CSE 6140 Project - Minimum Vertex Cover-Group22

Mengzhen Chen*
Georgia Institute of Technology
Atlanta, GA, USA
mchen328@gatech.edu

Xiao Jing
Georgia Institute of Technology
Atlanta, GA, USA
xjing33@gatech.edu

Judu Xu
Georgia Institute of Technology
Atlanta, GA, USA
jxu490@gatech.edu

Yifan Li
Georgia Institute of Technology
Atlanta, GA, USA
yli3458@gatech.edu

Abstract

The Minimum Vertex Cover (MVC) problem is a widely-studied NP-complete problem which has been applied in different scenarios. In this project, we will apply different algorithms, such as Branch and Bound (BnB), Approximation and Local Search, to solve the MVC problem and the results will be evaluated based on some criteria. Furthermore, several real-world datasets will be utilized in this project for performance evaluation.

1 Introduction

In the paper, we discuss about different algorithms to resolve the minimum vertex cover(MVC) problems. The detailed problem introduction is in Part2. The algorithms that we choose are Branch and Bound, Approximation, Simulated Annealing and Hill Climbing. For Branch and Bound algorithm, it guarantees the accuracy of our result, but it will take exponential time to run. The Approximation takes the least time get a solution and its weakness is that it can not guarantee its accuracy. And for the rest two local search algorithms: Simulated Annealing and Hill Climbing, the SA algorithm is more accurate to get the solution for MVC. Neither algorithms is guaranteed to get the global optimal solution.

2 Problem Definition

A formal definition of the Minimum Vertex Cover problem can be defined as follows:

Given an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, a Vertex Cover (VC) is defined as a subset $C \subseteq V$ such that $\exists(u, v) \in E : u \in C \vee v \in C$. The Minimum Vertex Cover (MVC) problem is to find a vertex cover with the smallest size.

Figure. 1 shows two examples[15] of the MVC problem. The nodes in red is a minimum vertex cover.

3 Related Work

The Minimum vertex cover (MVC) problem is a well-known problem and is a NP-hard problem with wide applications.

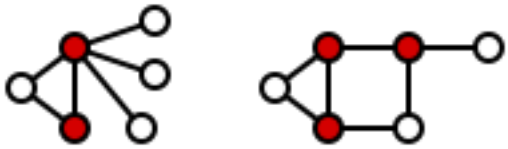

Figure 1. Example of Minimum Vertex Cover Problem

During the past several decades, multiple methods have been developed and proposed to solve this problem.

Branch and Bound (BnB) method [2, 12] is an algorithm aims at searching for an exact solution. Since it's not searching for an approximation, it will take a while to find a solution. In the work [14], Luzhi Wang et al. proposed a branch-and-bound algorithm to solve exactly the minimum vertex cover (MVC) problem, and defined two novel lower bounds to help prune the search space, this would be helpful for us to set up the upper and lower bound in our BnB algorithm.

There are several Approximation algorithms out there that provide a good approximation to the vertex cover solutions [10] detailed in Delbot et al.'s paper. They discussed six polynomial time algorithms for the MVC problem that have approximation ratios between 2 and Δ (the maximum degree of the graph) [10]. In our project, we implemented an approximation algorithm with random selection of edges with ratio of 2, which is considered the best algorithms [10]. It is not hard to augment it with some ways of prioritizing the remaining vertex set E' [11]. Such as by replacing the random selection of E' to selecting a vertex with maximum degree, which is so called *Maximum Degree Greedy* (MDG) that an approximation ratio of $\ln \Delta + 0.57$ [10]. *Depth First Search* (DFS) is another approximation algorithm that returns the non-leaf nodes of DFS spanning tree with the assumption that G is connected [10]. Overall, the best approximation ratio can be achieved so far is 2.

Another category of algorithms to solve such kind of problem is called local search approach. The core idea of local search is to keep trying new solutions by exchanging parts of its current solution with the candidates in the solution space. The objective of such approach is to reduce the gap

*Both authors contributed equally to this research.

between the current solution with the real optimum by keeping trying new solutions. It hopes the quality of the solution can reach a certain level after some pre-defined cutoff time.

There are multiple methods proposed to study this problem [5, 7, 16]. Cai et al. [6] proposed the NuMVC methods which including two new strategies: the two-stage exchange and the edge weighting with forgetting techniques. Later, in 2015, Cai et al. [4] proposed a method named as FastVC which can achieve relatively fast speed on massive graphs.

4 Algorithms

4.1 Branch and Bound

4.1.1 Algorithm Description: In general, the Branch and Bound (BnB) algorithm is used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems. For example, if we have an NP-Hard problem, BnB algorithm can help us to explore the entire search space of possible solutions and provides an optimal solution [9]. In this project, the end gold of the BnB (Brand and Bound) algorithm is to find the smaller subsets of the parent graph G for potential solutions (Vertex Covers), while cutting off parts of the search tree that cannot meet the defined bounds for the optimal solution. In BnB algorithm, the bounds are an important aspect, since explicit enumeration is usually impossible due to the exponentially-increasing number of potential solutions [14]. Therefore, when it branches on a vertex, the solution space is divided into a set of smaller subsets and it obtains the upper and lower bound for each node to further reduce search space [14]. The bounds to the optimal solutions can be defined in different ways, but they have to be able to be proven to reach the optimum and are constantly iterating to scope down the list of possible solutions and it will be described in the next subsection.

4.1.2 Algorithm Implementation: In our project, for the implementation of the BnB algorithm, first, we followed the description of the project given by the instructor [11], we let C' be a partial vertex cover of G and $G' = (V', E') \subseteq G$ and not covered by C'. Also, let $V' = V - C' - \{v \in V | \forall (v, u) \in E, u \in C'\}$ and $E' = \{(u, v) \in E; u, v \in V'\}$. Then, for the upper bound solution, we referred to the project instructions by the professor [11] and the paper was worked by Luzhi Wang et al. [14], the upper bound is supposed to be the overestimation of the size of the minimum vertex cover of the sub-graph, can be obtained by computing the size of the minimum vertex cover found so far. Thus, in our case, the **upper bound** solution is defined as the best solution for vertex cover of the smallest size met at any given step of the discovery process. At the same time, the lower bound solution in our algorithm is also referenced to the project instructions by the professor [11] and the academic paper work [14]. After the adoption, the lower bound solution for the subset graph is depicted as the following. Because the maximum degrees of nodes have the most possibility to be obtained in the vertex cover, the Bnb algorithm in our project explored nodes in decreasing order of degrees. Then each node has a binary state/indicator 0 or 1 to imply whether belongs to Vertex Cover or not. Also, we are given in the project instruction, $LB = |C'|$+(Lower bound on the VC for G'). Eventually, each choice will be evaluated with the **Lower Bound** for G' as:

$$Lower\ Bound = |C'| + \frac{\text{Number of edges in graph } G'}{\text{Maximum node degree in graph } G'}$$

The criteria behind this choice are that we defined a partial graph as a total of $|E|$ edges (given in project instructions) and d is the maximum node degrees, then the size of the vertex cover of the partial graph should be at most $\frac{|E|}{d}$. In this implementation, the initial upper bound for the root node will be the total number of nodes in G and it will be iterated and updated to the best vertex cover found in the process of the exploration.

4.1.3 Pseudo-code: The Pseudo-code for BnB algorithm is shown in the **Algorithm 1** below:

4.1.4 Time and space complexity: Through the algorithm we shown above, we can find that the set of the candidate vertices for subproblem ("CandidateVerticesSub" in pseudo-code) will include all the candidate vertices to be explored for the entire possibilities of the state variables. Therefore, the total number of possible scenarios in this case will be $2^{|V|}$ (V is given in the original project instructions). Therefore, the **time complexity** of the algorithm Will be $O(2^{|V|})$ that is also indicated the number of iterations for the while loop in the pseudo-code, and also for each iteration will take $O(V + E)$ (V and E all produced in the previous subsection)computational time to explore a specific branch of the searching space.
The **space complexity** will be $O(|V| + |E|)$ and we need $2V$ space to store all candidate vertices in the set of the candidate vertices for subproblem ("CandidateVerticesSub" in pseudo-code) and need E space to obtain Vertex Cover solution.

4.1.5 Pros and cons of this method: The advantages of the BnB algorithm is that it can provide relatively accurate solutions as well as generally exploring less subproblems compared to Brute Force to save computational time.
The disadvantages of the BnB is that we can find that it requires require more storage and more branching computation thus less computaional efficiently compared to other algorithms such as: Approx.

4.2 Approximation

4.2.1 Algorithm Description. The heuristic algorithm implemented to find a MVC with approximation guarantees is straightforward with randomly picking any remaining

Algorithm 1: Branch and Bound (BnB) for MVC

Input: InputGraphData, Cut-off-time
Initialize:
CurrentVertexCover ← []
CandidateVerticesSub ← []
CurrentGraphData ← InputGraphData
V_{max} ← Vertex with maximum degrees in CurrentGraphData
UpperBoundSize ← Size of nodes in InputGraphData
Times-List ← []
while $CandidateVerticesSub \neq \emptyset$ *and* $RunTime <$ *cut-off-time* **do**
  $(V_{max}, BinaryIndicator, ParentNodes)$ =Vertex with Maximum Degrees;
  if $BinaryIndicator == 1$ *(//1 means* V_{max} *is selected)* **then**
    Delete V_{max} *from CurrentGraphData ;*
    Append V_{max} *to CurrentVertextCover ;*
  else if $BinaryIndicator == 0$ *(//0 means* V_{max} *is not selected)* **then**
    Delete V_{max} *from CurrentGraphData ;*
    Append all incident nodes to CurrentVertexCover ;
  if *CurrentGraphData* == $\emptyset$ **then**
    if *length*$(CurrentVertexCover) <$ $UpperBoundSize$ **then**
      Backtracking is activated ;
      UpperBoundSize = length(CurrentVertexCover);
      Append length(CurrentVertexCover), $\_\delta time) toTimes - List$
    if $CandidateVerticesSub \neq \emptyset$ *and* $ParentNodes \in CurrentVertexCover$ **then**
      Iterating CurrentVertexCover and delete nodes;
      Append the removed nodes back to CurrentGraphData;
    else
      Reset CurrentVertexCover ← $\emptyset$ *;*
      Reset CurrentGraphData ← *InputGraphData;*
  else
    LowerBoundSize= $\frac{Number\ of\ edges\ in\ CurrentGraphData}{Maximum\ degree\ node\ in\ CurrentGraphData}$;
    LowerBoundSizeUpdating= int(LowerBoundSize) + length(CurrentVertexCover);
    if $UpperBoundSize > LowerBoundSizeUpdating$ **then**
      V_{maxnew}*= Maximum degree of CurrentGraphData;*
      Append (V_{maxnew}*, BinaryIndicator=0, ParentVertex);*
      Append (V_{maxnew}*, BinaryIndicator=1, ParentVertex) ;*
    else
      if $CandidateVerticesSub \neq \emptyset$ *and* $ParentNodes \in CurrentVertexCover$ **then**
        Backtracking is activated ;
        Iterate CurrentVertexCover and delete nodes;
        Append the deleted nodes back to CurrentGraphData;
      else
        Reset CurrentVertexCover ← $\emptyset$*;*
        Reset CurrentGraphData ← *InputGraphData;*
  _
  Return*: Optimal Vertex Cover, Times-List*

edge in E', as discussed in class. We initialize an empty vertex set C to keep track of vertices that are going to be added to cover the edges, and a set of edges E' that have yet to be selected from, essentially all edges E from the graph G in the beginning. For every iteration in the while loop, we randomly select an edge from the remaining edge set E', then check if one of the two vertices already exist in C, if yes, we mark that edge as selected by removing it from E', if not, we add the two vertices to C and marked it as selected by removing it from E'. We continue the while loop until no edges are left remaining and return the final vertex cover C [1].

4.2.2 Approximation guarantee.

We can observe that the set of edges randomly picked by this algorithm is a *maximal matching* that forms a edges disjoint with no two edges touch each other. As such, we can reframe the algorithm as follows: Find a maximal matching M and return the set of vertices of all edges $\in M$.

Claim 1: The modified algorithm returns a vertex cover

Proof: Every edge $\in M$ is covered and if not, then $M \cup e$ is a matching, which contradicts the M's maximality.

Claim 2: This vertex cover has size $\leq 2\times$ the size of vertex cover from the optimal solution OPT

Proof: Since the optimal vertex cover must cover every edge in M, it must include at least one of the two vertices of each edge $\in M$ where no two edges in M share an vertex. Therefore, the optimal vertex cover must have size

$$OPT(C) \geq |M|$$

Since the algorithm $Approx$ returns a vertex cover of size $2|M|$, we have

$$Approx(C) = 2|M| \leq 2 \times OPT(C)$$

proving that the approximation algorithm guarantees $2 - approximation$ of the optimal solution [1, 11].

4.2.3 Pseudo-code.

(see Algorithm 2)

Algorithm 2: Approximation (Approx) for MVC

Data: Graph: $G = (V, E)$
Result: Minimum vertex cover: C
$C \leftarrow \emptyset$;
$E' \leftarrow E$;
while $E' \neq \emptyset$ **do**
  pick any $\{u, v\} \in E'$;
  $C \leftarrow C \cup \{u, v\}$;
  remove all edge incidents in E' to either u or v
return C

4.2.4 Time and space complexity.

Time complexity and Space complexity are $O(|E|^2)$ and $O(|E| + |V|)$ respectively.

4.2.5 Pros and cons of this method.

The biggest advantage of this algorithm has is its simplicity, both in the time and space complexity. Compare to the heuristic algorithms, the structure is much easier to understand and implement. However, the obvious weakness is that it is often less optimal than the other algorithms due to its $2 - approximation$ nature. The vertex cover generated typically has larger size than those returned by others.

4.3 Local Search: Simulated Annealing

4.3.1 Algorithm Description and Pseudo Code.

Simulated Annealing (SA) is a combinatorial search technique inspired by the physical process of annealing[8, 13]. The

physical process of annealing is utilized during heating and cooling metals. By recursively reheating the metal with a decreasing probability over time, the strength of the metal can be enhanced since the atoms will actively move around during high temperature and will reach a lower internal energy during cooling down. By doing this multiple times, these atoms often find a lower internal energy state compared with their initial state and thus the strength of the metal has been enhanced.

Similarly, the idea of the simulated annealing in local search algorithm will also choose to accept a worse case based on some probability. It will have a controlling variable similar to the temperature used in the physical world to let the probability of accepting the worse case decrease with time. The benefits of such kind of design is to let the algorithm to have two different solution searching strategies during the entire process. In the early stage, higher probability of accepting the worse case enables algorithm to focus on exploring other solution possibilities. With the passage of time, the possibility of accepting a case which is worse than the current one is decreasing and the solving strategy switches to improve the solution quality of the current local solution.

A detailed description of the proposed Simulated Annealing algorithm is shown in Algorithm. 3. Based on the pseudo code, we can see the current solution is modified by randomly removing a vertex in the current vertex cover and then adding a vertex which is not included in the current vertex cover. Decision making of whether accepting the modified solution or not is purely defined by a probability equation in Eq. 1.

$$P = \frac{|Sol_{cur}| - |Sol_{best}|}{Temperature} \tag{1}$$

This probability is proportional to the difference between the current solution and the current best solution and such difference is scaled by the temperature which will be decreasing as the passage of time. It will give higher possibility to accept the randomly modified solution during the early stage while hard to accept that during the late stage. According to Ben-Ameur [3]'s work, the initial temperature is selected as 0.8 and the decreasing rate is selected as 0.95.

The process of generating an initial solution is described as follows: At the beginning, we include all the nodes as the vertex cover, then each time we randomly select a node to remove. Before each remove, we check all its neighboring node, if any one of them are also not in the current vertex cover, we will stop the removing work and the resulting vertex cover will be our initial vertex cover.

For the cutoff time selection, briefly we first run the model with a large cutoff time, let's say 500s. Based on the trace file, we can check how long it has reached the real optimum or hasn't reached that yet. If it has already reached the global optimum far before the cutoff time, we will decrease that value for faster computation. If it's still sub-optimal, we will increase the cutoff time and we set the maximum cutoff time for SA is 1000s.

Algorithm 3: Simulated Annealing (SA) for MVC

Data: Graph: $G = (V, E)$, Initial solution: Sol_{ini}, Computation cutoff time: $cutoff$
Result: Minimum vertex cover: Sol

```
Temp ← 0.8;
Sol_best ← Sol_ini;
Sol ← ∅;
while Hasn't reached the cutoff time do
    Temp ← 0.95× Temp;
    if The current solution Sol_cur is a vertex cover then
        Sol ← Sol_cur;
    Randomly remove a vertex u from the current solution Sol_cur;
    Randomly add a vertex v into the current solution Sol_cur which previously is not in the the solution set.;
    if |Sol_cur| < |Sol_best| then
        P = exp(|Sol_cur|−|Sol_best|) / Temp;
        Randomly pick a number α in the range [0, 1];
        if α > P then
            Sol_best ← Sol_cur;
return Sol
```

4.3.2 Time and Space Complexity.

Since the SA algorithm will keep running until reaching the cutoff time. Therefore, the time complexity discussed here will focus on each round. For each round, we remove a vertex and add a vertex and finally calculate a probability. During removal and addition, we will search the neighbors of the current node and that has the time complexity of $O(|V|)$. For space complexity, since we will only store the information of the graph, therefore, the space complexity is $O(|V| + |E|)$.

4.3.3 Pros and Cons.

Simulated Annealing (SA) can find a solution which is close to the real optimum in quite a short time. Though global optimum is not guaranteed, the resulting sub-optimum usually has good solution quality.

4.4 Local Search: Hill Climbing

4.4.1 Algorithm Description.

Hill climbing is an classic local search algorithm. It is an iterative algorithm which will start from a arbitrary solution to the problem. It will continue to update the solution until it reaches an peak. It is also called the greedy local search. For the minimum vertex cover problem, we will firstly start from a arbitrary set of vertex, which realize the vertex cover. And the next step we need to delete the vertex which won't break the vertex

cover and make the size of vertex set become smaller. The algorithm will continue on these steps until it reaches the peak. The peak means that we get the minimum set of vertex that is still vertex cover. Since we will check at one node each time, so the neighbors of the current solution are the sets of vertex that deletes one node from it.

4.4.2 Detailed implementation. Firstly, we start a with a initial solution, the most simple method to get a initial solution is to get all the vertex into the initial solution. It is guaranteed to be vertex cover. Then we store all the vertex and their degrees into a dict and the key is the degree. We start to choose random vertex that has the smallest degrees. And then we will delete the vertex from the current solution and if it is not vertex cover, it means the deleted vertex is necessary to our solution. So we add the vertex back to the solution. By this way, we can keep our solution to be vertex cover to make it a guaranteed solution. And for the nodes have been checked, we will delete them from unchecked set of nodes. So all the nodes will be checked only once and it won't be considered for multiple times. When all the nodes are checked or beyond the cutoff, the algorithm will finish.

4.4.3 Pseudo-code. A detailed description of the proposed Hill Climbing algorithm is shown in Algorithm. 4.

Algorithm 4: Hill Climbing (HC) for MVC

Data: Graph: $G = (V, E)$ and Computation cutoff time: $cutoff$
Result: Minimum vertex cover: *Sol*

```
Sol ← initialize(G.V);
Dict ← ∅;
for v in G.V do
    Dict[v.degrees] add v ;
while smaller than cutoff time and Dict not empty do
    Randomly get a vertex v from Dict[min degrees];
    Sol_cur ← Sol delete v;
    if The current solution Sol is vertex cover then
        Sol ← S_cur;
    Del v from Dict[min degrees];
    if Dict[min degrees] is empty then
        Dict del the key of min degrees;
return Sol
```

4.4.4 Time and Space complexity. For time complexity, the step to sort all the vertex is $O(|V| \log |V|)$. And for the while loop, we will iterate all the nodes and to check if it is still vertex cover, it will take $O(|E|)$, since the worst case is all the nodes are connected. The time complexity will be $O(|V||E|)$. So the time complexity will the larger value between $O(|V| \log |V|)$ and $O(|V||E|)$. For space complexity, the max space that we use is for information of the graph, therefore, the space complexity is $O(|V| + |E|)$. If we just consider the algorithms part excluding the input graph, it will be $O(|V|)$, since we just store all the nodes for calculation.

4.5 Pros and cons

The advantage of hill climbing is that it is good to solve problems with limited computation power. For each step of our local search, the search range is narrowed to nodes that has minimum degree. It is direct and easy understanding. We don't need to consider nodes for multiple times and it will be easy for us to get a vertex cover solution. The disadvantage is that we can not guarantee that we get the best solution. We can just get the local maximum and it may not be the global optimal solution. So there may always has a solution that has a better quality. And when the size of graph is large, it may be time-consuming.

5 Empirical Evaluation

5.1 Branch and Bound

5.1.1 Platform description.

- CPU: Apple Silicon M1Pro
- RAM: 16.0 GB
- Language: Python 3
- Editor: VS Code

5.1.2 Experiment procedure. Ran program with fixed 600 cut-off time on 11 provided .graph files and obtain runtime and results. BnB algorithm is depicted in Algorithm 1, and the detailed work-flow description for the BnB implementation can be found in the README file and the coding files.

5.1.3 Evaluation criteria. We have calculated the percentage of error against Optimal solution provided. (Please see Table 1. for results) In the table, Relative Error(RelError)is defined as $RelError = (VCValue - OPT)/OPT$, where OPT is the number of vertices in the optimal vertex cover, and this relative error is used as a metric (Provided by the professor) to evaluate the quality of solutions.

5.1.4 Results(plots and tables). As can be seen in Figure14. *BnB* algorithm has a very good performance to have very small relative error. And also, compared to other algoritnms in Table 1 results, the BnB takes a not short time to travel the entire search space and run longer in larger graph data sets. All presented data are obtained within 600 second time limit. More discussions of BnB method will be found in Discussion section below.

5.2 Approximation

5.2.1 Platform description.

- CPU: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- RAM: 16.0 GB

Table 1. Summary of Performance of All Algorithms

	BnB	Approx	LS1	LS2
as-22july06.graph				
Time	53.81	11.79	19.52	707.23
VC Value	3312	6036	3303	3327
RelError	0.002725	0.82	0.00003	0.0073
delaunay_n10.graph				
Time	1.54	0.18	37.12	0.40
VC Value	739	910	703	745
RelError	0.051209	0.29	0.00057	0.05974
email.graph				
Time	0.32	0.25	1.54	2.05
VC Value	605	828	594	612
RelError	0.018519	0.39	0.00000	0.03030
football.graph				
Time	0.15	0.01	0.01	0.01
VC Value	95	106	94	97
RelError	0.010638	0.13	0.00000	0.03192
jazz.graph				
Time	105.34	0.03	0.04	0.03
VC Value	158	180	158	160
RelError	0.000	0.14	0.00000	0.01266
karate.graph				
Time	0.00	0.00	0.00	0.00
VC Value	14	22	14	14
RelError	0.0000	0.57	0.00000	0.00000
netscience.graph				
Time	0.54	0.24	0.03	0.95
VC Value	899	1208	899	899
RelError	0.000	0.34	0.00000	0.00000
power.graph				
Time	5.85	1.90	38.61	12.20
VC Value	2272	3606	2203	2290
RelError	0.031321	0.64	0.00000	0.03949
star.graph				
Time	39.33	28.97	940.77	105.76
VC Value	7366	10202	6944	7222
RelError	0.067227	0.48	0.00610	0.04636
star2.graph				
Time	45.34	27.57	398.15	443.91
VC Value	4677	6794	4546	4855
RelError	0.067227	0.50	0.00084	0.06891
hep-th.graph				
Time	16.16	5.64	20.52	49.94
VC Value	3947	5800	3926	3940
RelError	0.005349	0.48	0.00003	0.00357

- Language: Python 3.7.7
- Editor: VS Code

5.2.2 Experiment procedure. Ran program with random seed set to 1045 on 11 provided .graph files and obtain runtime and results.
Bash file provided to run *Arppox* on all files.

5.2.3 Evaluation criteria. Calculate percentage error against Optimal solution provided. (Please see Table 1. for results)

5.2.4 Results(plots and tables). As can be seen in Figure 14. and not surprisingly, the *Approx* algorithm has the largest percentage errors than the others. Since every time we select a random edge, two vertices are added, and it's only bounded by twice the size of the optimal solution, therefore the results with large percentage errors are as expected.

5.3 Local Search: Simulated Annealing

5.3.1 Platform description. The platform information for this study is described as follows:

- CPU: 9th Gen Intel(R) Core(TM) i5-9600K@3.7GHz.
- RAM: 16.0GB
- Language: Python

5.3.2 Experiment Setup. Follow the guidelines, this study utilized the datasets extracted from the 10th DIMACS challenge. For each dataset, 10 different cases with random seeds ranging from 1 to 10 is being applied to generate the results and used for later Qualified Runtime Distribution (QRTD) and Solution Quality Distribution (SQD) study.

For each dataset, a cutoff time is selected and as discussed in previous section, datasets which are struggling to get to the global optimum will use longer cutoff time while datasets with smaller size will apply smaller cutoff time for saving computational time. Detailed setting of cutoff time for each dataset can be found in Table. 2.

Table 2. Simulated Annealing Experiment Results

Dataset	**Time(s)**	**VC**	**Relative Error**	**Cutoff (s)**
as-22july06	19.52	3303	0.00003	100
delaunay-n10	37.12	703	0.00057	100
email	1.54	594	0.00000	10
football	0.01	94	0.00000	1
hep-th	20.52	3926	0.00003	100
jazz	0.04	158	0.00000	1
karate	0.00	14	0.00000	1
netscience	0.03	899	0.00000	1
power	38.61	2203	0.00000	100
star	940.77	6944	0.00610	1000
star2	398.15	4546	0.00084	500

5.3.3 Evaluation criteria and results. The evaluation metrics used for SA algorithm includes the averaged computational time, averaged solved vertex cover size and its relative error compared with the provided real optimum. Based on the results demonstrated in Table. 2, SA algorithm can quite easily to achieve a good result within a short amount of time for the majority of the cases. *star.graph* and *star2.graph* are the only two instances to take a while to achieve a good solution. Based on our later analysis, we will show it can achieve a relatively acceptable result even within a short amount of time. The long running time showed here is purely because we want to achieve a super good quality solution so we run extra time for these cases.

The relative error utilized here is defined in Eq. 2:

$$RelativeError = \frac{Alg - OPT}{OPT} \tag{2}$$

Besides these common criteria, for local search algorithm, we also applied the Qualified Runtime Distribution (QRTD) and Solution Quality Distribution (SQD) criteria to study the performance of our proposed SA algorithm on two selected datasets: *power.graph* and *star2.graph*.

Figure. 2 and Figure. 3 are the QRTD plots, while Figure. 4 and Figure. 5 are the SQD plots for these two datasets.

For the *power.graph*, we selected 0.0%,0.1%,0.2%,0.3% and 0.4% as the q^* for the QRTD analysis. Here 0.0% means the solution is exactly the same of the real optimum. For the *star2.graph*, we selected 0.07%,0.09%,0.11%,0.13% and 0.15% as the q^* for the QRTD analysis.

Based on these two plots, we can easily find that when narrowing the allowable gap between the local search optimum and the real optimum, it will take more time for the algorithm to achieve such accuracy. However, if we can loose the requirement of our solution quality, SA algorithm can pretty easily to achieve a good solution within a short amount of time. There will always have a trade-off between the computational time and the solution quality.

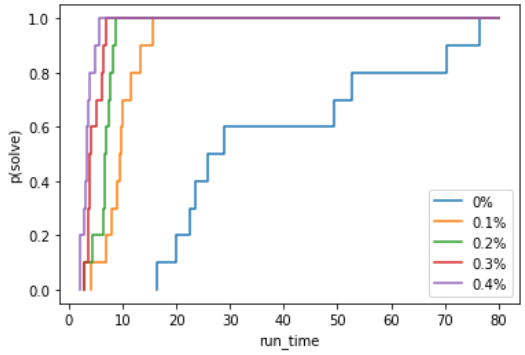

Figure 2. QRTD(Simulated Annealing): power

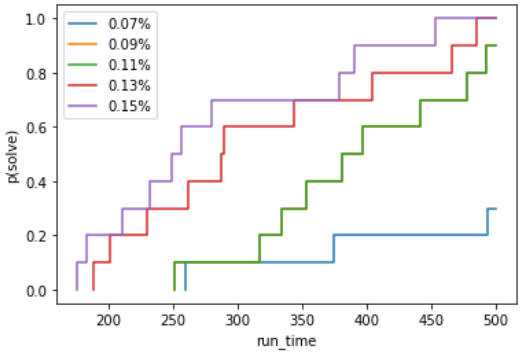

Figure 3. QRTD(Simulated Annealing): star2

For the SQD analysis, we choose 5s, 10s, 15s, 20s and 25s for the *power.graph* while for the *star2.graph*, we choose 300s, 350s, 400s, 450s and 500s.

Based on Figure. 4 and Figure. 5, we can easily observe that with the increasing of computation time, more cases will reach high quality solution as the purple line is always above the rest lines if we fix a solution quality.

Therefore, based on both the QRTD and the SQD plots, we can have the following two conclusions:

- More computation time will benefit the solution quality.
- If we don't have a strict solution quality requirement, SA algorithm can easily achieve a good solution within a short amount of time.

These observations also explain the cutoff settings we used in Table. 2. If we don't need to achieve a super high quality solution, the required computation time for *star.graph* can be lowered.

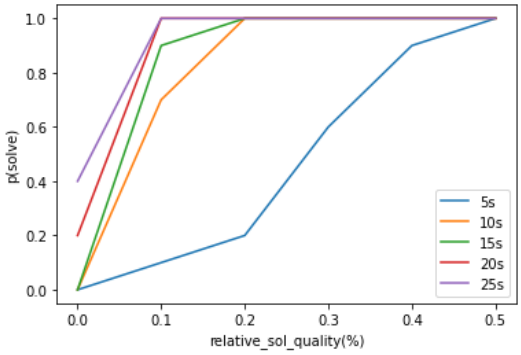

Figure 4. SQD(Simulated Annealing): power

Moreover, since we utilized 10 random seed ranging from 1 to 10 for each dataset, we can also study the randomness effect on our computational time.

Figure. 6 and Figure. 7 are the boxplots of the two graphs. Based on these figures, we can see there is non-negligible effects of randomnes on the computational time. For the

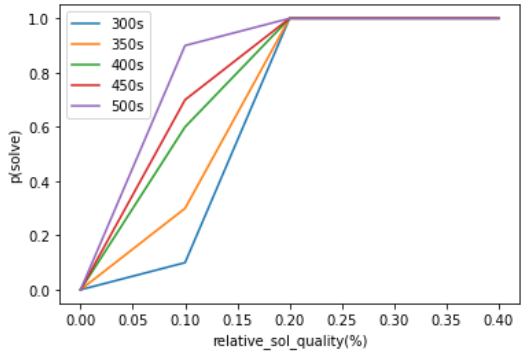

Figure 5. SQD(Simulated Annealing): star2

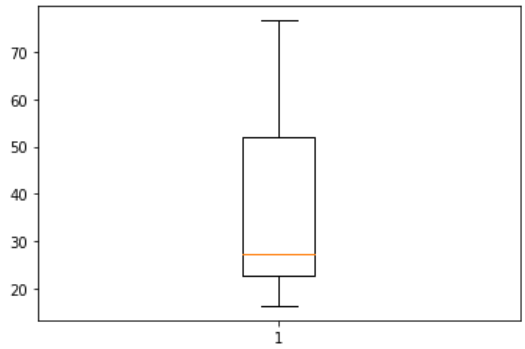

Figure 6. Box plot (Simulated Annealing): power

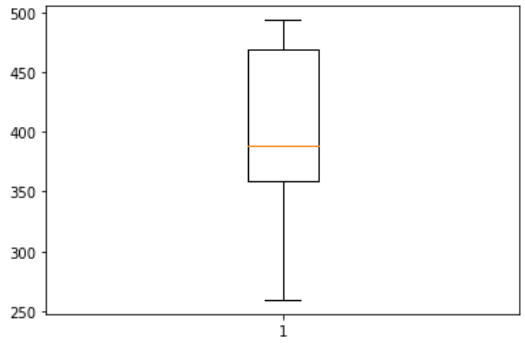

Figure 7. Box plot (Simulated Annealing): star2

power.graph, even though the mean is around 30s, the longest case took more than 70s to finish while the shortest only took more than 10s. The variance is not negligible, but the majority of the cases will reach final solution within 30-50s. For the *star2.graph*, it showed similar trend as some cases can reach its local search solution less than 300s while some cases will take almost 500s which is near the cutoff time of our settings.

Therefore, we can see the effects of randomness is not negligible.

5.4 Hill Climbing

5.4.1 Platform description. The platform information for this study is described as follows:

- CPU: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz.
- RAM: 16.0GB
- Language: Python

5.4.2 Experiment Procedure. The programming language we use is Python3. The 10 random seeds chosen are from 1 to 10. After we get results of these 10 samples, we draw the QRTDs, SQDs, Box plots based on them.

5.4.3 Evaluation Criteria and Results. Calculate related error against Optimal solution provided. And the detailed results are in the Table1. As shown in the table, hill climbing has the highest relative error and the speed is slow. And other results for local search are provided in figures. The graph chosen are power and star2. Figure. 8 and Figure. 9 are the QRTD plots and Figure. 10 and Figure. 11 are the SQD plots for these two datasets. And the running time and relative error we choose are based on the experimental results. Figure. 12 and Figure. 13 are figures about box plots. From these figures we can see the relationship between running time and relative error. Also the change of random seeds will influence the results.

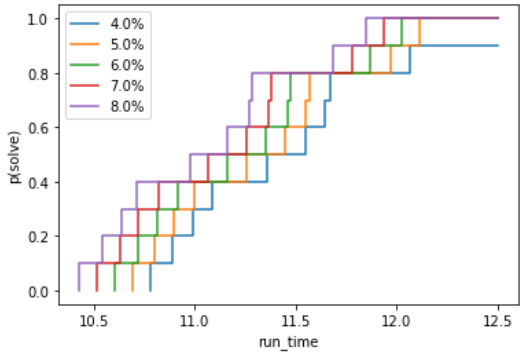

Figure 8. QRTD(Hill Climbing): power

6 Discussion

For **BnB** algorithm, in this project, based on the results of Table 1, although it has good performance of accuracy, however, it takes long time to run, for some extreme cases, it needs over 1 hour time bound, it is time-consuming, especially compared to the Approx method. We have to think about how we can make it more faster, in this case, we computed the lower bound as the ratio of number of remaining edges in the subgraph to the maximum node degrees, probably we can estimate a better lower bound to make the calculation more efficient. In the future, we can also consider advanced techniques to improve the backtracking or other parts to

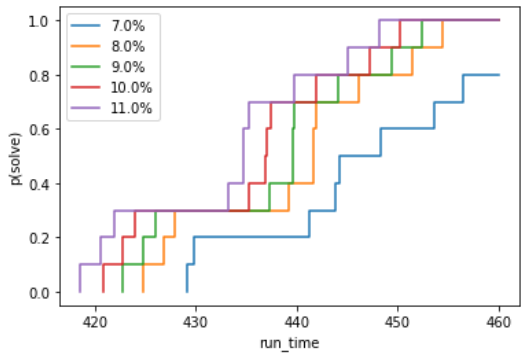

Figure 9. QRTD(Hill Climbing): star2

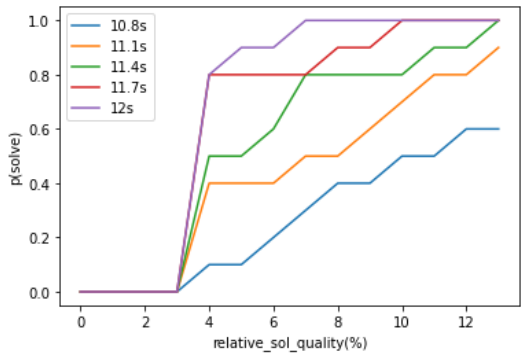

Figure 10. SQD(Hill Climbing): power

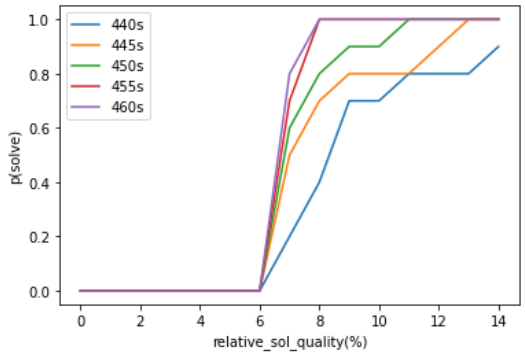

Figure 11. SQD(Hill Climbing): star2

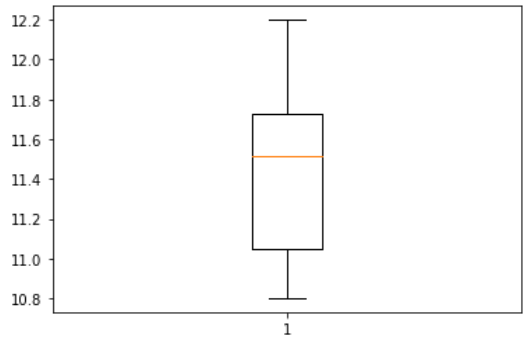

Figure 12. BOX(Hill Climbing): power

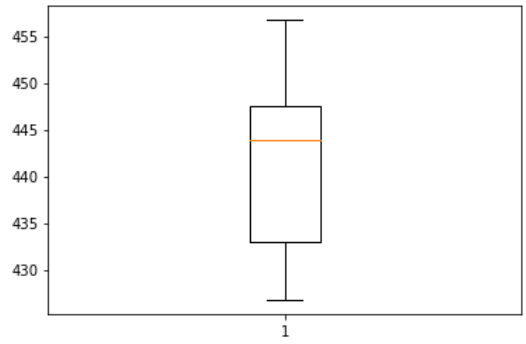

Figure 13. BOX(Hill Climbing): star2

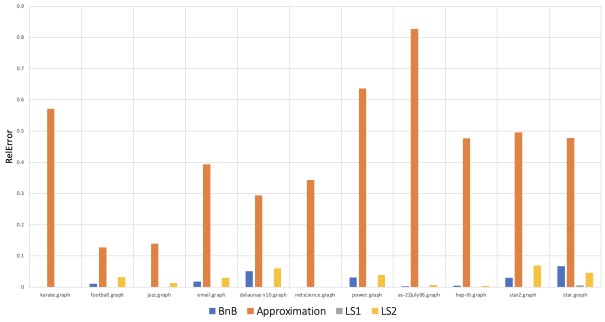

Figure 14. Comparison of Relative Error for All Algorithms

improve the speed of filtering the candidates to speed up the whole process.

For **Approx**, the running time used to run this algorithm is significantly faster (so the cutoff of 600s is trivial in this case) in any case that the other three with the cost of relatively low accuracy. As mentioned earlier, the best approximation we can reach so far is 2. As expected, the result table shows much larger rel errors compare to the rest. If we were to looking for a better ratio, we have to turn to other type of algorithms.

For **simulated annealing**, it can achieve pretty promising results compared with other algorithms. Though some of them require a large amount of time, it also produces a high quality solution compared with others. As discussed in the former section, based on the QRTD and SQD plots, if we lower the cutoff time, these datasets may also generate acceptable solution within a short amount of time.

For **hill climbing**, although it finds the solution that has a small value of relative error, its error value is still larger than other algorithms. It reflects one of the disadvantage of hill climbing. It is finding a local peak, but not a global optimal solution. As analyzed before, the time complexity of hill

climbing is $O(|V||E|)$. When the graph size becomes larger, the running time of the algorithm will increase quickly. It matches the running time of our experiments.

7 Conclusion

In conclusion, we implemented four algorithms to solve the MVC problem: Branch and Bound, Approximation, Simulated Annealing and Hill Climbing algorithms. And we evaluate the algorithm based on their accuracy and other performance like time complexity. If correctness is considered as the most important factor, Bnb algorithm will be the best choice. For the two local search algorithms, simulated annealing is a better choice compared with hill climbing. And in general, approximation may be an appropriate algorithm to solve the MVC, it keeps a small relative error and it takes the least time to find the solution. What's more, it is easy to be implemented.

In the real-world case, the selection of the algorithm will depend on the trade-off between accuracy and the speed. In the future works, we can generate the metrics to help people evaluate the criteria to choose the best option of the algorithm to satisfy people's needs. Of course, we can also implement the advanced techniques to improve the speed or accuracy on our algorithms to encounter the more complex problems.

References

[1] CS 105. 2005. Approximation Algorithms: Vertex Cover. http://tandy.cs.illinois.edu/dartmouth-cs-approx.pdf.

[2] Sangeeta Bansal and Ajay Rana. 2014. Analysis of various algorithms to solve vertex cover problem. *International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN* (2014), 2278–3075.

[3] Walid Ben-Ameur. 2004. Computing the initial temperature of simulated annealing. *Computational optimization and applications* 29, 3 (2004), 369–385.

[4] Shaowei Cai. 2015. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[5] Shaowei Cai, Kaile Su, and Qingliang Chen. 2010. EWLS: A new local search for minimum vertex cover. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

[6] 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 (2013), 687–716.

[7] Shaowei Cai, Kaile Su, and Abdul Sattar. 2011. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artificial Intelligence* 175, 9-10 (2011), 1672–1696.

[8] Vladimír Černỳ. 1985. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications* 45, 1 (1985), 41–51.

[9] Subham Datta. 2022. Branch and Bound Algorithm. https://www.baeldung.com/cs/branch-and-bound#:~:text=Branch.

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

[11] CSE 6140 Class @ GT. 2022. CSE6140 Fall 2022 Project Minimum Vertex Cover. (2022), 2.

[12] Alexander K Hartmann and Martin Weigt. 2003. Statistical mechanics of the vertex-cover problem. *Journal of Physics A: Mathematical and General* 36, 43 (2003), 11069.

[13] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

[14] Mingyang Li Junping Zhou Luzhi Wang, Shuli Hu. 2019. An Exact Algorithm for Minimum Vertex Cover Problem. *Mathematics* (2019), 3.

[15] Miym. 2009. Minimum-size vertex cover. Retrieved February 22, 2009 from https://commons.wikimedia.org/wiki/File:Minimum-vertex-cover.svg

[16] 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.