 9 and 10, we can see that EWCC outperforms SA for smaller graphs like *power* in terms of running time. But for large graphs like *star2*, these two algorithms perform similarly.

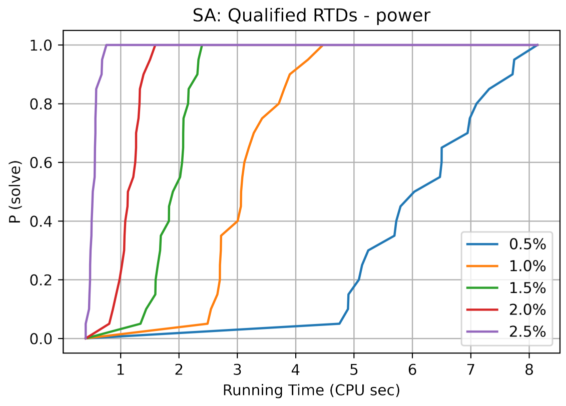

**Figure 1.** SA: QRTD-power

## 6 Discussion

The Branch and Bound is the only algorithm here to guarantee an optimal solution. However, this is hard to be realized in implementation. Compared to other algorithms, the Branch and Bound can only perform similarly for easy graphs. The time cost will increase fast as the graph gets more complex.

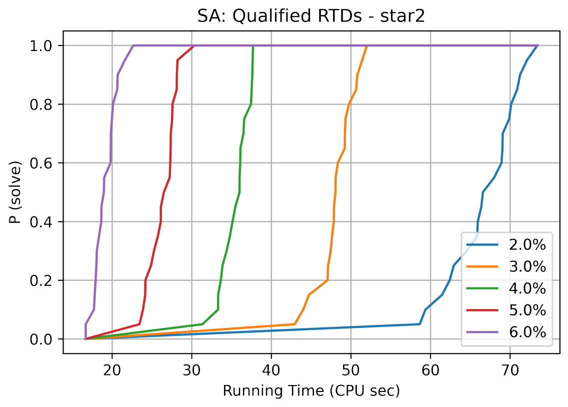

**Figure 2.** SA: QRTD-star2

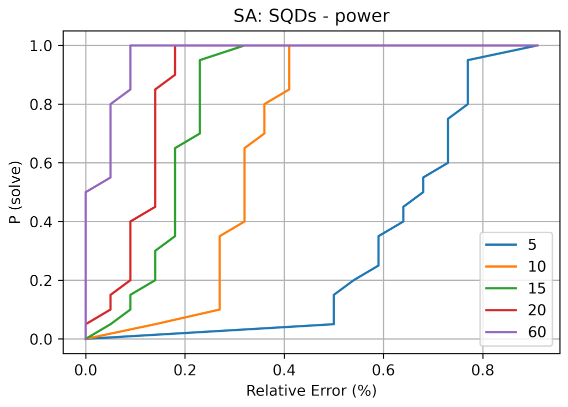

**Figure 3.** SA: SQD-power

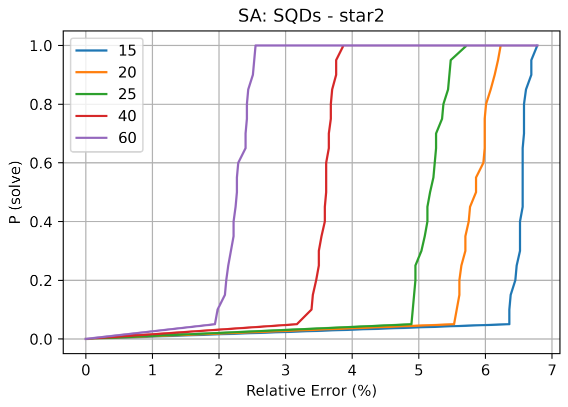

**Figure 4.** SA: SQD-star2

And when the graph becomes really large, the Branch and Bound algorithm can generate only one solution within our cutoff time, and we don't have any insurance for the quality of this solution. The relative error for large graphs is also

**Table 1.** Comprehensive Table

| | BnB | | | Approx | | | LS1: SA | | | LS2: EWCC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dataset | Time(s) | VC Value | RelErr | Time(s) | VC Value | RelErr | Time(s) | VC Value | RelErr | Time(s) | VC Value | RelErr |
| jazz | 153.57 | 158 | 0.0000 | 0.00 | 159 | 0.0063 | 0.00 | 158 | 0.0000 | 0.02 | 159 | 0.0063 |
| karate | 0.00 | 14 | 0.0000 | 0.00 | 14 | 0.0000 | 0.00 | 14 | 0.0000 | 0.00 | 14 | 0.0000 |
| football | 0.26 | 95 | 0.0106 | 0.00 | 96 | 0.0213 | 0.00 | 94 | 0.0000 | 0.00 | 95 | 0.0106 |
| as-22july06 | 95.59 | 3,312 | 0.0027 | 6.81 | 3307 | 0.0012 | 37.00 | 3,303 | 0.0000 | 303.96 | 3304 | 0.0003 |
| hep-th | 27.74 | 3,947 | 0.0053 | 2.52 | 3944 | 0.0046 | 155.00 | 3,926 | 0.0000 | 43.20 | 3927 | 0.0003 |
| star | 64.88 | 7,366 | 0.0672 | 4.19 | 7374 | 0.0684 | 576.00 | 6,973 | 0.0103 | 553.29 | 6925 | 0.0034 |
| star2 | 72.93 | 4,677 | 0.0297 | 5.69 | 4697 | 0.0341 | 522.00 | 4,548 | 0.0014 | 415.31 | 4543 | 0.0003 |
| netscience | 0.90 | 899 | 0.0000 | 0.09 | 899 | 0.0000 | 0.00 | 899 | 0.0000 | 0.00 | 899 | 0.0000 |
| email | 0.60 | 605 | 0.0185 | 0.09 | 605 | 0.0185 | 5.00 | 594 | 0.0000 | 0.86 | 595 | 0.0017 |
| delaunay_n10 | 2.72 | 739 | 0.0512 | 0.06 | 737 | 0.0484 | 102.00 | 703 | 0.0003 | 3.05 | 704 | 0.0014 |
| power | 10.06 | 2,272 | 0.0313 | 0.94 | 2277 | 0.0336 | 78.00 | 2,203 | 0.0000 | 33.73 | 2204 | 0.0005 |
| dummy1 | 0.00 | 2 | 0.0000 | 0.00 | 2 | 0.0000 | 0.00 | 2 | 0.0000 | 0.00 | 2 | 0.0000 |
| dummy2 | 0.00 | 3 | 0.0000 | 0.00 | 3 | 0.0000 | 0.00 | 3 | 0.0000 | 0.00 | 3 | 0.0000 |

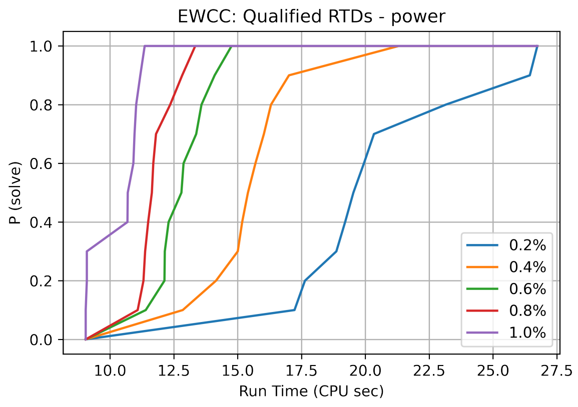

**Figure 5.** EWCC: QRTD-power

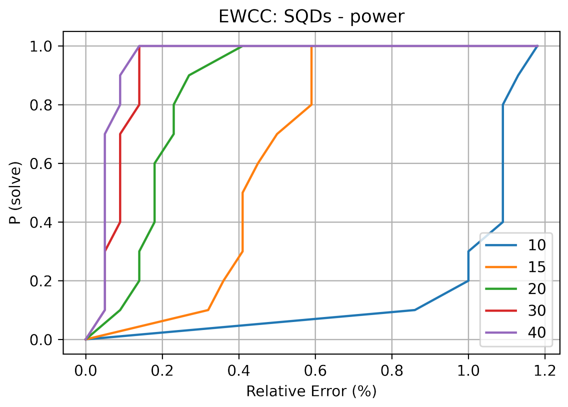

**Figure 7.** EWCC: SQD-power

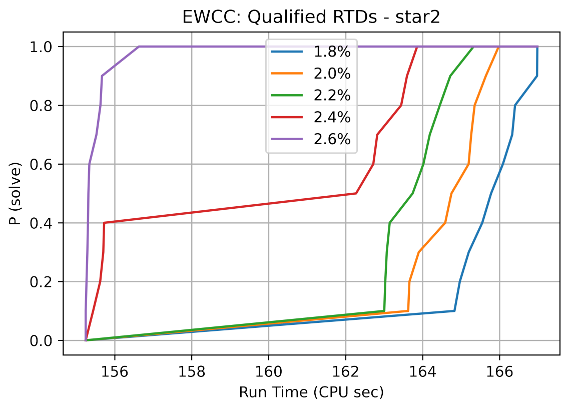

**Figure 6.** EWCC: QRTD-star2

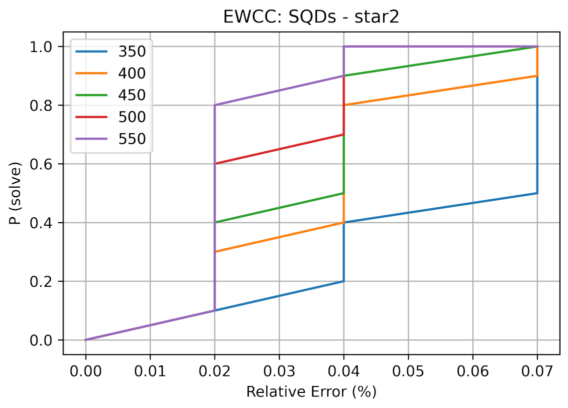

**Figure 8.** EWCC: SQD-star2

much higher than other models and varies from graph to graph.

The Approximation method gives us acceptable results within a very short time, like what we have mentioned before,the maximum time it consumes is only a few seconds,

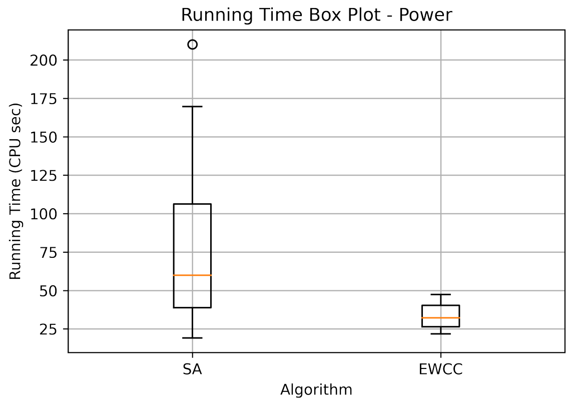

**Figure 9.** Running Time Box Plots -power

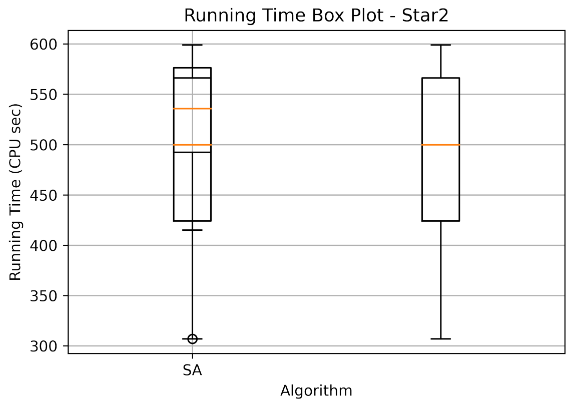

**Figure 10.** Running Time Box Plots -star2

but with the complexity of the graph increasing, its time and relative error also increase, especially for the relative error,it becomes much larger than other three methods.Also, we notice that the empirical time consumes is much smaller than our theoretical analysis,which may due to that there graphs are sparse,so when it comes to the checking and updating neighbor step in Approximation, the actual elements are far smaller than $|V|$, so there is a possibility that when the graph becomes more dense, the performance of this approximation might become worse.

Since two local search algorithms take the result of approximation algorithm as their initial solution, the local search algorithms are expected to take more time and yield better solutions than the approximation algorithms. Given a hard cutoff of 600 seconds, SA performs better than EWCC in terms of relative error. SA finds the optimal solution in more instances than EWCC. However, the running time shows that EWCC is more efficient in handling graphs such as weighted network, star-like structure, and delaunay triangulations. Take *delaunay_n10* as an example, SA takes 102 seconds to find the optimal solutions while EWCC only takes 3 seconds. Although EWCC is one vertex off the true optimal and SA hits the optimal, in practice if the user can tolerate the relative error at 0.14%, EWCC is the recommended algorithm. On the other hand, if users prioritize quality over time, SA can be a better choice. From the perspective of instance size, EWCC gives a smaller MVC within similar running time when the size of the graph instance is large. For example, *star* and *star2*. However, for smaller graphs like *jazz*, EWCC requires longer time to reach optimal due to the lack of forgetting strategy.

## 7 Conclusion

In this project, we implemented 4 algorithms to solve the Minimum Vertex Cover problem. The Branch and Bound algorithm guarantees to find the optimal theoretically but its running time can be exponential. The Approximation algorithm can quickly find a acceptable solution which may be not the optimal especially for complex instances. Our two local search algorithms, SA and EWCC, improve the solution from approximation algorithm. Compared with EWCC, SA can find the true optimal solutions for more instances but underperform in terms of time and relative error for complex graphs such as *star2* and *delaunayn10*. For future studies, it would be interesting to explore how the structure of graphs impact the performance of different algorithms.

## References

[1] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. 2013. NuMVC: An Efficient Local Search Algorithm for Minimum Vertex Cover. *Journal of Artificial Intelligence Research* 46, 1 (2013), 687–716.

[2] Shaowei Cai, Kaile Su, and Abdul Sattar. 2011. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artificial Intelligence* 175, 9 (2011), 1672–1696.

[3] Arjun Chintapalli. 2019. *Minimum-Vertex-Cover*. https://github.com/arjunchint/Minimum-Vertex-Cover

[4] Francois Delbot and Christian Laforest. 2019. Analytical and experimental comparison of six algorithms for the vertex cover problem. *Journal of Experimental Algorithmics(JEA)* 15 (2019), 1–4.

[5] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by Simulated Annealing. *Artificial Intelligence* 220, 4589 (1983), 671–680.

[6] Luzhi Wang, Shuli Hu, Mingyang Li, and Junping Zhou. 2019. An exact algorithm for minimum vertex cover problem. *Mathematics* 7, 7 (2019), 603.

[7] V Černý. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications* 45 (1985), 41–51.