Minimum Vertex Cover Project Report

CSE 6140 Fall 2022

Oojas Salunke
osalunke3@gatech.edu
Georgia Institute of Technology

Bao Li
libao@gatech.edu
Georgia Institute of Technology

Bilal Mufti
bmufti3@gatech.edu
Georgia Institute of Technology

Alexander A Reyna
areyna8@gatech.edu
Georgia Institute of Technology

1 INTRODUCTION

In this report, we solve minimum vertex cover, an NP-complete problem using different approaches. Minimum vertex cover is a problem of finding a set of vertices that at least include one end point of all the edges in the graph. The problem has numerous applications in areas ranging from computer science, and computational biology to supply chain optimization. We used an exact method, an approximation algorithm, and two local search approaches to solve the problem. The quality of each approach and the computational cost needed to achieve that quality was determined by finding the relative error and comparing computational time.

The exact branch and bound algorithm guarantee optimality with a high computational cost. Results indicate that for smaller and medium-sized graphs, the exact branch and bound algorithm was most accurate but took substantially more computational time. For larger graphs, this method struggled to find an accurate vertex cover. On the other hand, the approximation and local search algorithms do not guarantee optimality. However, these approaches are able to find a solution within a reasonable time and with a small error. The approximation algorithm and two local search algorithms: hill climbing and simulated annealing were also able to find a valid solution for larger graph sets within an acceptable relative error. The study shows that there is a cost vs accuracy trade-off offered by these algorithms and the choice of the algorithm to be used to solve minimum vertex cover problems should be made based on accuracy requirement and availability of computational budget.

2 PROBLEM DEFINITION

The minimum vertex cover problem is a well-known NP-complete problem. Given a graph $G = (V, E)$, a vertex cover is a subset $C \subseteq V$ such that every edge in E is incident to at least one vertex in C i.e., $\forall(u, v) \in E : u, v \in C$. The minimum vertex cover problem is to find a vertex cover C of minimum size [4].

3 RELATED WORK

Minimum vertex cover (MVC) has been used in different industries to solve complex problems. Computer scientists have successfully used the MVC algorithm to simulate the propagation of stealth worms on large computer networks. Through this, they were able to develop techniques and strategies to protect the network against virus attacks in real-time [2]. MVC can also be used to solve simple real-world problems. The placement of CCTV cameras inside a Smithsonian museum such that all the exhibits are covered is an example of a real-world problem that can easily be solved using the MVC.

Due to the importance of the MVC problem, a number of different ways have been proposed to solve it. Work by Delbot et.al [1] gives a detailed discussion on the analytical and experimental comparison of six different vertex cover problem algorithms. Based on the results they suggested a "practical hierarchy" of the algorithms. A lot of recent work has focused on developing new methods to improve exact algorithms such as branch and bound [3, 5]. These studies focus on developing improved lower bounds to build faster algorithms.

4 ALGORITHMS

In this section, we will give a detailed description of all the implemented algorithms along with their pseudo-codes.

4.1 Branch and Bound

4.1.1 Introduction and Description: Branch and bound, an exact algorithm is frequently used to solve combinatorial optimization problems. Due to the exact nature of this algorithm, if given sufficient computational time it is guaranteed to find the optimal solution. The branch and bound algorithm search through the exponentially sized search space for all possible solutions. As a result, the time required to obtain a solution is often exponential in time. The algorithm works for small-size graph problems or for graphs that have an underlying special structure. However, for most complex graph problems the computational time needed to get an optimal solution is intractable.

The algorithm works by selecting the node with the highest degree and deciding whether it is to be added to the vertex cover or not. The decision is made using the 2-approximation algorithm. If the decision is made to include it, we remove the vertex and all its edges from the graph. We select all the neighbors of the vertex if the decision is made to not include it in the vertex cover. The pseudo-code is shown in Algorithm 1. We initialize a global upper bound as the number of vertices in the graph. We update this bound with the smallest vertex cover that has been found so far by the algorithm. We also find a lower bound for each partial solution's possible outcome.

4.1.2 Time and Space Complexity: The algorithm prunes the nodes for which no better solution is possible. But still, for the worst-case scenario the branch and bound algorithm would explore every possible subset of vertices. The number of possible solutions is asymptotically exponential in the number of vertices in the graph. As a result, the time and space complexity for this algorithm is $O(2^{|V|})$

Algorithm 1 Branch and Bound (BnB)

```
function BnB($G$,$cutoff$)
    $C \leftarrow \emptyset$
    $bound \leftarrow |V|$
    while $time < cutoff$ and solution not found do
        select a vertex $u$ with maximum degree
        choose whether to add $u$ in $C$ using 2-approximation
        $LB = |C| + A$
        if $LB < bound$ then
            $bound = LB$
        end if
    end while
    return $C$
end function
```

4.2 Approximation Algorithm

4.2.1 Introduction and Description: To solve an NP-complete problem, we can use a heuristic algorithm to find a solution with a good approximation guarantee in polynomial time. To solve the minimum vertex cover problem, we can use the greedy algorithm. The greedy algorithm randomly selects a vertex and adds it to the vertex cover if it is not covered by the current vertex cover. The algorithm will stop when all the vertices are covered. The algorithm is as follows:

Algorithm 2 Approximation Algorithm

```
function Approx($G$)
    $E' \leftarrow$ edges of $G.E$
    $C \leftarrow \emptyset$
    while $E' \neq \emptyset$ do
        let $(u, v) \leftarrow$ be an arbitrary edge in $E'$
        $C \leftarrow C \cup \{u, v\}$
        remove all edges incident to $u$ and $v$ from $E'$
    end while
    return $C$
end function
```

We can use the approximation ratio to measure the quality of the solution found by the algorithm. The approximation ratio is a measure of how close the solution found by the algorithm is to the optimal solution. We say that the algorithm has an approximation ratio of $\rho(n)$ if, for any instance of size n, the cost C of the solution found by the algorithm is within a factor of $\rho(n)$ of the optimal cost C^*.

$$\max\{\frac{C}{C^*}, \frac{C^*}{C}\} \leq \rho(n) \tag{1}$$

Let $A \subset E$ be the edges chosen by the algorithm. An endpoint of each edge in A must be in C^*, and the number of edges in A is at most $|C^*|$. Therefore, the cost of the solution found by the algorithm is at most $2|C^*|$. The cost of the optimal solution is $|C^*|$. Therefore, the approximation ratio is at most 2. Hence:

$$|C^*| \leq |A| = \frac{|C|}{2} \quad \rightarrow 1 \leq \rho(n) \leq 2 \tag{2}$$

The approximation algorithm is not efficient enough. We can improve the algorithm by using the maximum degree greedy algorithm. The Maximum Degree Greedy (MDG) algorithm is an adaptation of the classical greedy algorithm for the set cover problem. It always selects the vertex with the highest degree and adds it to the vertex cover if it is not covered by the current vertex cover. The algorithm will stop when all the vertices are covered. The approximation algorithm using the maximum degree greedy algorithm is as follows:

Algorithm 3 Maximum Degree Greedy (MDG)

```
function MDG($G$)
    $C \leftarrow \emptyset$
    while $E \neq \emptyset$ do
        select a vertex $u$ with maximum degree
        $C \leftarrow C \cup \{u\}$
        $V \leftarrow V - \{u\}$
    end while
    return $C$
end function
```

The worst-case approximation ratio is $H(\Delta)$, with $H(n) = \sum_{i=1}^{n} \frac{1}{i}$, which is the harmonic series ($H(n) \approx \ln n + 0.57$ when n is large) and Δ is the maximum degree of the graph [1]. The limit of the expectation of the approximation ratio is $1 + e^{-2} \approx 1.13$ [1] for this problem.

4.2.2 Time and Space Complexity: For both algorithms, the worst case is that the greedy algorithm will try all the edges in the graph. Since there is a nested loop in both algorithms, one is the loop of the edges and the other is the loop of edge removal, the time complexity is $O(|E|^2)$ and the space complexity is $O(|V| + |E|)$ for both algorithms.

4.3 Local Search: Hill Climbing (LS1)

4.3.1 Introduction and Description: Hill climbing is a local search algorithm, that greedily picks degree of the node, as it always selects the best solution from the current neighborhood of solutions. This means that it is very likely to get stuck in local optima, where the current solution is already the best possible solution in its neighborhood, but there may be a better solution elsewhere.

This implementation's purpose is to keep deleting vertices till a minimum is found. The algorithm starts with the entire set of vertices and based on the priority queue it starts to delete vertices. A list of vertices with the smallest priority queue is selected, and then a random vertex is picked from this list. Then the algorithm then starts by deleting this node from the priority queue and the vertex cover and checks if deleting this node will result in a valid vertex cover. If a valid vertex cover is not reached then it will add the vertex back to the set. This process is repeated until the priority queue is empty or the cutoff time is reached.

Algorithm 4 Hill Climbing (LS1)

```
function HILL CLIMBING($G$)
    $cover$ ← list of all nodes in G
    $pq$ ← list of all nodes in G
    while $pq \neq \emptyset$ do
        $l$ ← list with nodes with the smallest and same priority
        $node$ ← random node from list l
        delete $node$ from $pq$
        delete $node$ from $vc$
        if is-vertex-cover(vc)==False then
            add $node$ to $vc$
        end if
    end while
    return $vc$
end function
```

4.3.2 Time and Space Complexity: The time complexity will be dominated by the input graph and the random order of the priority queue. Worst case it will go through all the vertices and check if it is a valid vertex cover or not, which would result in a time complexity of $O(V^2)$. As the input graph will dominate time complexity, therefore the upper bound for the run time for this algorithm will be defined by its cutoff time.
For the space complexity we have our input graph of $|V|$ vertices and $|E|$ edges, we then have our variables which store our vertex cover (vc) with at most $|V|$ vertices and the priority queue which also stores at most $|V|$ vertices, resulting in $O(|V| + |E| + |V| + |V|)$ space taken.

4.4 Local Search: Simulated Annealing (LS2)

4.4.1 Introduction and Description: Simulated annealing, on the other hand, is a probabilistic algorithm that uses the concept of "annealing" in metallurgy to avoid getting stuck in local optima. In simulated annealing, the algorithm will occasionally accept a solution that is not the best in the current neighborhood, with a probability that decreases as the number of iterations increases. This allows the algorithm to explore a wider range of solutions and makes it less likely to get stuck in local optima.

This implementation's purpose is to iteratively improve based on the probability to take a step in the opposite direction. Early on, the probability to take steps in the opposite direction is high, but as we near the end it takes less risky steps and tries to converge on a solution. The probability is calculated as follows:

$$p = exp\{-\frac{(1 + G.degree(v))}{current\_temp}\} \tag{3}$$

The goal is to move in an opposite direction, based on the above probability, to get out of a possible local minimum. This process is repeated till the current temperature becomes lesser than the final temperature or if the cutoff time is reached.

Algorithm 5 Simulated Annealing (LS2)

```
function SA($G$)
    $end\_temp$ ← 0.00001
    $current\_temp$ ← 1
    $alpha$ ← 0.999
    $vc$ ← list of nodes in G based on MDG
    while $current\_temp > end\_temp$ do
        $u$ ← random node from G
        if v in vc then
            $vc^*$ ← vc
            $vc^*.remove(v)$
            if is-vertex-cover(vc)==False then
                $vc = vc^*$
            end if
        else
            $p = exp\{-\frac{(1+G.degree(v))}{current\_temp}\}$
            if p > rand(0,1) then
                $vc.append(v)$
            end if
        end if
        $current\_temp * alpha$
    end while
    return $vc$
end function
```

4.4.2 Time and Space Complexity: The time complexity will be proportional to the input graph and the number of edges in that graph, and also to the construction of the initial solution $O(V^2 \log V)$. Fundamentally, we are deciding to add or not, a randomly picked node from graph G. If we add it, this will simply take $O(1)$ time, and if we remove it we will have to check if it is a valid vertex cover or not. Therefore, the overall cost is $O(V^2 \log V) + O(V + E)$. As we also have a cutoff time, this will finally define the upper bound for the rum time for this algorithm.
For the space complexity, we have our input graph of $|V|$ vertices and $|E|$ edges, a variable to keep track of the vertex cover of size $|V|$, and we need to iterate over all the edges of size $|E|$. Therefore, resulting in $O(|V| + |E| + |V| + ||E)$ space taken.

5 EMPIRICAL EVALUATION

In this section, we would analyze the results for all implemented algorithms by finding the relative error and computational time. For local search algorithms, we will also generate quality run-time distribution (QRTDs), solution quality distribution (SQDs), and run-time box plots.

5.1 Branch and Bound

Platform:

(1) CPU: 3.0 GHz Intel Core i7
(2) RAM: 16 GB DDR3
(3) System: Window 11
(4) Language: Python 3.10

The results shown in Table.1 were obtained for a cutoff time of 1500 sec (25 min). A "-" means that the branch and the bound algorithm were not able to obtain a vertex cover solution. For small graphs, the algorithm was able to find an optimal solution with a low relative error. As we increase the number of vertices in the graph the algorithm takes relatively more time to find an optimal solution but the relative error remained low. For large-sized graphs, the branch and bound algorithm fail to find a valid vertex cover within the cutoff time limit.

Table 1: Branch and Bound Algorithm - cutoff time 1500 s

Dataset	Time (s)	VC Value	RelErr
jazz	0.84	159	0.0063
karate	0.01	14	0
football	0.15	95	0.011
as-22july06	1195.5	33305	0.000606
hep-th	1157.65	3933	0.001783
star	> 1500	-	-
star2	> 1500	-	-
netscience	18.05	899	0
email	13.94	602	0.0135
delaunay_n10	13.83	725	0.031
power	252.71	2222	0.008625

Among the vertex cover solutions found by branch and bound, the maximum relative error was 0.031294 which shows its advantage in terms of accuracy. An interesting observation can be made regarding the *netscience* graph. Despite the relatively large graph size, the branch and bound algorithm were able to find the optimum VC with a relative error of 0. This can be due to the inherent structure of the graph which allowed the algorithm to find the exact VC.

Table 2: Branch and Bound Algorithm - Modified

Dataset	BnB t(s)	BnB-Modified t(s)
jazz	0.84	0.51
karate	0.01	0.0036
football	0.15	0.097
as-22july06	1195.5	202.83
hep-th	1157.65	93.48
star	> 1500	306.43
star2	> 1500	278.73
netscience	18.05	3.93
email	13.94	3.24
delaunay_n10	13.83	2.92
power	252.71	31.52

Since the approximation algorithm will be called for each *while* loop, it acts as the bottleneck of the algorithm and slows it down. We can speed up the algorithm by using the approximation algorithm from *networkx* library which is implemented in C++. Using this approximation algorithm in C++ makes our algorithm run much faster. The comparison of the two algorithms is shown in Table 2. From the table, we can see that the *networkx* implementation is much faster than our implementation. We can use the *networkx* implementation to speed up the algorithm by a factor of 5 to 10 times.

5.2 Approximation Algorithm

Platform:

(1) CPU: Apple M1 Max
(2) RAM: 64 GB DDR4
(3) System: macOS 13
(4) Language: Python 3.10.8

From the description in the previous section 4.2. The approximation ratio of the original approximation algorithm is no more than 2, and the approximation ratio of the maximum degree greedy algorithm is no more than 1.13. Since $\rho(n) = C/C^*$ and the relative error is $(C - C^*)/C^*$, we can use the relative error to measure the quality of the solution found by the algorithm using the following equation:

$$\rho(n) = \frac{C}{C^*} = 1 + \frac{C - C^*}{C^*} = RelErr + 1 \tag{4}$$

Therefore, we can quickly calculate the approximation ratio from the relative error by adding 1 to the relative error.

Table 3: Approximation Algorithm

Dataset	Time (s)	VC Value	RelErr	$\rho(n)$
jazz	0.014	184	0.16	1.16
karate	0.00023	20	0.42	1.42
football	0.0033	112	0.19	1.19
as-22july06	19.73	5688	0.72	1.72
hep-th	7.06	5606	0.43	1.43
star	31.45	10748	0.56	1.56
star2	31.69	6806	0.50	1.5
netscience	0.27	1214	0.35	1.35
email	0.196	816	0.37	1.37
delaunay_n10	0.19	958	0.36	1.36
power	2.75	3722	0.70	1.7

Table 4: Maximum Degree Greedy Algorithm

Dataset	Time (s)	VC Value	RelErr	$\rho(n)$
jazz	0.012	160	0.013	1.013
karate	0.00028	14	0.00	1.00
football	0.0039	96	0.021	1.021
as-22july06	23.12	3312	0.0027	1.0027
hep-th	9.33	3947	0.0053	1.0053
star	26.29	7366	0.067	1.067
star2	26.35	4677	0.03	1.03
netscience	0.39	899	0.00	1.00
email	0.22	605	0.019	1.019
delaunay_n10	0.24	740	0.053	1.053
power	3.32	2272	0.031	1.031

From Table 4, we can see that the approximation ratio of the maximum degree algorithm is always less than 1.13, and from Table 3 the approximation ratio of the original approximation algorithm is always less than 2. The maximum degree algorithm is much more accurate than the original approximation algorithm. At the same time, the run time of the maximum degree algorithm is also faster than the original approximation algorithm. One reason is when implementing the MDG algorithm, the group used the lambda function to speed up the process find maximum degree node.

5.3 Local Search

Platform:

(1) CPU: 3.00GHz Intel i9-10980XE
(2) RAM: 128 GB DDR4
(3) System: Linux 20.04
(4) Language: Python 3.8.10

Table 5: Hill Climbing (LS1) results for a cutoff time 1500 s

Dataset	Time (s)	VC Value	RelErr
jazz	0.063	160	0.013
karate	0.0012	14	0.00
football	0.014	95	0.01
as-22july06	394.4	3328	0.0078
hep-th	33.19	3937	0.00025
star	103.0	7220	0.046
star2	203.8	4849	0.067
netscience	1.069	899	0.00
email	0.89	612	0.030
delaunay_n10	0.698	741	0.054
power	12.29	2273	0.031

Table 6: Simulated Annealing (LS2) - cutoff time 1500 s

Dataset	Time (s)	VC Value	RelErr
jazz	3.19	160	0.013
karate	0.31	14	0.00
football	1.023	96	0.021
as-22july06	53.12	3316	0.0039
hep-th	33.87	3940	0.0035
star	97.6	7100	0.028
star2	53.85	4651	0.024
netscience	5.79	899	0.00
email	5.45	602	0.013
delaunay_n10	4.91	730	0.038
power	14.78	2247	0.019

Hill climbing and simulated annealing local search algorithms were run for 10 different random seeds with a cutoff time of 1500 sec. Results for both algorithms are shown in table 5 and table 6. The results were obtained after averaging the different random seed observations. Both local search algorithms were able to find valid minimum vertex cover solutions for all data sets. The relative error for simulated annealing was much lower when compared with hill climbing. However, hill climbing was much faster to execute. The better accuracy and higher run time of simulated annealing is a consequence of initialization by using a maximum degree greedy algorithm.

Figures 1 and 2 below show the QRTD plots for the power and star2 graphs obtained using the hill climbing local search algorithm. The figure shows that higher relative solution quality values converge faster to the optimal solution plus the relative solution quality, or that a higher proportion of the runs reach the relative optimal solution sooner. All of the instances of the power graph using the hill climbing algorithm reached the relative optimal solution of $q^* = 0.01$ by 12.9 seconds, while all of the instances of the star2 graph were able to reach the relative optimal solution found of $q^* = 0.01$ by 205 seconds, as the runtimes for the hill climbing instances using star2 ranged from 202 to 208 seconds. For the SQD plots of the graphs using the hill climbing algorithm, the instances were found to converge to lower relative solution quality values for greater time values, which is explained by the algorithm having a smaller optimality gap at later timesteps. The box plots for the computational time in Figures 5 and 6 show that for both graphs the hill climbing algorithm is relatively consistent in computational time. We can observe similar trends for QRTD, SQD, and box plots for simulated annealing.

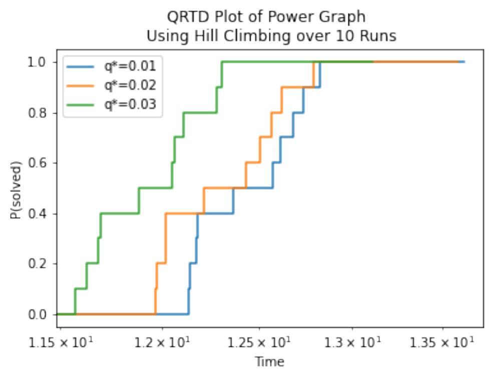

Figure 1: QRTD for Power, Hill Climbing

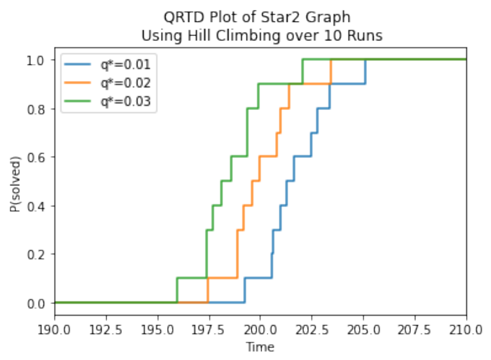

Figure 2: QRTD for Star2, Hill Climbing

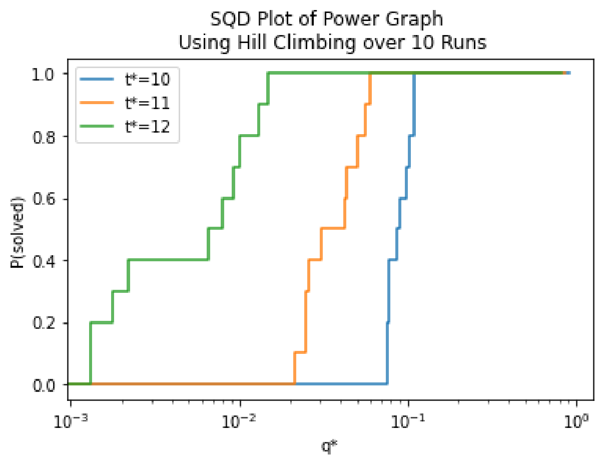

Figure 3: SQD for Power, Hill Climbing

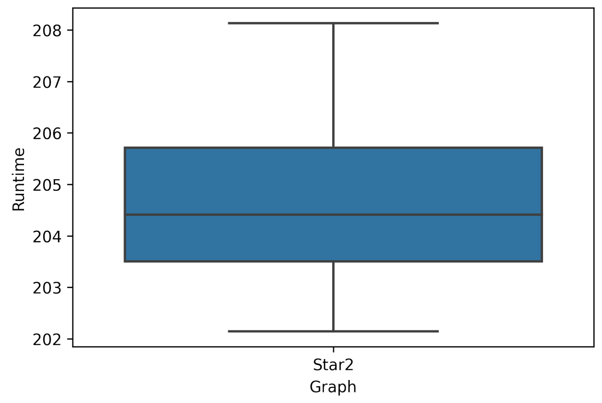

Figure 6: Boxplot for Star2 Graph, Hill Climbing

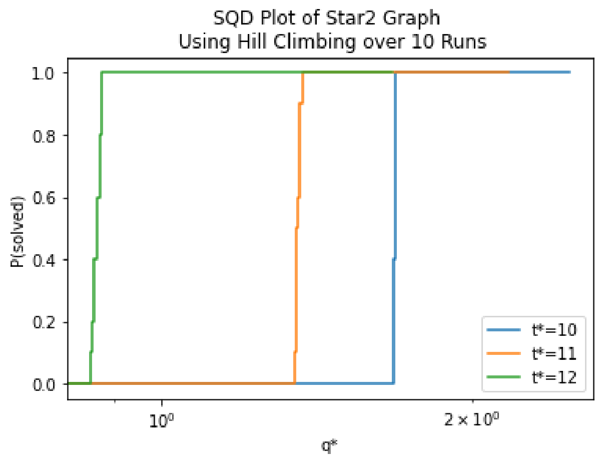

Figure 4: SQD for Star2, Hill Climbing

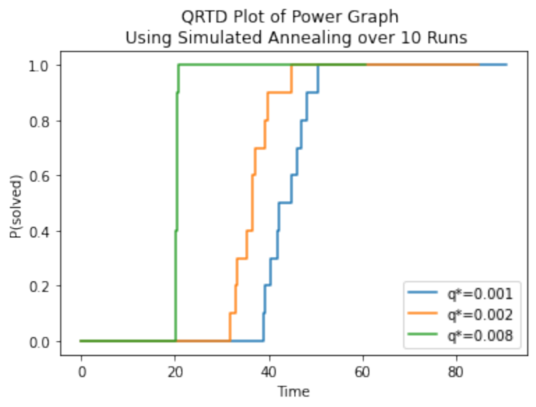

Figure 7: QRTD for Power Graph, Simulated Annealing

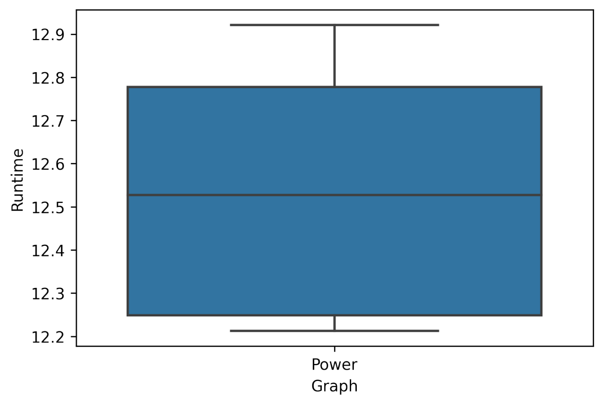

Figure 5: Boxplot for Power Graph, Hill Climbing

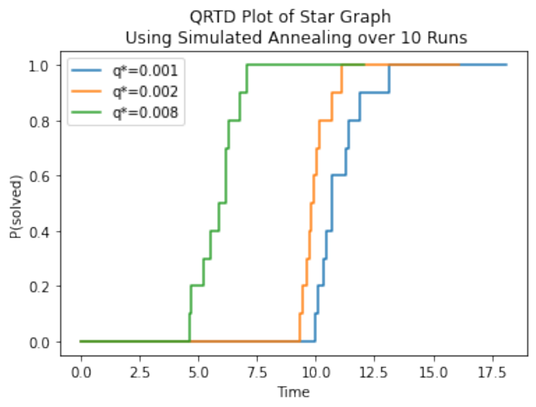

Figure 8: QRTD for Star2, Simulated Annealing

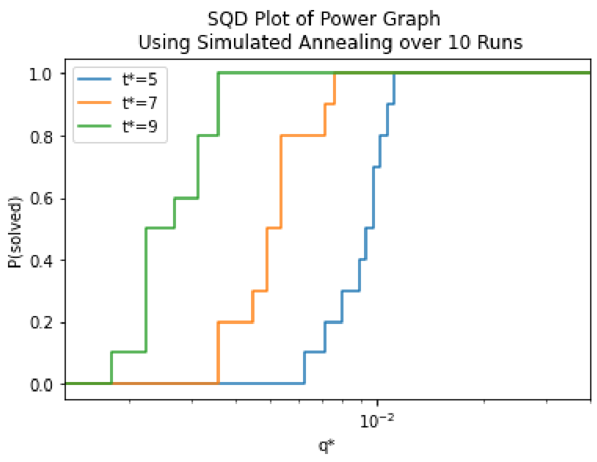

Figure 9: SQD for Power, Simulated Annealing

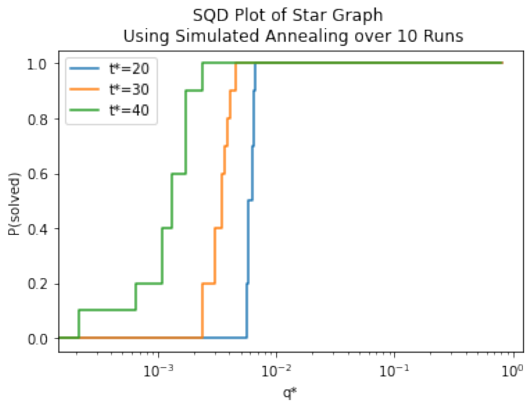

Figure 10: SQD for Star2, Simulated Annealing

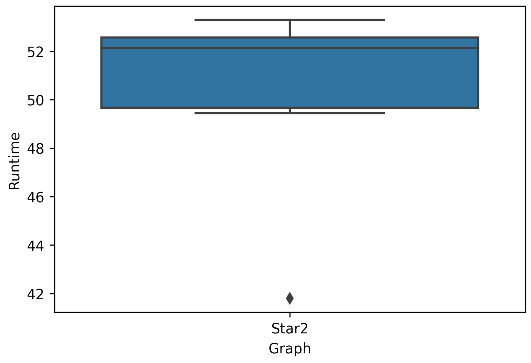

Figure 11: Boxplot of Star2, Simulated Annealing

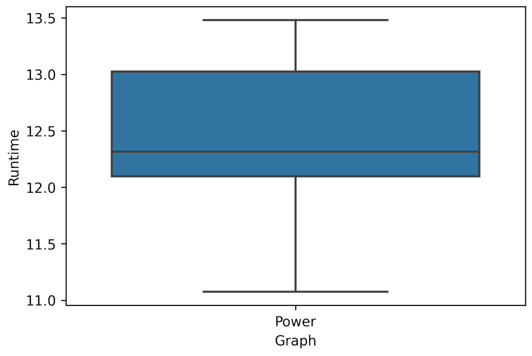

Figure 12: Boxplot of Power, Simulated Annealing

6 DISCUSSION

To make a fair comparison of performance characteristics of all the implemented algorithms, we ran four graphs *jazz, email, as-22july06, star* on the same platform under similar loading. The details of the platform are as under:

(1) CPU: Apple M1 Max
(2) RAM: 64 GB DDR4
(3) System: macOS 13
(4) Language: Python 3.10.8

The random seed and cutoff time were also kept constant for each run. The table below shows the computational time VC value found and the relative error for all five implemented algorithms.

The *jazz* and *email* graphs can be categorized as small-size graphs. All five implemented algorithms were able to find a minimum vertex cover with good accuracy and small computational time. The error for the branch and bound algorithm was the lowest, signifying the importance of this algorithm in finding the exact solution for simple problems. The simple approximation heuristic had the largest error. We can improve the accuracy of the approximate algorithm by using the maximum degree greedy approach. This approach selects the vertex with the highest degree and adds it to the vertex cover. Among the local search algorithms, simulated annealing had the lowest relative error. Maximum Degree Greedy was used to initialize the simulated annealing algorithm, giving the algorithm a good initial solution helped in getting much better results. However, this increased the computational time needed to find the solution.

The *as-22july06* and *star* graphs were chosen to represent medium and larger-sized graphs. The modified branch and bound implementation had the lowest relative error despite the increased graph size but took substantially more time to reach to the solution. This highlights the limitation of the branch and bound algorithm in terms of increased computational time when dealing with large problems. The approximation algorithm with maximum degree greedy heuristic again offered a significant improvement both in terms of computational time and accuracy. For *as-22july06* graph hill climbing algorithm had the maximum computational time. The increased computation time is due to the complexity of the graph. The algorithm uses a nested loop structure to loop through all the edges. For a graph with a large number of edges, this would result

Table 7: Algorithm Comparison

Algorithm	Time (s)	VC Value	RelErr
jazz			
BnB-Modified	0.51	159	0.0063
Approx	0.013	184	0.16
Approx-MDG	0.39	160	0.012
Hill Climbing	0.058	160	0.012
Simulated Annealing	2.59	160	0.012
email			
BnB-Modified	10.20	602	0.013
Approx	0.22	816	0.37
Approx-MDG	0.16	605	0.018
Hill Climbing	0.67	615	0.035
Simulated Annealing	4.09	602	0.013
as-22july06			
BnB-Modified	202.83	3312	0.0027
Approx	19.53	5688	0.72
Approx-MDG	13.81	3312	0.0027
Hill Climbing	354.47	3328	0.0075
Simulated Annealing	42.76	3321	0.0054
star			
BnB-Modified	306.43	7371	0.067
Approx	32.26	10748	0.56
Approx-MDG	17.31	7366	0.067
Hill Climbing	72.61	7239	0.048
Simulated Annealing	76.21	7108	0.029

in increased computation time. We can improve the performance by using a modified implementation of the function to check for vertex cover.

We can also deduce a lower bound on the solution quality by looking at the results of all five algorithms. From the result tables, we can see that the simple approximation algorithm had the least accuracy. We can look for the highest relative error in the approximation approach results to find the lower bound. We can rule out any solution from any approach that has an error higher than this lower bound.

7 CONCLUSION

In this project, we solved the minimum vertex cover problem using five different approaches. The exact branch and bound algorithm guarantees optimality and gave the lowest relative error. However, computational time increased substantially, especially for larger-sized graphs. The approximation algorithm was generally the fastest to run but also the least accurate. We implemented an improved version of the approximation algorithm by using a maximum degree greedy approach to increase the accuracy. Local search algorithms provided a good trade-off between cost and accuracy, this trade-off was more obvious for large-sized graphs where these algorithms were able to find a minimum vertex within an acceptable error limit quickly. We also constructed quality run-time distribution (QRTDs), and solution quality distribution (SQDs) plots for local search algorithms to help understand their performance. The final choice of the algorithm to be used to solve such problems has to be made based on the desired accuracy and availability of the computational budget.

REFERENCES

[1] 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.

[2] Eric Filiol, Edouard Franc, Alessandro Gubbioli, Benoit Moquet, and Guillaume Roblot. 2007. Combinatorial optimisation of worm propagation on an unknown network. *International Journal of Computer Science* 2, 2 (2007), 124–130.

[3] Wael Mustafa. 2021. Shrink: an efficient construction algorithm for minimum vertex cover problem. *Information Sciences Letters* 10, 2 (2021), 9.

[4] Boris Veytsman. 2016. LaTeX Class for Association for Computing Machinery. (2016).

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