CSE 6140: Fall 2022 Project

Analysis of Algorithms to solve Minimum Vertex Cover Problem

Niharika Gali
niharikagali@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Manvitha Kalicheti
mkalicheti3@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Keerthan Ramnath
kramnath6@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Aishwarya Vijaykumar Sheelvant
asheelvant3@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

ACM Reference Format:
Niharika Gali, Manvitha Kalicheti, Keerthan Ramnath, and Aishwarya Vijaykumar Sheelvant. 2022. CSE 6140: Fall 2022 Project Analysis of Algorithms to solve Minimum Vertex Cover Problem. In . ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

The Minimum Vertex Cover (MVC) problem is a classical NP-complete optimization problem. In this paper, we tackle the problem in three directions, with an exact algorithm using Branch and Bound, and an approximate algorithm using the Maximum Greedy Algorithm. For local search, we chose to implement Simulated Annealing and FastVC algorithms. In order to compare different algorithms, we record the time and relative error. For local search, we also plot Qualified Run Time Distribution (QRTD) and Solution Quality Distributions (SQD) to interpret the algorithm's performance over time. In our analysis, we found Simulated Annealing to be a better performer than other algorithms in terms of relative error given a cutoff time. The rest of the report is organized as follows: Firstly, we describe the problem statement more formally in section 2. Subsequently, we perform a literature survey in section 3 and describe the four algorithms in detail in section 4. We then present our results in section 5 and discuss how different algorithms compare in section 6. Finally, we conclude our analysis in section 7.

2 PROBLEM DEFINITION

Given an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, a vertex cover is a subset $C \subseteq V$ such that $\forall (u, v) \in E : u \in C \vee v \in C$. The Minimum Vertex Cover thus boils down to the problem of minimizing $|C|$.

3 RELATED WORK

3.1 Branch and Bound

The Branch and Bound method was first introduced in 1960 by Land and Doig [9]. It is an algorithm design paradigm that is widely used to solve optimization problems. It involves using branching and pruning techniques [3, 13] to explore a state space tree and obtain the optimal solution. Akiba and Iwata, through their effective implementation of the branch and bound method for the Minimum Vertex Cover Problem, showed that these algorithms are actually quite practical and competitive with other state-of-the-art approaches for several kinds of instances. In general, exact algorithms are still quite slow when compared to other heuristic-based approaches for the MVC problem. Our implementation of the Branch and Bound method is discussed in subsection 4.1.

3.2 Approximation Algorithm

Since an exact solution is not always feasible, scientists have experimented with various approximations that lead to a feasible and close-to-optimal solution as a trade-off between run time and efficiency. Åstrand et al. propose a deterministic approximation algorithm that can find a 2-approximation for the MVC in $(maxDegree+1)^2$ cycles or loops, where $maxDegree$ is the maximum degree of the graph. Hanckowiak et al. propose a maximal matching approximation algorithm to find a solution in the order of $O(log^4 n)$ time. Delbot and Laforest proposes six different algorithms, one of which is the Maximum Degree Greedy algorithm, which takes the pros of a greedy search and results in very fast run times, in the order of $O(h(maxDegree))$. As detailed in subsection 4.2, we implemented Delbot and Laforest's Maximum Degree Greedy algorithm.

3.3 Local Search

Developed to provide better solutions in a time-constrained environment, the local search algorithms were one of the go-to algorithms for quickly finding solutions by searching the neighborhood. Over the years, numerous efforts have been made to develop efficient and scalable local search algorithms. The basic ones include the Hill Climbing Algorithm [11], which employs a greedy search technique to improve the solution by choosing the neighbor that yields the largest improvement as the next state. Although the technique is extremely fast (because of its greediness), it fails miserably when there are multiple peaks / better solutions, in which case, it could get stuck at a local optimum. A slightly extended variant of Hill Climbing is Hill Climbing with Restart [11], which attempts to prevent getting stuck at a local optimum by providing some room to explore bad moves. Inspired by the physical phenomenon, Johnson et al. proposed Simulated Annealing for optimization, which had extensions that would include restarts. Other local search algorithms which gained significance for solving minimum vertex cover problems were EWLS[15], COVER[12], and NuMVC[15]. However, the major drawback of these algorithms is that they performed poorly for massive graphs. The most representative algorithm for massive graph inputs is the FastVC algorithm. This algorithm uses

a fast heuristic for constructing a vertex cover and a cost-effective heuristic for choosing the vertex[4]. It should also be noted that local search is used in many Artificial Intelligence research areas [6], thereby making it an interdisciplinary and important class of algorithms.

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Introduction. We implemented a branch and bound algorithm to obtain the exact solution to the MVC problem. This algorithm explores the entire state space tree to get the optimum vertex cover. A global upper bound and a node-wise local lower bound for the optimal solution are maintained. Before expanding a branch from a node, we check to see if it's lower bound is smaller than the global upper bound. If not, we will prune this branch and move to the next one. At each node, we maintain the vertex cover (partial solution) and the subproblem (part of the graph not covered by the vertex cover).

4.1.2 Description. As taught in class, we start with an empty Frontier stack which maintains the list of configurations to be explored. *Choose* step involves picking which node to expand on next. Here, picking the maximum degree node from our subproblem makes our algorithm faster as there is a higher likelihood of this node being a part of our optimum vertex cover set. *Expand* step adds two configurations to the frontier, one without this maximum degree node in our VC, and one with the maximum degree node in our VC. We keep popping our Frontier stack and performing the *Check* step on each configuration which involves three possible results. We reach a feasible and complete solution if our subproblem is empty. We check if our feasible solution is better than the best one found so far and update our global upper bound accordingly. Then, we backtrack and continue searching the rest of the tree. When we reach a partial solution, if our lower bound exceeds our global upper bound, we prune this branch and backtrack to continue with the remaining branches. Otherwise, we again perform *Choose* and *Expand* on the subproblem and add configurations to the Frontier stack. A more detailed description of the algorithm can be found in the pseudocode presented in Algorithm 1.

Although Branch and Bound is an exponential time algorithm whose complexity offsets its ability to find exact solutions, it can be made more efficient by finding tighter bounds for pruning the state space tree. In our implementation, we use the maximal matching approximation algorithm to help us find the lower bound on the MVC of the subproblem. A matching of a graph is the subset of edges of the graph in which no nodes are repeated. A matching is said to be maximal when we cannot add any more edges to it. We use the fact that the approximation ratio is 2 to define our lower bound on the VC for a subproblem as $size(\text{maximal matching}(subproblem))$. The global upper bound is updated to store the best solution we have found so far in the algorithm.

4.1.3 Time and Space Complexity. In the worst case, we scour the entire state space tree of all possible configurations ($2^{|V|}$) to reach the optimal solution. Performing *Choose*, *Expand*, and *Check* for each configuration takes $|V|^2$ times. This gives us a time complexity in the order of $\mathbb{O}(|V|^2 2^{|V|})$. In the worst case, our Frontier might contain both versions of all nodes (in VC, not in VC), giving us a max size of $2|V|$. Our VC's largest possible size would be $|E|$. Thus overall space complexity will be of the order $O(|V| + |E|)$.

4.1.4 Strengths & Weaknesses. The biggest strength of the Branch and Bound algorithm is that if allowed to run to completion, it will always give us the optimal solution. But as the size of the input grows, the costs grow exponentially. Thus, Branch and Bound is not a good fit for large input sizes and is not scalable.

4.2 Approximation Algorithm

4.2.1 Maximum Degree Greedy: Introduction. The algorithm we picked as an Approximation Algorithm for solving the MVC problem is the Maximum Degree Greedy (MDG) approach. The MDG algorithm works by picking the vertex with the highest degree in the graph that is not currently in the vertex cover and adding it to the vertex cover. Once the vertex is added to the vertex cover, it is removed from the graph, and the degrees for all vertices are recalculated. This process is repeated until all the edges have been accounted for.

4.2.2 Description. The algorithm starts by taking the Graph G with a set of vertices V with edges E, along with start and cutoff times, though the algorithm is pretty fast and will never exceed a reasonable cutoff time. We first find the degree of each vertex of Graph G and pick the vertex with the maximum degree. We add this vertex to our vertex cover and remove the vertex from the graph. We recalculate the degrees of all vertices and find the next vertex with the maximum degree. This process is repeated for as long as we have a vertex in the graph with a positive degree. A trick to improve the algorithm's run time is to manually set the degree of the maximum degree vertex to -1 in each cycle instead of removing it from the graph since removing a vertex takes significantly more time. The pseudocode for the Maximum Degree Greedy algorithm can be found at 2.

4.2.3 Approximation Guarantee. The worst-case approximation ratio for the Maximum Degree Greedy Algorithm is guaranteed to be $H(\delta)$ where δ is the maximum degree of any vertex $V \in G$, and $H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ is the harmonic function. Details on the worst-case approximation ratio can be found in Table 1.

4.2.4 Strengths and Weaknesses. In picking the Approximation Algorithm, we researched several other methods, such as the Edge Detection algorithm and a basic Depth First Search algorithm. The advantage of the Maximum Greedy Approach is the ease of implementation, along with a fast run time and decent accuracy. The weakness of the Maximum Degree Greedy approach is its loss of accuracy as the graphs turn bigger. In real-world applications with huge graphs, the MDG algorithm might not meet the efficiency requirements. Details on the relative error to optimal solutions can be found in Table 2.

4.2.5 Time and Space Complexity. The time complexity for the Maximum Degree Algorithm is in the order of $O(|V^2|)$. The first $O(V)$ comes from the while loop that traverses over the vertices of the graph as long as they have a positive degree. In the worst case, it can loop V times. Additionally, we will remove the node from the graph at each step. If we use an efficient data structure

Algorithm 1 Branch and Bound for MVC of G

Input: Graph G, cutoff time
Output: Minimum Vertex Cover of G

```
1:  procedure BranchBound(G, cutoff time)
2:      Initialize candidate to be max degree node in G
3:      Initialize Frontier = [] as an empty stack
4:      Push 2 branches to Frontier: one with candidate in VC, one without candidate in VC
5:      Initialise BestVC = [] to store optimum solution
6:      Initialise BestVCSize = UpperBound to be no. of nodes in G
7:      Initialise VC_curr = [] to store current solution
8:      Initialise subG = G to store the subproblem
9:      while Frontier is not empty and time spent so far < cutoff time do
10:         candidate = Frontier.pop
11:         if candidate in VC then
12:             remove candidate from subG
13:         else
14:             add all neighbours of candidate in G to VC_curr
15:             remove all neighbours of candidate in G from subG
16:         end if
17:         if subG is empty then
18:             if size of VC_curr < BestVCSize then
19:                 BestVC = VC_curr
20:                 BestVCSize = size of VC_curr
21:                 set Backtrack = True
22:             end if
23:         else
24:             if LB(subG) + size(VC_curr) > BestVCSize then      ▷ Lower Bound is the size of a maximal matching of the subproblem.
25:                 Backtrack = True
26:             else
27:                 newCandidate = max degree node in subG
28:                 Push 2 branches to Frontier: one with newCandidate in VC, one without newCandidate in VC
29:             end if
30:         end if
31:         if Backtrack is true and Frontier is not empty then
32:             set par = parent node of last node in Frontier
33:             if par not in VC_curr then
34:                 clear VC_curr
35:                 set subG = G
36:             else
37:                 for each node added to VC_curr after par  do
38:                     add node to subG
39:                     for each nbr of node do
40:                         if nbr in subG and nbr not in VC_curr then
41:                             add edge between nbr and node to subG
42:                         end if
43:                     end for
44:                 end for
45:             end if
46:         end if
47:     end while
48:     return BestVC
49: end procedure
```

such as a HashMap, we can achieve it in O(1) time. Then, we need to recalculate the degrees of each remaining vertex in the graph, which in the worst case, will lead to the time complexity of $O(V)$, thus giving us an overall time complexity of $O(|V^2|)$.

The space complexity of the algorithm is in the order of $O(|V|)$ where V is the number of vertices in Graph G. We use this space to store the degree of each vertex of a graph in a dictionary.

Algorithm 2 Approximation Algorithm for MVC of G

Input: Graph G(V, E), startTime S, cutoffTime C
Output: Find a Minimum Vertex Cover for G(V, E)

```
procedure MaxDegreeGreedy(G)
    mvc ← []
    degree ← dict(G.degrees())
    maxDegree ← max(degree)
    while S < C & degree[maxDegree] > 0 do
        mvc ← mvc ∪ maxDegree
        V ← V − maxDegree        ▷ Remove maxDegree from G
        degree ← dict(G.degrees())
        maxDegree ← max(degree)
    end while
    return mvc, len(mvc)
end procedure
```

4.3 Local Search 1: FastVC

4.3.1 Introduction. We implemented the FastVC algorithm as one of the local search algorithms. FastVC uses a two-stage exchange framework in local search instead of vertex pair exchange. In the FastVC algorithm, we calculate the loss and gain of adding and removing vertices to the vertex cover. We define the gain of a vertex as the number of edges that would be newly covered if that vertex were added to the cover. The loss of a vertex in a vertex cover is the number of edges that would become uncovered if that vertex were removed from the cover. Basically, the algorithm adds the more connected vertex of each uncovered edge in the graph, covers edges when vertices next to them are added, and then removes the vertices whose loss is zero.

4.3.2 Description. Initially, we construct a vertex cover. The loss values for each of the vertices are calculated, and the gain values are assigned to 0. There are two steps that are performed repeatedly till cut off time has elapsed: 1) We pick one vertex $v \in C$ in the vertex cover and remove it from the vertex cover. To remove a vertex, we choose the vertex with minimum loss. The loss and gain values for its neighbors are updated after the vertex is removed. 2) We add a vertex with the highest gain from a randomly uncovered edge to the vertex cover. As vertices are added to and deleted from the candidate, losses and gains are recalculated. After the allocated time for iterations has elapsed, the algorithm returns the best candidate solution.

4.3.3 Strengths and Weaknesses. The major strength of FastVC is that it performs better than any state-of-the-art algorithm for massive graphs. The execution time is faster.
However, there are a few drawbacks to this algorithm. The initial vertex cover is found using a greedy approach which may result in a longer execution time if the solution is far from the optimal solution. Removing one uncovered edge may result in narrowing the search range.

4.3.4 Time and space complexity: The graphs are represented as adjacency matrices and the vertex cover solution is maintained as an array. Updating the loss and gain values depends on the degree of the vertex selected. Therefore we have $O(N_d)$ where N_d denotes the degree of the vertex. Selecting a vertex with minimum loss takes O(|V|). While picking an uncovered edge we have to look at all the edges in the graph and this takes O(|E|) time. After edges have been deleted and added, we verify the candidate solution which takes O(|V|+|E|) time. Therefore the algorithm takes O(|V|+|E|) time. The space complexity of the algorithm is O(|V|+|E|).

4.4 Local Search 2: Simulated Annealing

4.4.1 Introduction: As the next approach in local search, we implement the Simulated Annealing algorithm [8] as taught in class. The basic idea of this algorithm is to look for a better solution in the neighborhood and accept it if we find one. If not, we still accept it with some probability. This probability is related to how much worse the neighborhood solution is in comparison to the current solution, along with a constant that is called temperature, which decreases in each iteration. Hence, the probability of accepting a worse solution decreases over time, and in the limit of $t \to \infty$, or when the temperature goes to 0, we return the greedy optimal solution with the current vertex cover.

4.4.2 Description: In our algorithm, we first initialize the vertex cover as the set of all nodes. Subsequently, till we reach the cutoff time, we search the neighborhood and accept the solution if it's better. The neighborhood search is carried out by removing and adding a random node to the vertex cover. We also keep track of the set of unreachable nodes to measure the quality of the solution. We chose this metric because the number of unreachable nodes is zero when we find a vertex cover. This metric is inspired by the work of Xu and Ma. Thus, we measure the improvement using the difference between the number of unreachable nodes before and after the neighborhood search. If the number of unreachable nodes increases after the modification, it means we have a worse solution than before. Thus, we accept it with a probability exp $(\Delta E/T)$ where ΔE is the solution improvement, and T is the temperature. The pseudocode for the algorithm is presented in Algorithm 4. As given in the lecture slides, we have a geometric schedule with $\alpha = 0.95$. In order to fix the initial temperature, we referred to [2] to set it to 0.8. For averaging the effects of randomness, we ran the simulation for each dataset 10 times and took the average.

4.4.3 Strengths and Weaknesses: Simulated Annealing achieves good performance in general as its framework allows both exploration and exploitation (in reinforcement learning terms). On top of that, as discussed in the lecture, it has good convergence properties. However, the room for exploration may have a significant impact on the run time of the algorithm.
Time and space complexity: The worst case scenario is when we have a high temperature, in which case, the probability of choosing a sub-optimal neighbor is high. In this case, since we traverse through all the neighbors of the nodes, the time complexity is

Algorithm 3 FastVC Algorithm

Input: Graph G start_time, cutoff time
Output: Optimal solution for Vertex Cover of Graph

```
procedure FastVC(Graph, start_time, cutoff time)
    Construct a vertex cover solution VC from the graph
    E ← set(Edges)
    for every edge(u,v) in E do
        if VC(u) + VC(v) == 0 then
            if Graph[u] > Graph[v] then
                VC[u] ← 1
            else
                VC[v] ← 1
            end if
        end if
    end for
    Update the loss and gain after adding edges
    for every edge(u,v) in E do
        if VC(u) + VC(v) == 1 then
            if VC(u) > VC(v) then
                loss[v] ← loss[v] + 1
            else
                loss[u] ← loss[u] + 1
            end if
        end if
    end for
    Remove edges with loss=0
    VC, runtime ←Calculation(Graph,VC,gain,loss,cut_offtime,start_time)
    t ←vertices in VC
    return VC,t,runtime
end procedure
procedure Calculation(Graph, VC, gain,loss,cut_offtime,start_time)
    while current time - start_time < cut_offtime do
        if VC is a valid solution then
            Find the vertex with minimum loss(v)
            Remove the vertex from the VC.
            for i in Graph[v] do
                if VC(i) == 0 then
                    gain[v] ← gain[v] + 1
                else
                    loss[v] ← loss[v] + 1
                end if
            end for
            Calculate the gain and loss after removal of vertex
        end if
        for v ∈ Graph with minimum loss do
            Find the edges associated with v
            If edge not in VC
            Add edge to VC
        end for
        Remove v from VC
        Pick a random edge and add the most connect vertex to VC (u,v)
        Find the most connected vertex by checking gain[u] and gain[v]
        Update loss and gain after adding vertex
    end while
    return VC,current_time-start_time
end procedure
```

$O(|V|)$ where V is the number of nodes in a graph. The space complexity is $O(|V| + |E|)$ where E is the number of edges in the graph since we store the graph object with the nodes and edges.

5 EMPIRICAL EVALUATION

5.1 Platform Description

- **Processor:** Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
- **Memory:** 8.00 GB RAM
- **System type:** 64-bit operating system, x64-based processor
- **OS Edition:** Windows 10 Home Single Language
- **Version:** 21H2
- **Python:** 3.9.6

5.2 Results: Branch and Bound

We ran the Branch and Bound algorithm for 600s on each of the graphs. Since Branch and Bound is an exact algorithm, we get optimal or near-optimal solutions with very low relative errors for smaller graphs. If allowed to run for longer, we would see similar results for larger graphs as well. In this experiment, this algorithm gave us a maximum relative error of 0.0484 for the *delaunay* graph. We got exact solutions for the *jazz*, *karate* and *netscience* graphs, and nearly exact solutions (relative error of the order 10^{-3}) for the *as-22july06* and the *hep-th* graphs. The results are presented in Table 2.

5.3 Results: Approximation Algorithm

The Maximum Degree Greedy Algorithm is the fastest algorithm implemented out of the four approaches. For the *karate* and *netscience* graphs, the Maximum Degree Greedy algorithm achieved the optimal solution in less than a second, indicating that for smaller graphs, an Approximation Algorithm can provide an exact solution. It performed the worst on the *star* graph, with a relative error of 0.067 in 4.26s. If this trade-off is acceptable for the application at hand, the Maximum Degree Greedy algorithm proves to be a great choice for finding the vertex cover of a graph.

Table 1 shows the maximum degree for each graph, along with the optimal vertex covering number and the one achieved by our approximation algorithm. The last column indicates the worst-case approximation algorithm, which happens to be $H(\delta)$ with H being the harmonic function. As is evident, the approximation ratio is well within the bounds of $\frac{approx\_ans}{opt\_ans}$ for all graphs. For example, our highest relative error is measured for the *star* graph with a max ratio of 1.067, wherein the worst-case approx ratio for the *star* graph as seen in Table 1, is 8.23.

5.4 Results: FastVC

Due to hardware constraints, we ran the simulated annealing algorithm on *power*, *star*, and *star2* graphs with a cutoff time of 600 seconds whereas, for the other graphs, we ran the algorithm with a cutoff time of 120 seconds. As recommended in the project description, we run the algorithm for 10 different random seeds (seeds being from 0 to 9) and take the average of the maximum time taken and the minimum vertex cover while reporting results. From Table 2, we can infer that the FastVC algorithm has slightly longer run times. Since an initial vertex cover is constructed based on the number of edges, we get an upper bound on the size of the minimum vertex cover quickly. We notice that the time taken for reaching the optimum solution does not depend on the size of the graphs. We also notice that changing the seed does not have an effect on the solution of the algorithm. All iterations of the algorithm with different seed values produce the same optimum value. The highest relative error was observed for star2 graph. For karate and netscience graphs we got the exact optimum solution.

We then plot QRTD and SQD analysis on *power* and *star2*. We select q^* as 0.1%,1%,5%,10%,12% for both graphs. Figure 1 and Figure 3 represent these plots respectively. From the QRTD plot for power graph, $q^* = 10\%, 12\%$, we notice that all the runs will result in a vertex cover solution with a size of atmost optimal size plus q^* percent. On the other hand, for $q^* = 0.1\%, 1\%, 5\%$, we see that the fraction of runs having a marginal optimal solution stays at zero. This is because the algorithm did not improve before reaching the relative error mentioned in Table 2. The lines are constant because there was not much improvement after the initial solution was found. We observe the same from QRTD of the star graph as well. For $q^* = 12\%$ we can see that we achieve the optimal solution plus q^* percent for all the runs of the algorithm. For SQD analysis of the *power* graph, we selected $t = 10, 15, 20, 25, 30, 35$ seconds. The results are displayed in Figure 2. For SQD analysis of the *star* graph, we selected $t = 0.01, 0.1, 1, 10, 15, 20$ seconds. The results are presented in Figure 4. From both these graphs, we can see that the increase in time does not lead to better optimization which confirms our observations from the QRTD plots.

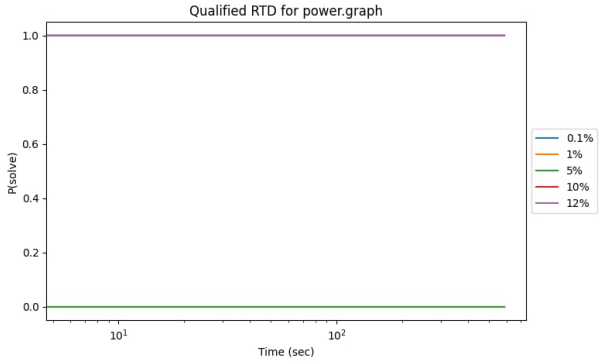

Figure 1: QRTD plot using FastVC on *power.graph*

5.5 Results: Simulated Annealing

As carried out in subsection 5.4, due to hardware constraints, we ran the simulated annealing algorithm on the *power*, *star*, and *star2* graphs with a cutoff time of 600 seconds.We ran the algorithm with a cutoff time of 120 seconds for all other graphs. As recommended in the project description, we run the algorithm for 10 different random seeds (seeds being from 0 to 9) and take the average of the maximum time taken and the minimum vertex cover while reporting results. From Table 2, we infer that the simulated annealing algorithm performs better than other algorithms. However, this performance is at the cost of higher run times. While running the experiments, we noticed that the minimum vertex cover changes

Algorithm 4 Simulated Annealing for Minimum Vertex Cover

Input: Graph G, Temperature T, cutoff time, start time, Temperature Scaling factor α
Output: Optimal set of vertex cover VC with |VC| minimized

procedure SimulatedAnnealing(G, T, cutoff time, start time, α)
  Initialize VC_{curr} to be set of all nodes in G
  Initialize an empty list denoted as unreachableNodes
  while current time - start time < cutoff time **do**
    if $T == 0$ **then**
      return greedy(G, VC_{curr}, cutoff time, start time)
    end if
    while |unreachableNodes| == 0 **do** ▷ When the current solution is a vertex cover
      Copy VC_{curr} to VC_{opt}, the optimal vertex cover
      Append all neighbours of the nodes present in VC_{curr} that is not present in VC_{curr} to unreachableNodes. Remove the corresponding node from VC_{curr}
    end while
    Save a copy of VC_{curr} and unreachableNodes
    Randomly delete a node from VC_{curr} and update unreachableNodes
    Randomly add a node from unreachableNodes to VC_{curr} and update unreachableNodes
    Let ΔE = |unreachableNodes| − |$modified$ unreachableNodes|
    if $\Delta E < 0$ **then**
      Let $p = \exp(\Delta E/T)$
      With prbability p, accept the modified VC_{curr} and unreachableNodes
    end if
    Decrement temperature by $T = \alpha T$
  end while
  return VC_{opt}
end procedure
procedure greedy(G, VC_{curr}, cutoff time, start time)
  Let $N = |\text{VC}_{curr}|$
  Sort VC_{curr} in the increasing order of degree
  Initialize nodeIdx to 0
  while current time - start time < cutoff time **and** nodeIdx < N **do**
    Let $u = \text{VC}_{curr}$[nodeIdx]
    If u has all its neighbours in VC_{curr}, remove u from VC_{curr}
    Increment nodeIdx by 1
  end while
  return VC_{curr}
end procedure

graph	$max(\delta)$	opt_ans	$approx\_ans$	$approx\_ratio$
as-22july06.graph	2390	3303	3312	8.35
delaunay n10.graph	12	703	740	3.10
email.graph	71	594	605	4.84
football.graph	12	94	96	3.10
hep-th.graph	50	3926	3947	4.49
jazz.graph	100	158	160	5.18
karate.graph	17	14	14	3.43
netscience.graph	34	899	899	4.11
power.graph	19	2203	2272	3.54
star.graph	2109	6902	7366	8.23
star2.graph	1531	4542	4677	7.91

Table 1: Approximation Ratio Stats for Maximum Degree Greedy

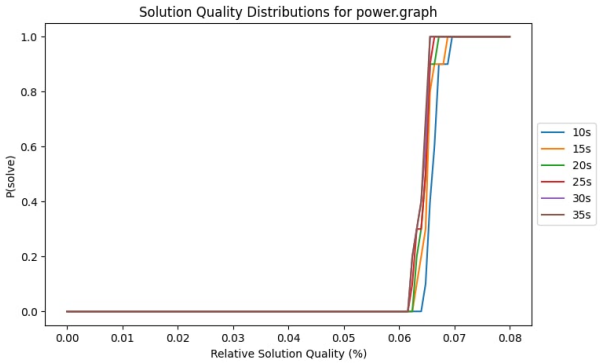

Figure 2: SQD plot using FastVC on *power.graph*

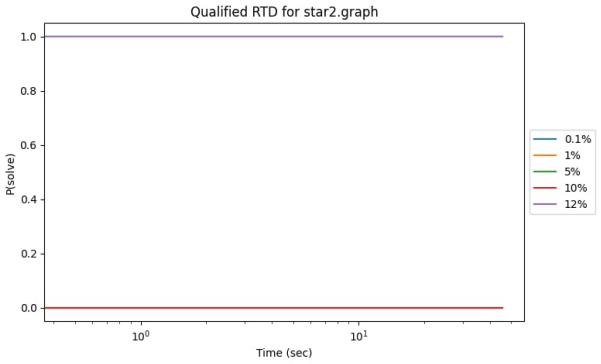

Figure 3: QRTD plot using FastVC on *star2.graph*

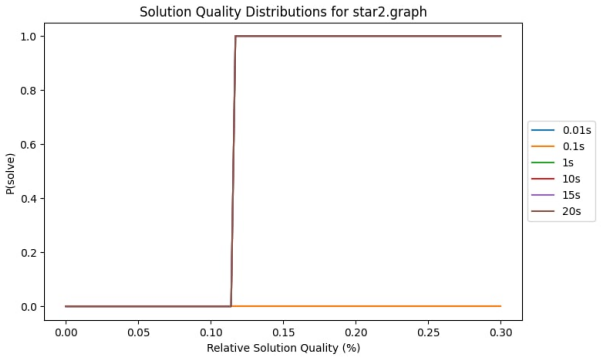

Figure 4: SQD plot using FastVC on *star2.graph*

with a change in random seed, meaning that the performance is not robust to the randomness involved in the process. It should also be noted that a higher number of vertices need not necessarily mean longer run times. Hence, it is possible that the nature of the graph (which includes how connected/complex it is) plays an important role in determining the run time.

We then perform the QRTD and SQD analysis on the *power* and *star2* graphs. We select 0.05%, 0.1%, 0.5%, 0.8% and 1% for QRTD analysis on the *power* graph and selected 0.1%, 0.8%, 1%, 1.5%, 2%

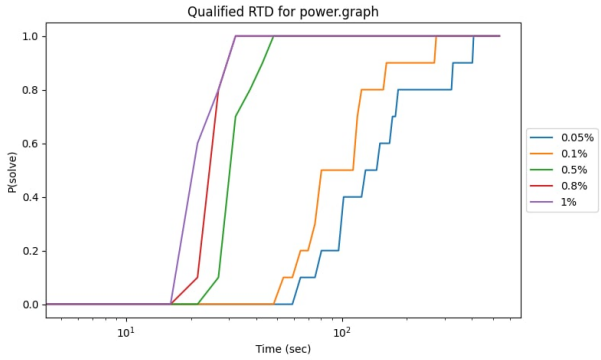

Figure 5: QRTD plot using simulated annealing on *power.graph*

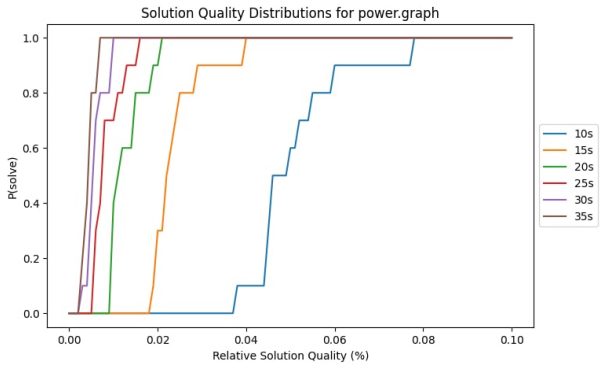

Figure 6: SQD plot using simulated annealing on *power.graph*

for QRTD analysis on the *star2* graph. The QRTD plots for both graphs are presented in Figure 5 and Figure 7 respectively. In the *power* graph plot, we notice that for $q^* = 0.05\%, 0.1\%$, we obtain the optimal solution plus q^* percent of the optimal in every run. We notice it takes longer time to completion when $q* = 0.05\%, 0.1\%$. However, with a slightly larger q^*, we do not have to run the algorithm for 600 seconds but can rather terminate it well before and still get an optimal solution with an insignificant performance drop. Similar QRTD inference can be made on *star2* wherein, for almost all $q^* > 0.1\%$, we achieve VC of size at most optimal size plus $q^*\%$ of that with very similar run times. Hence, in this setting, we believe that one could go for a smaller q^* that is greater than 0.1% and still not see a significant difference in run times. With respect to SQD analysis, we selected $t = 10, 15, 20, 25, 30, 35$ seconds for *power*, whose results are presented in Figure 6. It shows us that with time greater than 20 seconds, we are able to achieve optimal performance in each run with a very small performance drop, which corroborates our QRTD analysis. Similarly, for SQD analysis on *star2*, we selected $t = 80, 90, 100, 150, 180, 240$ seconds. The results are presented in Figure 8. We notice that with a q^* that is lesser and greater than 0.1%, we are able to achieve performance gain with an insignificant increase in run time (only around 10 seconds increase on the total run time of 600 seconds). This again confirms our inference from the QRTD plot of *star2*.

Dataset	Branch and Bound			Approximation			FastVC			Simulated Annealing		
	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr
as-22july06	420.54	3309	0.0018	3.81	3312	0.0027	12.980	3347	0.0217	119.7798	3452	0.0451
delaunay	43.53	737	0.0484	0.041	740	0.0530	39.631	745	0.0597	62.5919	704	0.0014
email	18.89	604	0.0168	0.036	605	0.0190	9.128	640	0.0774	19.1821	594	0
football	0.25	95	0.0106	0.0008	96	0.0210	0.3081	96	0.0212	0.3179	94	0
hep-th	54.07	3947	0.0053	1.49	3947	0.0053	0.0330	3979	0.0134	107.924	3928	0.0005
jazz	16.15	158	0	0.0023	160	0.0130	6.497	162	0.0253	0.4905	158	0
karate	0.00	14	0	0.00009	14	0	0.00008	14	0	0.0052	14	0
netscience	0.91	899	0	0.0670	899	0	0.00367	899	0	2.3676	899	0
power	12.19	2271	0.0309	0.57	2272	0.0310	397.23	2329	0.0571	229.5765	2203	0
star	147.99	7366	0.0670	4.02	7366	0.0670	0.5181	7492	0.0850	591.6219	7263	0.0523
star2	449.02	4662	0.0264	3.29	4677	0.0300	5.500	5070	0.1162	594.4575	4566	0.0052

Table 2: Comprehensive comparison of results after running all 4 algorithms on the given datasets

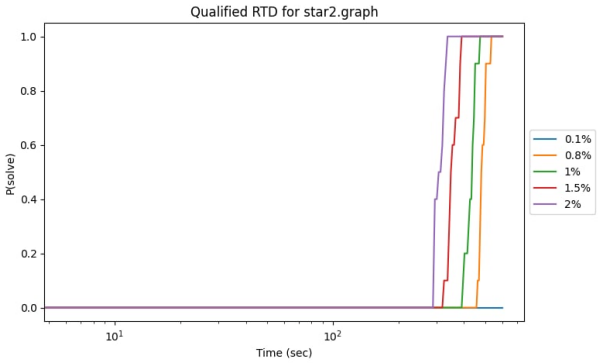

Figure 7: QRTD plot using simulated annealing on *star2.graph*

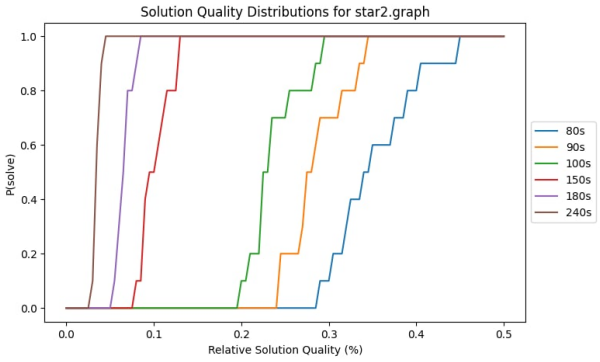

Figure 8: SQD plot using simulated annealing on *star2.graph*

6 DISCUSSION

Overall, Simulated Annealing had the best performance with exact solutions for most graphs. Branch and Bound was expected to take a long time to give optimal solutions, but with efficient implementation using tight bounds to prune the tree, we got near-optimal solutions more quickly.

Maximum Degree Greedy gives extremely fast results but at the cost of a slightly higher relative error. The Maximum Degree Greedy approach is a good choice for graphs where the run time is more important than accuracy.

FastVC showed us the highest relative error, being slightly worse than Approx while taking more time than it. Since the other Local Search Algorithm (Simulated Annealing) proved to be our best, the choice of the local search approach chosen can greatly vary the accuracy of finding a minimum vertex cover.

In order to compare run times of the local search algorithms, we also make box plots on the *power* and *star2* graphs, which are presented in Figure 9 and Figure 10 respectively. We observe that the FastVC algorithm has more variation and takes more time in the *power* graph when compared to the Simulated Annealing algorithm. We also noticed that about three runs of SA algorithm took significantly longer than the average time. Similarly, there is one run by FastVC which achieved significantly lower run time than the mean. The circles in the box plot represent these outliers. On the other hand, Simulated Annealing takes significantly more time than FastVC for *star2*.

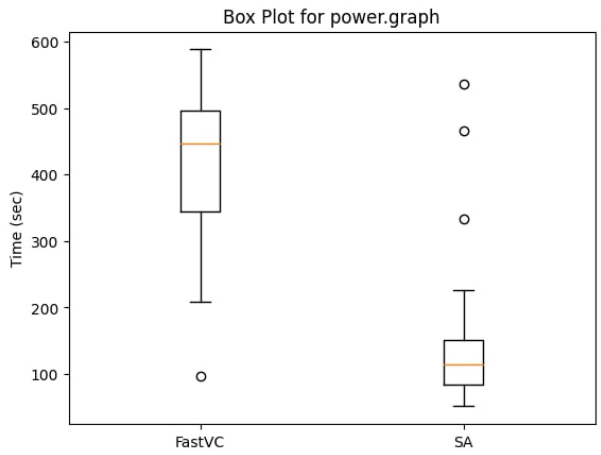

Figure 9: Box plot for runtime comparison of local search algorithms on power.graph

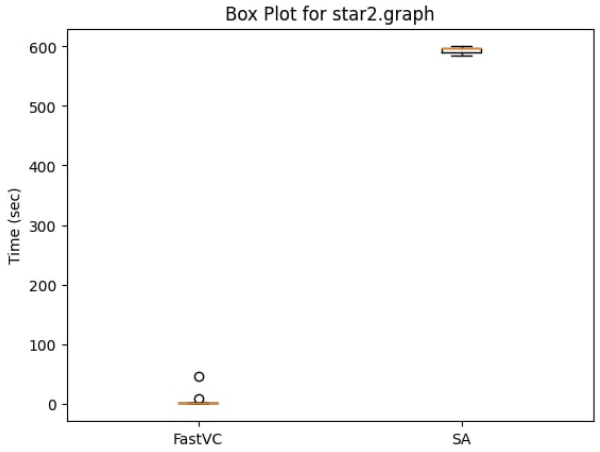

Figure 10: Box plot for runtime comparison of local search algorithms on star2.graph

The best result was found by the Branch and Bound algorithm for *karate*, solving the problem optimally in a negligible amount of time. The least accurate solution was found by the FastVC algorithm on *star2*, with a relative error of 0.1162.

7 CONCLUSION

Ultimately, the choice of algorithm will depend greatly on the application and the nature of the input. If time is of the essence, and we are prepared to compromise slightly on the quality of the solution, Approximation Algorithms are ideal. If solution quality is of prime importance, and time is not an issue, Branch and Bound is the right algorithm, as it always gives us the optimal solution. If our application demands a balance between the time taken and the quality of the solution, Local Search algorithms like Simulated Annealing take the cake.

REFERENCES

[1] Takuya Akiba and Yoichi Iwata. 2016. Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover. *Theoretical Computer Science* 609 (2016), 211–225. https://doi.org/10.1016/j.tcs.2015.09.023

[2] Walid Ben-Ameur. 2004. Computing the Initial Temperature of Simulated Annealing. *Comput. Optim. Appl.* 29, 3 (dec 2004), 369–385. https://doi.org/10.1023/B:COAP.0000044187.23143.bd

[3] Nicolas Bourgeois, Bruno Escoffier, Vangelis Th. Paschos, and J. M. M. van Rooij. 2008. Fast algorithms for MAX INDEPENDENT SET in graphs of small average degree. (Sept. 2008). https://hal.archives-ouvertes.fr/hal-00321618 working paper or preprint.

[4] Shaowei Cai. 2015. Balance between Complexity and Quality: Local Search for Minimum Vertex Cover in Massive Graphs. (2015).

[5] Francois Delbot and Christian Laforest. 2010. Analytical and experimental comparison of six algorithms for the vertex cover problem. *ACM Journal of Experimental Algorithms - JEA* 15 (03 2010). https://doi.org/10.1145/1865970.1865971

[6] Fred Glover and Harvey J. Greenberg. 1989. New approaches for heuristic search: A bilateral linkage with artificial intelligence. *European Journal of Operational Research* 39, 2 (1989), 119–130. https://doi.org/10.1016/0377-2217(89)90185-9

[7] Michał Hanckowiak, Michał Karonski, and Alessandro Panconesi. 1997. On the Distributed Complexity of Computing Maximal Matchings. 4 (Jun. 1997). https://doi.org/10.7146/brics.v4i38.18964

[8] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. 1989. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research* 37, 6 (1989), 865–892. http://www.jstor.org/stable/171470

[9] A. H. Land and A. G. Doig. 1960. An Automatic Method of Solving Discrete Programming Problems. *Econometrica* 28, 3 (1960), 497–520. http://www.jstor.org/stable/1910129

[10] Matti Åstrand, Patrik Floréen, Valentin Polishchuk, Joel Rybicki, Jukka Suomela, and Jara Uitto. 2009. A Local 2-Approximation Algorithm for the Vertex Cover Problem. In *Proceedings of the 23rd International Conference on Distributed Computing* (Elche, Spain) *(DISC'09)*. Springer-Verlag, Berlin, Heidelberg, 191–205.

[11] Stuart J. Russell and Peter Norvig. 2002. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0137903952

[12] Charles Gretton Silvia Richter. 2007. A Stochastic Local Search Approach to Vertex Cover. *Springer* (2007). https://doi.org/10.1007/978-3-540-74565-5_31

[13] Mingyu Xiao and Hiroshi Nagamochi. 2017. Exact algorithms for maximum independent set. *Information and Computation* 255 (2017), 126–146. https://doi.org/10.1016/j.ic.2017.06.001

[14] Xinshun Xu and Jun Ma. 2006. An efficient simulated annealing algorithm for the minimum vertex cover problem. *Neurocomputing* 69, 7 (2006), 913–916. https://doi.org/10.1016/j.neucom.2005.12.016 New Issues in Neurocomputing: 13th European Symposium on Artificial Neural Networks.

[15] Mengjie Zhang Ke Tang Xiaodong Li Qingfu Zhang Ying Tan Martin Middendorf Yaochu Jin Yuhui Shi, Kay Chen Tan. 2017. Simulated Evolution and Learning. *Springer* (2017). https://doi.org/10.1007/978-3-319-68759-9