CSE 6140 Project - Minimum Vertex Cover

SAI PRASATH S[*], Georgia Institute of Technology, USA
PRANOY RAY, Georgia Institute of Technology, USA
RUPESH KUMAR MAHENDRAN, Georgia Institute of Technology, USA
ANAHITAA RADHAKRISHNAN, Georgia Institute of Technology, USA

ACM Reference Format:
Sai Prasath S, Pranoy Ray, Rupesh Kumar Mahendran, and Anahitaa Radhakrishnan. 2022. CSE 6140 Project - Minimum Vertex Cover. 1, 1 (January 2022), 16 pages. https://doi.org/0000000.0000000

1 INTRODUCTION

The Minimum Vertex cover (MVC) problem is a well known NP-complete problem with numerous applications in computational biology, operations research,the routing and management of resources. In this project, we study four different algorithms for computing the vertex cover for a given graph: Branch and Bound (BnB), Approximation (Approx), Local Search 1 - Simulated Annealing (LS1), and Local Search - 2 Hill Climbing (LS2). For each of the algorithms we discuss their advantages and disadvantages, and describe the approximation ratio (if exist), time and space complexities, and the pseudo codes used in this project. We then empirically compare the performances of these algorithms by evaluating their run time and quality of the vertex cover on different sets of graph. From our analysis we find that Branch and Bound takes the most time to converge where as the approximation algorithms take the least time. Interestingly, the quality of solutions achieved by the approximation algorithms is better than Branch and Bound, and comparable with the Local Search algorithms but achieves it in lesser time. On the other hand, Local Search algorithms achieve the best solution quality but at the cost of more time. Both the Local Search algorithms achieve similar quality solutions but the simulated annealing techniques takes lesser time in most situations. We explore these results in detail in this project.

2 PROBLEM DEFINITION

Given an undirected graph $G(V,E)$ with the set of vertices as V and set of edges as E, a vertex cover S of the graph G is a subset of vertices in V, $S \subseteq V$ such that $\forall e(u,v) \in E$, $u \in S \vee v \in S$. Minimum Vertex Cover of the graph G, is the vertex cover S of smallest possible size, i.e. $|S|$ is minimum.

[*]All authors contributed equally to this project.

Authors' addresses: Sai Prasath S, Georgia Institute of Technology, Atlanta, Georgia, USA, saiprasath3344@gatech.edu; Pranoy Ray, Georgia Institute of Technology, Atlanta, Georgia, USA, pranoy@gatech.edu; Rupesh Kumar Mahendran, Georgia Institute of Technology, Atlanta, Georgia, USA, rmahendran3@gatech.edu; Anahitaa Radhakrishnan, Georgia Institute of Technology, Atlanta, Georgia, USA, aradhakr35@gatech.edu.

3 RELATED WORKS

The MVC problem is a famous NP-Complete problem in Computer Science. Various previous research works have proposed different techniques for solving the MVC problem. In this section, we highlight few of the relevant research works in Branch and Bound, Approximation and Local Search algorithms.

R Marti et al [6] developed a solution for maximum diversity problem using branch and bound algorithm and used different bounds for the partial solutions (sets of solutions with common vertices) such as sum of the distances between pairs of vertices, sum of edge weights where one vertex is in the considered partial solution and sum of edge weights with all vertices in the partial solution. On average, only 0.09% of the nodes in the search tree were explored due to the bounds. The empirical evaluation indicates that the algorithm performs well with medium-sized problems and performs as good as linear integer formulation approach, although none of the methods work as efficiently for much larger problems.

However, using Branch and Bound algorithms to find exact solutions can be time consuming for larger graphs, and therefore are not practically applicable. Therefore, research works have focused on developing algorithms that can reach a near-optimal solution with approximation guarantees relatively quick (trading optimality for time). Previous works by Delbot et al. 2010 analyzed the approximation ratio, run time and relative error of six different approximation algorithms: MAXIMUM DEGREE GREEDY, GREEDY INDEPENDENT COVER, DEPTH FIRST SEARCH, EDGE DELETION, LISTLEFT, and LISTRIGHT on various different graphs [5]. We utilize the GREEDY INDEPENDENT COVER (GIC) algorithm for this project.

Further, Local search is yet another strategy that is viable and is based on the oldest optimization technique: trial and error. Local search algorithms travel from solution to solution in its neighbourhood of candidate solutions, then we swap a portion of the solution until the quality is close to the ideal or a cut-off period has been achieved. According to several Lin-Kernighan (LK) versions, local search algorithms have successfully reduced error rates in big graphs to between 1-4 percent. To shorten the execution time of the present technique, improve the current implementation, and surpass cutting-edge heuristic algorithms, Cai introduced NuMVC two-stage exchange and edge weighting with forgetting. [8]. Local search algorithms for MVC have shown efficiency, with a heuristic idea to assign some scores to nodes or edges. However, single edge-weighting algorithms are not robust, given its greedy search strategy. As for graph instances that defeat greedy heuristics, the performance is not good. Thus, the authors in [8] introduce a two-weighting mechanism to counter the greedy effect of single weight. Even more, as shown in [7], the ability to combine different techniques on algorithms such as k-opt and stochastic methods, without affecting the run time of the model serves extensive study and implementation of such models. Some examples of the previous may be, simulated annealing, tabu search, Hill climb, and random walking. However, this study will be focused on the implementation of Simulated Annealing and Hill-Climb. Xu and Ma proposed a Simulated Annealing algorithm for finding vertex cover where they introduced a new accepting factor [4]. Su and Cai proposed a more optimized version of the Hill Climbing algorithm for finding the vertex cover by proposing two new strategies: propose two new strategies: two-stage exchange and edge weighting with forgetting. [9]

4 ALGORITHMS

In this section, we describe the various algorithms used in this project along with their pseudo codes, approximation ratio (if exits), time and space complexities and their advantages and disadvantages.

4.1 Branch and Bound

4.1.1 Description. Branch and bound is an algorithm that solves problems using a combinatorial approach, where it considers different combinations of elements in a set to come up with a logical and possibly optimal solution. It revolves around the concept that out of the total set of possible solutions, smaller subsets could be considered and examined to determine the solution. In this case where we consider a graph with vertices and edges, the combinations of vertices are considered based on two possibilities pertinent to each vertex: one where a vertex is considered to be part of the solution, another where the vertex is not part of the solution. This implies that the total possible combinations of vertices would be 2^n, where n = number of vertices.

This approach would require to run at exponential time, which may not be feasible all the time, especially when dealing with graphs that contain large number of vertices. Thus, this algorithm is implemented with a time limit, such that it returns the current best solution once the time limit has exceeded. The advantage of using branch and bound lies in the fact that it finds the optimal solution by keeping track of the best solution found up to that certain point. The disadvantage would be the time required to find the optimal solution for larger graphs, where it would either take up the entire cut-off time or traverse the entire tree, whichever one is faster. Due to the restriction in time, the solution returned after termination of the algorithm may not necessarily be the most optimal, but is the best solution that the approach could yield for the given computational period.

The specifics of the algorithm implemented are as follows: the upper-bound of branch and bound is the length of the vertex cover with the smallest size. The lower-bound is generally defined as the number of edges in the problem, divided by the node with the highest degree. The lower-bound of a sub-set of the problem is defined as size of vertex cover added to the lower-bound calculated for the un-visited graph.

This approach picks the vertex of the graph with the highest degree each time as ‘candidates’. These candidates are used to arrive closer to the optimal solution and each of these candidates are assigned the states ‘True’ or ‘False’, implying that they are either considered, or not considered to be part of the optimal solution (respectively). The id of the candidates along with their chosen states are appended to the frontier. The algorithm progresses by considering the state of each candidate and following the given logic:
If a vertex candidate is considered, its neighbours need not be considered as part of the optimal solution. If a vertex candidate is not considered, its immediate neighbours are surely considered as part of the optimal solution.

The pseudo-code for this algorithm is described in Algorithm 1.

4.1.2 Time and Space complexity. The branch and bound algorithm explores all possible combinations of vertices of the graphs in the two states: considered as part of optimal VC, not considered part of optimal VC. Thus, the time complexity of branch and bound is $O(2^n)$, where $n = |V|$. The space complexity is $O(|V| + |E|)$ since the graph is to be stored. $2|V|$ is the space required to store the frontier. $|E|$ is the space to find the solution for vertex cover.

Algorithm 1: Branch and Bound Algorithm

```
Data: Graph $G = (V, E)$ , TimeLimit
Result: Vertex Cover of $G$
$VC \longleftarrow \emptyset$
$upperbound \longleftarrow \mid V \mid$
$v \longleftarrow$ selected vertex with highest degree
$frontier \longleftarrow [(v, False, -1, None), (v, True, -1, None)]$
#two different branches based on vertex 'v's consideration as part of optimalVC are created.
#Their parent values are set to -1, None.
while $frontier \neq \emptyset$ do
    $curr\_v, curr\_v\_state, parent\_id, parent\_state = frontier.pop()$
    $VC.append(curr\_v, curr\_v\_state)$
    if $curr\_v\_state == True$ then
        G=G-curr_v
    end
    $curr\_v\_state == False$  $VC.append($all neighbours of curr_v with state=True$)$
    G = G- all neighbours of curr_v
    if $edges\ in\ G == 0$ then
        if $len(VC) < upperbound$ then
            upperbound=len(VC)
            opt_ans=VC
            #BACKTRACKING CODE:
            if $len(frontier)!=0$ then
                parent, parent_state=frontier[-1][-1]
                if $parent\ in\ VC$ then
                    i=index of parent in VC
                    VC$_{removed}$ $= VC[i:]$
                    VC=VC-VC$_{removed}$
                    G=G+ VC$_{removed}$
                end
                else
                    VC=[]
                    G=original G
                end
            end
        end
    end
    else
        if $lowerbound(G) + len(VC) < upperbound$ then
            vj=vertex with highest degree from set of unexplored vertices
            frontier.append([vj, False, v, v_state])
            frontier.append([vj, True, v, v_state])
        end
        else
            #USE BACKTRACKING. (ref: BACKTRACKING CODE abv)
        end
    end
end
return $opt\_ans$
```

4.2 Approximation Algorithms

4.2.1 Description: In general, approximation algorithms are a class of algorithms that trade quality of a solution for reaching the solution much quicker. Therefore, the solutions reached by the approximation algorithms are not optimal but have some approximation guarantee in terms of how comparable they are to the optimal solution. However, by compromising on the quality of the solution, they are able to reach the solution much quicker. Therefore, they are valuable for practical applications where finding a possible solution in short time is more important than finding an optimal solution.

For this project, we consider the GREEDY INDEPENDENT COVER (GIC) algorithm analyzed in Delbot et al. 2010 [5]. We also considered other approaches in Delbot et al. 2010 such MAXIMUM DEGREE GREEDY (MDG) and EDGE DELETION (ED), but selected the GIC approach because the VC reached by this algorithm was near optimal. Although, the algorithm takes more time (in few cases) compared to MDG and ED to reach the solution, we believe that the quality of the solution generated by this algorithm out weights the trade off in terms of the time taken to reach the solution. Moreover, we use the ED algorithm to create the initial solution for the Local Search algorithms which will be discussed in detail in the next few sections.

The GIC algorithm uses the relation between Vertex Cover and Independent Set to find the VC. In any graph $G(V, E)$, if $S \subseteq V$ is an independent set (no two vertices in S are adjacent to each other), then $V \backslash S$ is a Vertex Cover because there are no edges between vertices in S, all edges are between vertices in $V \backslash S$. For finding the maximum independent set, we start by choosing the min degree vertex and adding it to the current independent set, and then we delete all its neighbours along with the vertex itself. We continuously perform this operation until there are no vertices left to be chosen. Instead, for finding the MVC, we add all the neighbours instead of the vertex (analogous to $V \backslash S$ instead of S) to the current VC and continuously perform the same operation. The pseudo-code for this algorithm is described in Algorithm 2.

Algorithm 2: Approximation Algorithm - Greedy Independent Cover

```
Data: Graph $G = (V, E)$
Result: Vertex Cover of $G$
$C \longleftarrow \emptyset$
while $E \neq \emptyset$ do
    select vertex $u$ with minimum degree
    $C \longleftarrow C \cup N(u)$
    $V \longleftarrow V - (N(u) \cup \{u\})$
end
return $C$
```

4.2.2 Approximation Ratio: The worst-case approximation ratio is $\frac{\sqrt{\triangle}}{2}$, where $\triangle$ is the maximum degree of the graph [5].

4.2.3 Time and Space Complexity: The while loop runs for at most $O(|E|)$ times, and all the operations in the inner loop can be performed in $O(|V|)$ time. Finding the vertex with minimum degree, adding vertices to the current VC and removing vertices from the graph all take at most $O(|V|)$ time. Therefore, the time complexity of the algorithm is

$O(|V| * |E|)$. Storing the vertex cover takes at most $O(|V|)$ space and storing the graph takes at most $O(|V| + |E|)$ space (using a adjacency list). Therefore, the overall space complexity is $O(|V| + |E|)$.

4.3 Local Search - Simulated Annealing

4.3.1 Description. Simulated annealing (SA), which is based on the concept of the physical process of annealing, is used to roughly approximate the global optimum of the target function [1]. When the temperature is high, there is a larger search space to be examined, which is subsequently condensed to a smaller number of likely spaces when the temperature drops. The principle of Monte Carlo sampling introduces unpredictability, preventing the algorithm from becoming stuck in local solutions. This is how the algorithm is implemented: (1) All the unnecessary vertices are eliminated to provide the basic solution. This is achieved by starting from the whole set of vertices and removing the higher degree vertices (if all their neighbours are in the VC) continuously until no other vertex can be removed. (2) The original solution is changed by adding a new vertex at random and deleting a vertex from the present solution that is now covered by the vertex cover. (3) At each iteration, a better optimal solution is chosen by comparing the size of the vertex cover of the current solution and the candidate solution. Depending on the likelihood, which is inversely proportional to the quality gap between the best and current solutions, and the temperature, the new solution is either approved or rejected. The temperature drops at a rate of 0.95 from an initial value of 0.8 [3]. Simulated annealing may be used to achieve a reasonable solution quickly, and it can be done using parallel computing, which can considerably boost speed. On the other hand, while the sub-optimal answer is usually sufficient in most situations, the optimal solution cannot be guaranteed. **Cost function:** Amongst the candidate vertex covers, we select the candidate solution whose vertex cover has the least number of vertices. The pseudo-code for this algorithm is described in Algorithm 3.

Algorithm 3: Local Search 1 - Simulated Annealing

```
Input: Graph $G = (V, E)$, initial solution $S_{init}, cutoff$
Output: Vertex Cover of $G$ ($S_{return}$)
$T = 0.8$
$S_{best} \longleftarrow S_{init}$
$S_{return} \longleftarrow \emptyset$
while $runningtime < cutoff$ do
    $T = T * 0.95$
    if $S$ is a VC then
        $S_{return} \longleftarrow S$
    end
    randomly delete a vertex $u \in S$
    randomly add a vertex $u \notin S$
    if $|S| < |S_{best}|$ then
        $p = \frac{exp(|S|-|S_{best}|)}{T}$
        $\alpha$ = uniform sampling from $[0, 1]$
        if $\alpha > p$ then
            $S_{best} \longleftarrow S$
        end
    end
end
return $S_{return}$
```

4.3.2 Time Complexity and Space Complexity. Given a graph $G(V, E)$ where V is nodes and E is edges, time complexity is $O(|V|)$, since we consider all the node's neighbours when the temperature is high and that is the worst case of the algorithm. To store the graph, takes $O(|V| + |E|)$ time using an adjacency list, therefore the space complexity is $O(|V| + |E|)$.

4.4 Local Search - Hill Climbing

4.4.1 Description. Hill Climbing is a heuristic local search algorithm that is used for mathematical optimization in the field of Artificial Intelligence. When it is provided with a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. However, this solution might not be the global maximum/minimum. 'Heuristic search' means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in a reasonable amount of time. A heuristic function is one that will rank all the possible alternatives at any branch in the search algorithm based on the info already available to it. It helps the algorithm to select the best route out of possible routes. The implemented algorithm has a similar approach to the min-conflicts hill-climbing heuristic problem [2]. Broadly Hill-Climbing examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as the next node. The two-stage exchange strategy implemented, yields an efficient two-pass move operator for MVC local search algorithms, which significantly reduces the time complexity per step. The forgetting mechanism enhances the edge weighting scheme by decreasing weights when the averaged weight reaches a threshold, to periodically forget earlier weighting decisions. The algorithm uses the Edge Deletion (ED) Approximation algorithm in [5] to compute the initial solution. The ED algorithm starts by considering the whole set of vertices as the initial VC. Then, the algorithm randomly chooses an edge and removes their associated vertices from VC, and deletes them from the graph. The algorithm repeats the same steps continuously until removing any more vertices makes the set of vertices not a VC. Also, as mentioned earlier, with this algorithm, the optimal solution cannot be guaranteed, but the sub-optimal is usually good enough in most cases. **Cost function:** Amongst the candidate vertex covers, we select the candidate solution whose vertex cover has the least number of vertices. The pseudo-code for this algorithm is described in Algorithm 4.

4.4.2 Time Complexity and Space Complexity. Given a graph $G(V, E)$ where V is nodes and E is edges, R and A denote the candidate vertices for Removing and Adding separately, the time complexity per step for selecting the vertex pair separately is $|R| + |A|$ ($|R|$ and $|A|$ in worst case $\in O(|V|)$) so the time complexity is $O(|V|)$; since we search for all the neighbor of a node when the cost is high and that's the worse case. The space complexity is $O(|V| + |E|)$ since we store the information of the graph.

5 EMPIRICAL EVALUATION

In this section, we explain the environment and the experimental setup of the project, and we analyze the performance of the algorithms on various different graphs.

5.1 Environment Setup

All the algorithms were implemented in Python 3.7 and we used the time, numpy, random and networkx libraries for the project. The networkx graph object was used across all the algorithms and the algorithms were test on the same PC, which has a Intel Core i7-7500 @ 2.70GHz and 12 GB RAM to compare the result fairly. We measure the time taken to reach a solution, the size of the VC and the vertices in the VC for all the algorithms across different graphs selected from

Algorithm 4: Local Search 2 - Hill Climbing

```
Input: Graph $G = (V, E)$, cutoff
Output: Vertex Cover of $G$ ($C^*$)
START
initialize edge weights and $dscores$ of vertices;
initialize the $confChange$ array as an all-1 array;
construct $C$ greedily until it is a vertex cover;
$C^* \longleftarrow C$;
while $runningtime < cutoff$ do
    if there is no uncovered edge then
        $C^* \longleftarrow$C;
        remove a vertex with the highest $dscore$ from $C$;
        continue;
    end
    choose a vertex $u$  $C$ with the highest $dscore$, breaking ties in favor of the oldest one;
    $C \longleftarrow C_u$; $confChange(u) \longleftarrow 0$; $confChange(z) \longleftarrow 1$ for each z ∈ N(u) ;
    choose an uncovered edge $e$ randomly;
    choose a vertex v ∈ e such that $confChange(v) = 1$ with higher $dscore$, breaking ties in favor of the older one;
    $C \longleftarrow C \cup v$; $confChange(z) = 1$ for each z ∈ N(v) ;
    $w(e) \longleftarrow w(e) + 1$ for each uncovered edge $e$ ;
    if $w \geq \gamma$ then
        $w(e) \longleftarrow \rho.w(e)$ for each edge $e$;
    end
    return $C^*$
end
STOP
```

the 10th DIMACS challenge. We execute the Branch and Bound and Approximation algorithm only once for all graphs but execute the local search algorithms 5 times for each graph to account for the randomness in the algorithms and reduce the bias in the results. Subsequently, we report the average time taken and average size of the VC for these local search algorithms. We consider a cutoff time of 300 seconds for the Branch and Bound, and Approximation algorithms, but use the following cutoff times for the local search algorithms: {20, 40, 60, 80, 100}. The running times and associated sizes of minimum vertex covers found for all the graph instances are presented in Table 2.

5.2 Branch and Bound

Branch and bound algorithm goes through all possible combinations in order to arrive at the optimal solution, so this algorithm will definitely provide the best solution. For smaller problems, the time taken is not very high, but as the size of the problem increases, the time taken to go through the search tree increases exponentially ($O(2^n)$). This would be the main disadvantage of branch and bound. When branch and bound is implemented with a time constraint, it may not necessarily return the optimal solution, but returns the best solution it could find within the time limit. Based on the experiment conducted using this algorithm on various graphs, the worst relative error achieved was 6.7% for the star graph, whereas the best relative error achieved was 0%. The graph that required the maximum amount of time for computation was as-22july06, which required 112.66 seconds. The minimum amount of time taken was 0.00096 seconds for the karate graph. Thus, the performance of branch and bound algorithm was fair with an average relative error of 2.1%. The time consumed, length of optimal vertex cover, along with relative error for all the other graphs are tabulated in Table 2.

5.3 Approximation Algorithms

The GIC approximation algorithm achieved almost similar performance compared to the Branch and Bound algorithm in terms of the quality of the VC, but took much less time to get to the solution. The GIC approximation algorithm has a worst case relative error of 1.6% on the delaunay_n10 graph and takes at most 299.86 seconds (around 5 minutes) for the as-22july06 graph. Except the as-26july06 graph where the approximation algorithm used almost all of its time, in all other graphs it was able to reach a solution quickly. Especially on smaller graphs with number of vertices less than 1000, the approximation algorithm was able to reach a solution under 1 second. The time taken to reach a solution increased with the increase in the number of vertices, but no pattern was observed between the time taken and number of vertices. The optimal values, solutions obtained by the approximation algorithm and (worst-case) approximation bounds for all the graphs are compared in Table 1. Interestingly, the solutions obtained by the approximation algorithm was well within the approximation bounds and was mostly closer to the optimal solution. Note that, the GIC algorithm was specifically chosen for their good performance at the cost of more time after comparing them with the MDG and ED approaches. While the GIC algorithm achieved great performance for these sets of graphs, the performance is not guaranteed to extend to different graphs. Hence, testing multiple possible approximation algorithms on a subset of graphs and choosing the best one in terms a solution quality and time taken can be a potential strategy to consider.

Table 1. Performance of the GIC Approximation Algorithm

Graph	Optimal	Approximation	Max Degree ($\triangle$)	$H(\triangle) * OPT$
jazz	158	159	100	790
karate	14	14	17	29
football	94	95	12	163
as-22july06	3303	3303	2390	80738
hep-th	3926	3928	50	13881
star	6902	6943	2109	158483
star2	4542	4571	1531	88860
netscience	899	899	34	2622
email	594	597	71	2503
delaunay_n10	703	714	12	1218
power	2203	2205	19	4802

5.4 Local Search - Simulated Annealing

Among the implemented algorithms, Simulated Annealing (SA) consistently outperformed the others in terms of relative error. On virtually all graphs, it was able to provide optimal outcomes, with non-optimal solutions only starting to show up on bigger graphs like the star. There isn't a clear relationship between the run time of the algorithm and the size of the graph; one explanation might be that performance also depends on the graph's structure. Even though the graph is small, it might take a very long time if there are too many edges. On the other hand, if there are few edges, a large number of nodes could be processed quickly.

For the QRTD plots, we have selected 0.005%, 0.01%, 0.02%, 0.05%, 0.1% are selected for power, and 0.5%, 1.0%, 1.5%, 2.0%, 3.0% are selected for star2 as the solution quality. From Figures 1 and 2, we can observe that SA is able to achieve high-quality solution within a short amount of time even when the graph is large. For example, SA achieves a 0.05%

solution quality for the Power graph for all its seeds in just 40 seconds. Similarly, for the star2 graph, SA achieves 2.0% solution quality for 100% of its seeds in 60 seconds. Also, from the SQDs in Figures 3 and 4, where cutoff times of 20s, 40s, 60s, 80s, 100s is selected for power and star2, we can observe that SA algorithm reaches a good quality solution quickly and then the performance saturates. For example, in Figure 3, we can see that a 0.02% solution quality is obtained for 20s cut off time itself, therefore there is no need for a higher cutoff time. However, the initial solution the algorithm starts with also affects its performance. If we start from a good quality solution, finding improvements will be difficult, but if we start from a relatively bad quality solution the we can improve the solution iteratively, but the might not be significant differences in the overall final solution. Note that here the SA algorithm on the Power graph achieves a solution quality of 0.1% for 80% of the seeds in 20 seconds. On the other hand, from Figure 4, we can observe that having higher cutoff times can give better quality solutions. Therefore, we conclude that whether or not higher cut off times lead to better solutions is dependent on the nature of the graph and the initial solution.

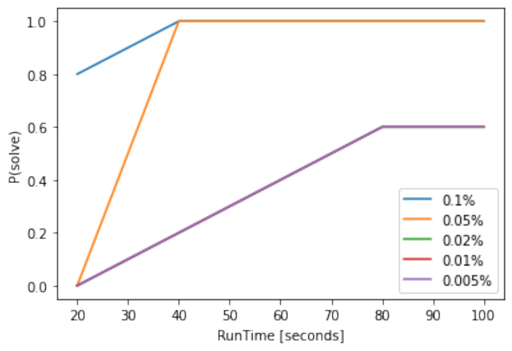

Fig. 1. Simulated Annealing - Qualified Runtime Distribution on Power Graph

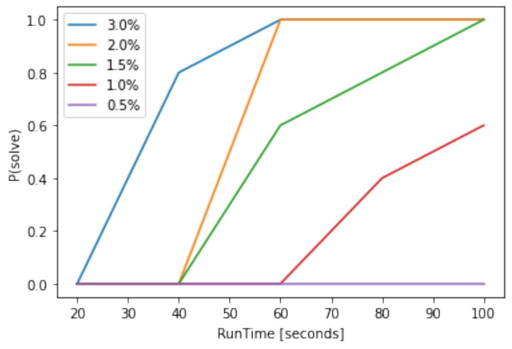

Fig. 2. Simulated Annealing - Qualified Runtime Distribution on star2 Graph

5.5 Local Search - Hill Climbing

The Hill Climbing algorithm performs poorer in terms of relative error compared to the Simulated Annealing Local Search algorithm when the number of nodes increases (as-22july06, hep-th, star, star2). It was able to achieve optimal

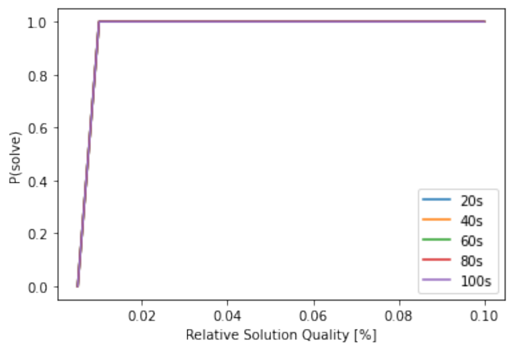

Fig. 3. Simulated Annealing - Solution Quality Distribution Graph on Power Graph

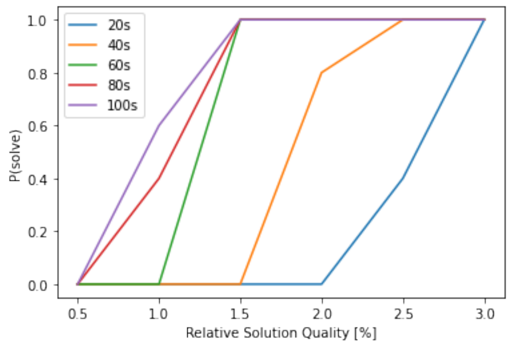

Fig. 4. Simulated Annealing - Solution Quality Distribution Graph on star2 Graph

results for almost all graphs with non-optimal solutions appearing only on larger graphs. It is interesting to note that for smaller graphs, the run time of the Hill Climbing algorithm is either better or comparable to that of Simulated Annealing, while for larger graphs HC takes a longer time to run (except star) and ends up with a higher relative error than SA. So, HC might be more suited for smaller graphs and SA might be more efficient for larger graphs. An interesting observation that was noted during the HC implementation is that, if the size of the vertex cover plateaus, the algorithm terminates and does not go into a deeper search space. Using iterated local search or random restart might help mitigate some of these problems.

For the QRTD plots, we have selected 0.005%, 0.01%, 0.02%, 0.05%, 0.1% are selected for power, and 0.5%, 1.0%, 1.5%, 2.0%, 3.0% are selected for star2 as the solution quality. From Figures 5 and 6, we can observe that the quality of the solution improves with increase in cut off times. Especially, in Figure 6 for the star2 graph, we can observe that at 80 seconds all the seeds achieve the 2% solution quality but it is achieved in 60 seconds in SA (Figure 2). This difference can be attributed to the initial solution qualities, SA starts from a really good solution but HC does not (no seed achieves a 0.1% solution quality until 40 seconds), therefore we can see the solution quality improve in HC. Figures 7 and 8, highlight the solution quality distributions for the power and star2 graphs where the cutoff times are 20s, 40s, 60s, 80s, 100s. Comparing Figures 3 and 7, we can observe that while all the seeds achieve a 0.01% solution quality for 20 seconds cutoff for SA, only 20% seeds achieve the same quality in HC for the same cutoff. Therefore, the initial solution significantly impacts the performance of the algorithm.

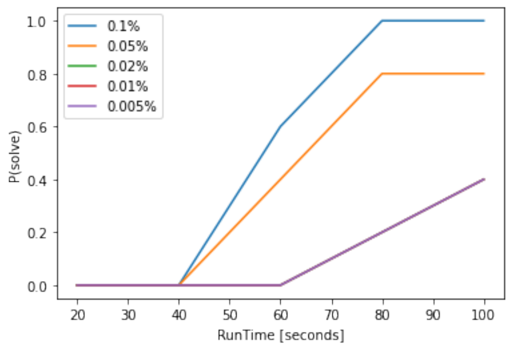

Fig. 5. Hill Climbing - Qualified Runtime Distribution on Power Graph

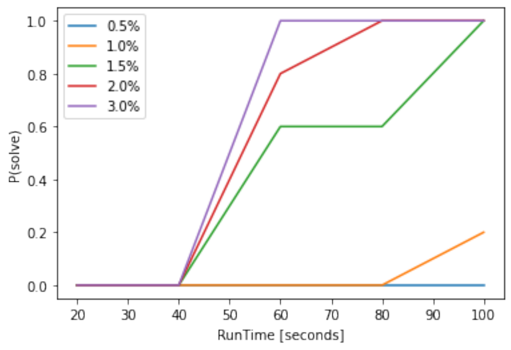

Fig. 6. Hill Climbing - Qualified Runtime Distribution on star2 Graph

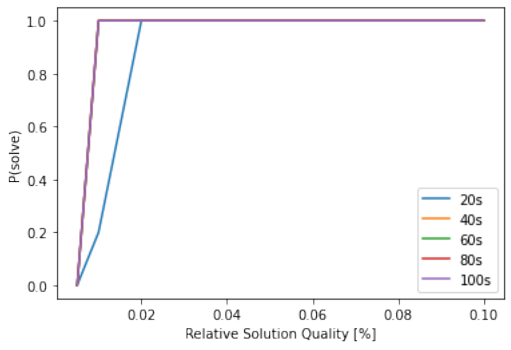

Fig. 7. Hill Climbing - Solution Quality Distribution Graph on Power Graph

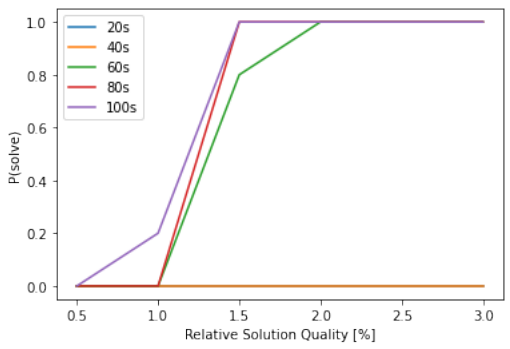

Fig. 8. Hill Climbing - Solution Quality Distribution Graph on star2 Graph

Run time Comparison: As the local search algorithms are not deterministic and involves a lot of randomness, we compare the run times of the algorithms to understand how their run time varies for different random seeds. Figures 9 and 10 summarize the run-time of the SA and HC algorithm on the power and star2 graphs using a box plot. For the power graph, there is significant difference between the run times of SA and HC. SA takes around 30-40 seconds with some deviations, whereas HC takes around 70-80 seconds. SA is almost twice as quick as HC for the power graph, however the variance in the individual run times is reasonable. For the star2 graph, both the approaches have identical run times around 97-99 seconds but SA has a lot of variance compared to HC.

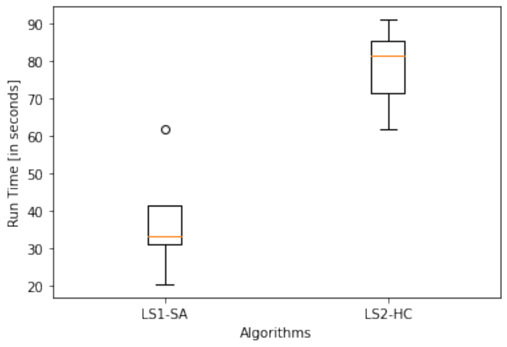

Fig. 9. Boxplot of Local Search Running Times on Power Graph

6 DISCUSSION - COMPARATIVE ANALYSIS

The Branch and Bound algorithm performed poorly compared to approximation and local search algorithms on this set of graphs. However, the BnB algorithms is the only one that guarantees optimal solution, albeit at the cost of exponential time (w.r.t the number of vertices). Especially, in large graphs the relative error is almost 10 times when compared to the approximation and local search algorithms. This is because the branch and bound algorithm has to analyze all possible sets of vertices to find the optimal solution, and when the time limit is not very high, the performance of the BnB algorithm is sub-optimal. However, in smaller graphs the performance remains comparable with approximation

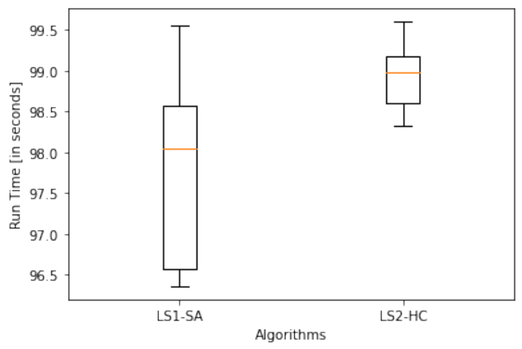

Fig. 10. Boxplot of Local Search Running Times on Star2 Graph

algorithms.

The approximation algorithms perform exceedingly well on these set of graphs, but there might be bias in these results as the GIC algorithm was explicitly chosen because of its good performance. As discussed previously, the performance of the approximation algorithms is comparable with the local search algorithms and better than BnB. There are specific graphs such as star, karate, as-22july06, and netscience where they either match or achieve better solution quality than the local search algorithms. However, for smaller graphs such as football, the approximation algorithms are not able to achieve the optimal solution. This is the major disadvantage of using the approximation algorithm, while we reach a reasonable solution quickly, we can't improve from there even with enough time.

The local search algorithms helps in solving the above problem. They start from some initial solution that is obtained through some heuristics (such as using an approximation algorithm). However, they have the capability to iteratively improve the solution by searching their neighbourhood. We can observe that the local search algorithms achieve the best solution quality for all the graphs except the star graph, however it takes more but limited time (input by the user) to arrive at these solutions. Comparing the local search algorithms, we observe that the simulated annealing approach takes lesser time for larger graphs and hill climbing algorithms takes lesser or comparable time for smaller graphs. More analysis across other graphs are required to find any robust conclusions. However, for almost all graphs the hill climbing approach quickly finds solutions during the initial 10-15 seconds, it only takes 0.001 seconds to find the next best solution but eventually this starts to take more time. In contrast, the simulated annealing approach takes 0.1 seconds to find the next best solution initially and this eventually slows down as well. But this is not a fair comparison as the initialization technique used to find the initial solution is different for these two approaches, therefore, for a more reliable comparison we must use the same initial solution.

In conclusion, BnB algorithm is really useful only when an optimal solution is definitely required. The exponential time complexity of the algorithms can be a huge drawback for large real-world graphs and making them not practically applicable. The approximation algorithms can be used when a solution is quickly required but the quality is not an issue. However, there isn't a one-fit-for-all approximation algorithm that performs really well across all graphs, so we must try different approximation algorithms and compare their performances. Local search are the best class of algorithms when both solution quality and time are important. They start from some initial solution thus reducing their complexity and can iteratively improve depending on how much time they have. Therefore, for real-world problems local search algorithms might be the best fit. They don't slow down too much on large graphs and therefore can be applicable on large real-world graphs.

Table 2. Comparative Analysis of Algorithms

	BnB			Approx			LS1 - SA			LS2 - HC		
Dataset	Time(s)	VC Value	RelErr	Time(s)	VC Value	RelErr	Time(s)	VC Value	RelErr	Time(s)	VC Value	RelErr
jazz	0.036	160	0.013	0.0050	159	0.0063	0.051	158	0.0000	0.0285	158	0.0000
karate	0.00096	14	0.00	0.0010	14	0.00	0.00	14	0.0000	0.0034	14	0.0000
football	7.70	95	0.011	0.0020	95	0.011	0.047	94	0.0000	0.0124	94	0.0000
as-22july06	112.66	3312	0.0027	299.83	3303	0.00	34.384	3303	0.0000	97.4125	3323	0.0061
hep-th	37.14	3947	0.0053	14.53	3928	0.00051	43.956	3926	0.0000	93.1885	3944	0.0046
star	98.91	7366	0.067	26.36	6943	0.0059	95.148	6963	0.0089	10.6985	6964	0.0089
star2	107.09	4677	0.030	102.27	4571	0.0064	97.819	4548	0.0013	98.9368	4593	0.012
netscience	1.51	899	0.00	0.41	899	0.00	0.0	899	0.0000	2.1680	899	0.0000
email	0.88	605	0.019	0.32	597	0.0051	4.235	594	0.0000	1.8106	594	0.0000
delaunay_n10	1.13	740	0.053	0.16	714	0.016	17.442	703	0.0006	4.5848	703	0.0000
power	11.05	2272	0.031	5.87	2205	0.00091	37.5581	2203	0.0000	78.0591	2204	0.0005

7 CONCLUSION

In this project, we implemented four different algorithms for solving the MVC problem, namely: branch and bound, approximation algorithms, local search 1 - simulated annealing, and local search 2 - hill climbing. The pseudo codes, space and time complexities, and approximation ratio (if exist) were elaborately discussed for all the algorithms. The run time and solution quality of these algorithms where compared and analyzed across 11 different graphs from the 10th DIMACS challenge. Also, the QRTD, SQD, and box plots were generated and analyzed for both the local search algorithms on the "power" and "star2" graphs. The following are the possible future extensions of this project:

(1) Analyze more approximation algorithms and heuristics to understand whether certain heuristics work well for a subset of graphs with certain properties.
(2) Explore Tabu search and Iterated Local Search algorithms to determine whether they can improve the performance of the local search algorithms.
(3) Understand how various heuristics such as cost function, thresholds (used in simulated annealing), vertex swap techniques affect the run time and solution quality.

REFERENCES

[1] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. “Optimization by simulated annealing”. In: *science* 220.4598 (1983), pp. 671–680.

[2] Steven Minton et al. “Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems”. In: *Artificial Intelligence* 58.1 (1992), pp. 161–205. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(92)90007-K. URL: https://www.sciencedirect.com/science/article/pii/000437029290007K.

[3] Walid Ben-Ameur. “Computing the initial temperature of simulated annealing”. In: *Computational optimization and applications* 29.3 (2004), pp. 369–385.

[4] Xinshun Xu and Jun Ma. “An efficient simulated annealing algorithm for the minimum vertex cover problem”. In: *Neurocomputing* 69.7 (2006). New Issues in Neurocomputing: 13th European Symposium on Artificial Neural Networks, pp. 913–916. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2005.12.016. URL: https://www.sciencedirect.com/science/article/pii/S0925231205003565.

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

[6] Rafael Martı, Micael Gallego, and Abraham Duarte. “A branch and bound algorithm for the maximum diversity problem”. In: *European journal of operational research* 200.1 (2010), pp. 36–44.

[7] César Rego et al. “Traveling salesman problem heuristics: Leading methods, implementations and latest advances”. In: *European Journal of Operational Research* 211.3 (2011), pp. 427–441.

[8] Shaowei Cai, Kaile Su, and Abdul Sattar. “Two new local search strategies for minimum vertex cover”. In: *Twenty-Sixth AAAI Conference on Artificial Intelligence.* 2012.

[9] Shaowei Cai et al. “NuMVC: An efficient local search algorithm for minimum vertex cover”. In: *Journal of Artificial Intelligence Research* 46 (2013), pp. 687–716.

Received Dec 2022