Algorithms to Compute the Minimum Vertex Cover

AAROHI SHAH, Georgia Institute of Technology, USA
ARMAN AFSHAR, Georgia Institute of Technology, USA
PRATHIK KAUNDINYA, Georgia Institute of Technology, USA
SAURABH DOODHWALA, Georgia Institute of Technology, USA
GROUP 22

Additional Key Words and Phrases: graph algorithms, approximation, simulated annealing, branch and bound, hill-climbing

1 INTRODUCTION

The Minimum Vertex Cover (MVC) is a well-known NPC problem. The problem is to find a set with minimum number of vertices that each edge has atleast one pint in the set. For this purpose, we implement four algorithms. *Branch-and-Bound* searches an accurate MVC for the given graph at the expense of computational efficiency whereas the *Approximation* method finds a reasonable answer in a given limited time. Further, we implement 2 local search techniques, namely *Hill Climbing* and *Simulated Annealing*. The hill climbing method greedily removes all the vertices it can, to reach a local minima, and performs good as is discussed in results and discussion section. The simulated annealing method tolerates some worsening moves in the beginning and lowers the tolerance probability gradually and finally produces a good result. Hence, these results are consistent with the the theoretical understanding of these algorithms studies during the class. We performed all these experiments on realistic datasets provided along with the project description.

2 PROBLEM DEFINITION

This section has the formal mathematical formulation for the MVC problem. Consider an undirected graph, $G = (V, E)$ having V number of nodes/vertices and E are the edges.
A vertex cover is a subset $C \subseteq V$ such that $\forall (u, v) \in E : u \in C \vee v \in C$. Based on this, MVC is to compute the minimum $|C|$ that satisfy the above condition of a vertex cover.

3 RELATED WORK

In Computer Science, the problem of finding a minimum vertex cover is considered a classical minimization problem. It is a perfect example of NP-Complete optimization problem. This problem also serves as a model for many real-life and theoretical problems. Some examples of the areas where this problem occurs in realistic scenarios are bioinformatics, communications, operations research, [8, 10] etc. MVC is also closely related to many other graph problems such as the Edge cover problem or the independent set problem. Exact algorithms can be used to find the solution for such a problem. However, these approaches are good only for small graphs due to the increase in computational inefficiency as the dataset/graph becomes large. Hence there are many alternate algorithms that don't entirely search the problem space to reduce the amount of time to find the solution. One of the methods is branch and bound to analyse all the possible solutions and find a solution that guarantees optimality within justifiable bounds. This method does help us get the exact solution but can still have exponential times. Many studies have been conducted to improve this method, either by efficient backtracking or by employing parallel computing algorithms [3, 11].

Authors' addresses: Aarohi Shah, Georgia Institute of Technology, Atlanta, Georgia, USA, aarohishah94@gatech.edu; Arman Afshar, Georgia Institute of Technology, Atlanta, Georgia, USA, __@gatech.edu; Prathik Kaundinya, Georgia Institute of Technology, Atlanta, Georgia, USA, prathik.k@gatech.edu; Saurabh Doodhwala, Georgia Institute of Technology, Atlanta, Georgia, USA, __@gatech.edu; GROUP 22.

Many alternative algorithms which perform in polynomial time have been established such as the Depth First Search [12], Edge Deletion [7], Maximum Degree Greedy [4], Iterated Local Search Algorithms [13], List Left and List Right algorithms [1, 6], etc. These methods have approximation ratios between 2 and the maximum degree of the dataset/graph. Further, a lot of work has also been done in parameterized vertex cover problem. Given a graph and a parameter (p), this approach finds the vertex cover of the graph with at most p vertices, hence is the decision problem version of MVC [2]. Hence, considering the enormous amount of work going on for this problem, it is evident that understanding and analysing this class of problems is essential.

4 PARSING INPUT

We define a common utility function to parse the input for all routines. Specifically, the inputs to the function are the file name and directory path. The function then opens a file stream and reads the data from the graph file. A NetworkX graph is produced as an output, which is returned to the calling algorithm.

5 ALGORITHMS

5.1 Branch and Bound:

5.1.1 Description: The idea of the Branch and Bound algorithm is to find optimal solution of the vertex cover problem by searching through smaller subsets of the main graph G. We explore each and every possible condition with the help of a tree structure and arrive at the most optimal solution by discarding branches that do not satisfy the bounds to the optimal solution. We start with an appropriate assumption for the bounds such that we are sure of optimal solution in that range. These bounds are continually updated to tighten the list of possible solutions. For the present implementation of the Branch and Bound algorithm, the upper bound is the best solution (Minimum Vertex Cover size) encountered so far, during exploration of the tree. We have assumed lower bound in the following way:

$$\text{Lower Bound} = \frac{\text{Total number of edges in the graph}}{\text{Maximum number of node degree for the graph}}$$

The reasoning behind this is that for a graph with E edges, a vertex cover of size v and maximum number of node degree as m, we can say that $m * v \geq E$, which gives us the following result $v \geq \frac{E}{m}$

5.1.2 Algorithm: Here, the algorithm considers vertices in decreasing order of their degree, i.e. the number of edges connected to a vertex and takes the highest degree node as the most promising candidate solution for minimum vertex cover solution. We assign a candidate vertex to a state 1, if it is present in Vertex Cover solution, otherwise it's given a state 0. Based on the solution obtained at node, we decide further. If a complete solution is found, we update upper bound if required and backtrack to find other solutions. If a partial solution is found, which looks promising, we keep going further in that branch. If an infeasible solution is obtained, we just prune the branch there and backtrack.

5.1.3 Pseudo Code: Branch and Bound Algorithm: The pseudo-code for Branch and Bound Algorithm is shown below:

Algorithm 1 MVC: Branch and Bound

```
Input: Graph G with V vertices and E edges
Output: Vertex Cover Set for graph G
$G \leftarrow (V, E)$
$VC \leftarrow \Phi$
$a \leftarrow$ vertex having highest node degree m
Frontier $\leftarrow [(a, 0, (-1, -1)), (a, 1, (-1, -1))]$
Upper Bound $\leftarrow |V|$
OPT $\leftarrow$ Upper Bound
while Frontier $\neq \Phi$ do
  (CN, state, parent) = Frontier.pop()
  $VC \leftarrow (CN, state)$
  backtracking = false
  if state == 0 then
    Update state = 1 for every neighbor vertex of CN
    $G \leftarrow G \setminus$(All neighbors of CN)
  end if
  if state == 1 then
    $G \leftarrow G \setminus$(CN)
  end if
  if No Edges left in Graph G then
    if |VC| < Upper Bound then
      Upper Bound $\leftarrow$ |VC|
      OPT $\leftarrow$ |VC|
    end if
    backtracking = true
  else
    if |VC| + LowerBound(G) < Upper Bound then
      newnode $\leftarrow$ Vertex with maximum degree among the remaining vertices
      Frontier $\leftarrow$ Frontier $\cup$ [(newnode, 0, (CN, state)), (newnode, 1, (CN, state))]
    else
      backtracking = true
    end if
  end if
  if backtracking == true and Frontier $\neq \Phi$ then
    newparent $\leftarrow$ Last element of Frontier
    if newparent belongs to VC then
      position $\leftarrow$ take position value of newparent in VC
      n = |VC|
      for j = position to n do
        Delete element VC(j) from VC
        Add VC(j) and connected edges to graph G
      end for
    else
      Set VC $\leftarrow \Phi$ again
      Set G $\leftarrow$ original G again
    end if
  end if
end while
Return OPT
```

5.1.4 Time and Space Complexity: As we can see from the pseudo code, while loop will check all the elements of the Frontier (maximum will be equal to V vertices) and each element can have either state 0 or state 1. Total number of possible scenarios are $2^{|V|}$, hence worse case running time complexity will be $O(2^{|V|})$.
Here, we will need E (number of edges) space to calculate vertex cover set and |2V| space to store all elements in Frontier at once. Hence, our space complexity will be $O(|V| + |E|)$.

5.2 Approximation

5.2.1 Description. For extremely large instances of the vertex cover problem, on the scale of millions of nodes, approximation algorithms using heuristics are usually employed. Although they do not always yield an optimal solution, they are still valuable when the error in the results is "small enough" (the meaning of which depends on the specific application) In order to obtain a solution to the vertex cover problem in a computational efficient manner, we implement an approximation algorithm with a constructive heuristic. Construction heuristics algorithms typically use some kind of a greedy approach to construct the final solution. Again, as mentioned earlier the solutions by these algorithms will not be globally optimum, but these simulations run much faster than branch and bound and provide approximate solutions.

5.2.2 Implementation. In this work we implement this by staring with an empty set and populating it by using appropriate conditions over time. Hence, we pick an edge (E*) and add the node associated with it to the set (which is getting updated) followed by removal of all edges which are incident with either of the two nodes of the edge that was picked (E*). Note that this is opposite to the approach for local search wherein we start with a fully populated initial heuristic, and update the solution by removing nodes that already satisfy the vertex cover. Also for this implementation, it can also be shown [5] that this approximate algorithm never finds a vertex cover whose size is more than twice the size of the minimum possible vertex cover. In terms of complexity it is evident that the time complexity of $O(V + E)$ exists, and is achieved for such an approach.

5.2.3 Pseudo-code. Approximation

Algorithm 2 MVC: Approximation Algorithm

```
$VC \leftarrow \Phi$
$E \leftarrow E[G]$
while $E \neq \Phi$ do
  $(u, v) \leftarrow$ random edge from $E$
  $VC \leftarrow VC \cup \{u, v\}$
  for $e \in Edges$ do
    if $u \in e$ or $v \in e$ then
      Delete $e$
    end if
  end for
end while
return $VC$
```

5.3 Hill Climbing

Hill-climbing is a standard local search algorithm that typically obtains a local optimum solution. The local minimum is typically obtained by making an iterative update to the existing solution, with the algorithm making small steps toward the solution. For optimization problems involving convex functions, hill-climbing is equivalent to a global optimization; however, in most non-convex optimization applications (such as the one in this problem), only a local optimum solution is guaranteed if the algorithm is able to finish. The approach adopted in this project for the implementation of this algorithm is explained below followed by the pseudo-code:

(1) First, a set and priority queue containing all nodes in the original graph G are instantiated. The set is considered as the smallest obtained VC up to a certain point in the computation, so is dynamically updated throughout the algorithm. The priority queue is ordered such that nodes in the front (i.e. that are first to be popped) are those that have the lowest node degree in the entire graph. As this is a one time operation, it incurs a time cost of $O(V)$.
(2) Until the priority queue is empty, we repeatedly pop nodes from the front. During each iteration of the algorithm, all nodes with the same degree are popped, and shuffled.
(3) For each item popped, we test the removal of the node from the VC; if the VC remains intact despite removal of the node, it is permanently excluded from the final VC; else, we leave it in, and proceed to the next elements in the priority queue.
(4) This algorithm continues until i) The priority queue is empty, in which case removal of all nodes has been explored, or ii) The maximum time constraint (determined by the user) has expired. We find that setting a time limit of 600s for each run is reasonable.
(5) In addition, a solution trace is generated every time an improved solution is found, as stipulated in the project requirements.

Since the only terminating conditions of the algorithm (apart from the time limit) is the emptying of the priority queue (which initially contains all of the VC nodes), the time complexity of the HC algorithm is stipulated by this initial size and scales with the number of vertices V as $O(V)$. Moreover, since the auxiliary space used is mainly attributed to this priority queue, we have an auxiliary space complexity that also scales as $O(V)$. Note that a random seed is used in step (2) in the above procedure, to determine the shuffled order of the popped nodes.

Algorithm 3 MVC: Hill-climbing Algorithm

```
$maxtime \leftarrow$ Time limit of algorithm
$G \leftarrow (V, E)$
$P \leftarrow v_i \; \forall \; v_i \in V$ - Priority queue sorted by increasing node degree of $v_i$
while $P \neq \Phi$ and $time < maxTime$ do
    $MD \leftarrow$ set of elements popped from $P$ with same (minimum degree)
    Shuffle($MD$)
    while $len(MD) > 0$ do
        Pop $node$ from $MD$
        Delete $node$ from $VC$
        if $VC$ is invalid then
            Add $node$ back to $VC$
        end if
    end while
end while
return $VC$
```

5.4 Local Search: Simulated Annealing

5.4.1 Description. Simulated annealing typically involves selecting a neighbor candidate solution randomly and checking whether taking that move worsens the result or not and with the associated probability. The probability gradually reduces increasing the stability of results (reducing fluctuations) analogous to cooling in the annealing process. The way we implement this is for searching for a neighbor randomly, removing a vertex that is in the current solution. Then we randomly use a vertex that is not yet covered by the vertex cover set. The pricing function uses the number of edges that are not covered by the candidate solution, hence evaluating the quality of the candidate. It is important to understand that this algorithm does not guarantee an optimal solution, however for the cases/ graphs we used, we obtain optimal or near optimal solution and hence reasonably good solutions. Also, for cases when the input size is not very large, the algorithm is unnecessarily complex.

5.4.2 Implementation. We initialize the initial solution as the input graph's nodes and an initial temperature as 0.15 and the temperature decreasing rate as 0.95 used from established resources [9]. The fixed number of iterations after which the temperature goes down is $iter = (m - l - 1)^2$, where m is the number of nodes in the graph and l is the number of nodes in current solution. Furthermore, the max time for cut-off is set as 600s (10 minutes). Hence, the code/ operation breaks when it reaches this time even if an optimal solution is not yet found. Also, there is an automated tuning mechanism that as the iteration processes, the probability of accepting worse solution decreases. This helps us in accepting more possibilities at the beginning of the calculations and gradually reduces the scope to reject worse solutions.
Complexity analysis: Time complexity is O(n) wherein n is the maximum degree of the graph and because we scan neighbors of a vertex in each iteration of the loop. Further, the space complexity is $O(|V| + |E|)$ since we only store the information of this graph.

5.4.3 Pseudo-code. Simulated Annealing algorithm

Algorithm 4 MVC: Simulated Annealing Algorithm

```
$G \leftarrow (V, E)$
$temp \leftarrow 0.15$
$Updated\ sol \leftarrow \{\}$
sol ← Initial solution
while $time < cutoff$ do
    $temp \leftarrow 0.95 \cdot temp$
    while Search steps does not exceed fixed steps do
        if $sol$ is VC then
            $Updated\ sol \leftarrow sol$
            Remove random $v_i$ from $sol$
        end if
        Current Sol ← $sol$
        Delete random vertex from sol
        Add random vertex not in sol
        if Current sol better than sol then
            Accept sol based on probability
        end if
    end while
end while
return $Updated\ sol$
```

6 EMPIRICAL EVALUATION:

The algorithms were evaluated using Python 3.7.9 with the Anaconda distribution. For reproducibility, we have included an environment.yml file along with our submission which can be used to install the exact Anaconda environment in which we have run our algorithms. The algorithm test cases were run sequentially on a computer using a quad-core Intel i7-4610M CPU with a clock speed of 3.00 GHz and an available RAM size of 8GB. In this section, we will discuss the results of every algorithm with respect to the optimal solutions and the time they took to achieve it. Also, we have mentioned relative error for each result, to understand how far the solution is from optimal. The results presented for each algorithm correspond to runs with a random seed of 10, although the random seed is only applicable for the LS algorithms.

6.1 Branch and Bound Results

Here, we will present a table for Branch and Bound algorithm results. In general, this algorithm was very inefficient as the simulation for many large graph did not finish in the maximum time specified (1000 s). Hence, for the purpose of illustration, we pick the time value from the trace output file which reached closest to the optimal solution (thereby comparing the time at which we got solutions in close proximity to the optimal). From Table 1, we can say that for 2 graphs, we are able to achieve optimal result within seconds. For the rest, we are achieving results close to optimal within the time limit of 1000 seconds. We get a maximum relative error of 0.0672 for star.graph case. These results are very promising for the given time limit. Moreover, if we allow to run the graphs for 60 minutes, we are able to achieve the optimal solution. This is because Branch and Bound evaluates each and every case and takes exponential time while doing so.

Graph	Optimal Solution	Algorithm Solution	Time (s)	Relative Error
as-22july06.graph	3303	3312	88.28	0.0027
delaunay_n10.graph	703	740	0.6	0.0526
email.graph	594	605	0.63	0.0185
football.graph	94	95	1.15	0.0106
hep-th.graph	3926	3947	30.62	0.0053
jazz.graph	158	160	0.03	0.0127
karate.graph	14	14	0.01	0.0000
netscience.graph	899	899	1.06	0.0000
power.graph	2203	2272	11.13	0.0313
star.graph	6902	7366	71.33	0.0672
star2.graph	4542	4677	77.07	0.0297

Table 1. Branch and Bound Results

6.2 Approximation Results

Results for Approximation Algorithm are shown in Table 2. We can say that relative error is a little high for this as compared to branch and bound, but we are able to achieve solution with a short time period. For instance, for star2 graph, we achieve results (with 2-approximation) within 54 seconds for approximation, while it takes 77 seconds for Branch and Bound. Hence, as stated earlier, Approximation tries to generate a quick solution with a reasonable accuracy.

Graph	Optimal Solution	Algorithm Solution	Time (s)	Relative Error
as-22july06.graph	3303	8042	99.74	1.4348
delaunay_n10.graph	703	1018	0.6	0.4481
email.graph	594	967	0.5	0.6279
football.graph	94	111	0.01	0.1809
hep-th.graph	3926	6797	31.42	0.7313
jazz.graph	158	195	0.02	0.2342
karate.graph	14	24	0.00	0.7143
netscience.graph	899	1432	1.39	0.5929
power.graph	2203	4290	10.96	0.9473
star.graph	6902	10844	74.27	0.5711
star2.graph	4542	7580	53.31	0.6689

Table 2. Approximation Algorithm Results

6.3 LS1: Hill Climbing Results

Table 3 shows the results for Hill Climbing Local Search Algorithm. The results are very good and we obtain maximum relative error as 0.0696 for star2 graph case. It takes more time as compared to approximation algorithm, but produces more accurate results. Hence, this can be used when we can spare some more time as compared to Approximation but less when compared to Branch and Bound.

Graph	Optimal Solution	Algorithm Solution	Time (s)	Relative Error
as-22july06.graph	3303	3325	418.21	0.0067
delaunay_n10.graph	703	744	0.77	0.0583
email.graph	594	615	1.49	0.0354
football.graph	94	97	0.01	0.0319
hep-th.graph	3926	3941	47.49	0.0038
jazz.graph	158	160	0.08	0.0127
karate.graph	14	14	0.00	0.0000
netscience.graph	899	899	1.35	0.0000
power.graph	2203	2274	11.42	0.0322
star.graph	6902	7237	177.35	0.0485
star2.graph	4542	4858	502.48	0.0696

Table 3. LS1: Hill Climbing Results

6.4 LS2: Simulated Annealing Results

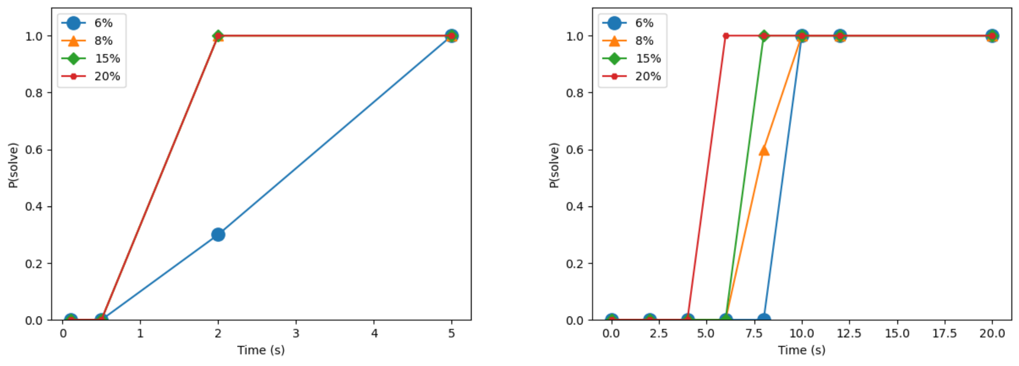
Table 4 shows the results for Simulated Annealing Algorithm. This algorithm achieves optimal solution for all cases except as-22july06, star and star2 graphs. It is one of the best performing algorithm to give optimal solutions quickly.However, it is important to note that this algorithm is highly dependent on the seed given. Hence, when we tried running the same graph with different seeds and observed high sensitivity. The results shown here and in rest of the tables are using seed 10 to ensure consistent comparison.

Graph	Optimal Solution	Algorithm Solution	Time (s)	Relative Error
as-22july06.graph	3303	3312	131.67	0.0027
delaunay_n10.graph	703	703	164.84	0.0000
email.graph	594	594	5.51	0.0000
football.graph	94	94	0.03	0.0000
hep-th.graph	3926	3926	261.78	0.0000
jazz.graph	158	158	0.12	0.0000
karate.graph	14	14	0.00	0.0000
netscience.graph	899	899	1.67	0.0000
power.graph	2203	2203	98.15	0.0000
star.graph	6902	7173	1000	0.0393
star2.graph	4542	4545	988.03	0.0007

Table 4. LS2: Simulated Annealing Results

6.5 Comparison between Hill climbing and Simulated Annealing

In this section, we describe the runtime analysis and discuss the QRTD and SQD plots for the results obtained with our two local search algorithms (hill climbing (HC) and simulated annealing (SA)). For each of these algorithms, 10 separate runs were conducted (with the random seeds varied as 10,20,30,...100), in order to obtain a rough estimate of the average performance of the algorithms. We discuss our findings in detail.

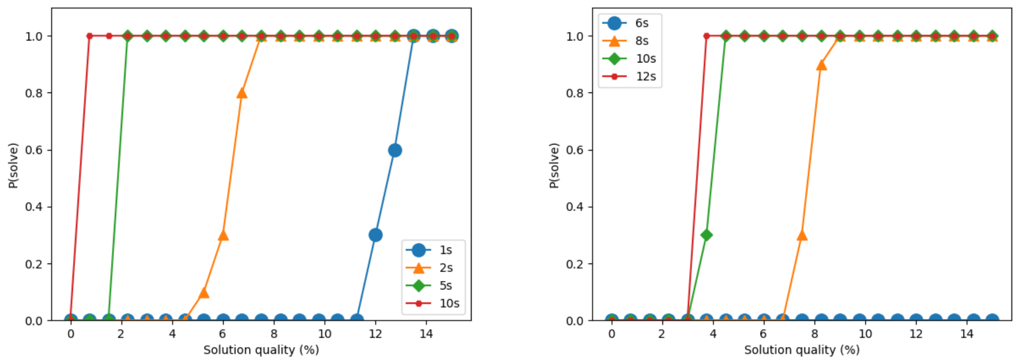


(a) Simulated annealing algorithm

(b) Hill-climbing algorithm

Fig. 1. The QRTD plots for the two local search algorithms implemented in this study. Both figures correspond to the power.graph instance.

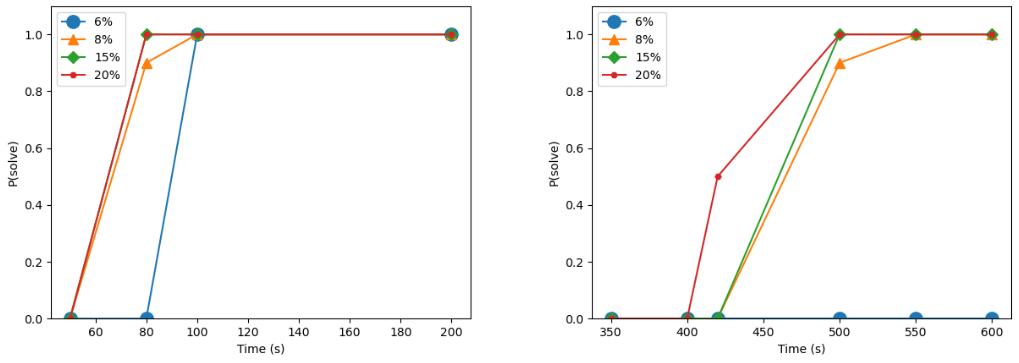
Figure 1 shows the QRTD plots corresponding to solution qualities of 6%,8%,15% and 20% above the optimal vertex cover on the power.graph instance. As expected, the higher percentages are reached earlier (since they correspond to a worse solution) for both algorithms. However, the SA algorithm appears to achieve peak performance for the three better cases (6%,8%,15%) earlier than the HC algorithm. However, it can be seen that both LS algorithms perform reasonably well (with a 5% solution quality being reached in both algorithms within 5s).



(a) Simulated annealing algorithm (b) Hill-climbing algorithm

Fig. 2. The SQD plots for the two local search algorithms implemented in this study. Both figures correspond to the power.graph instance.

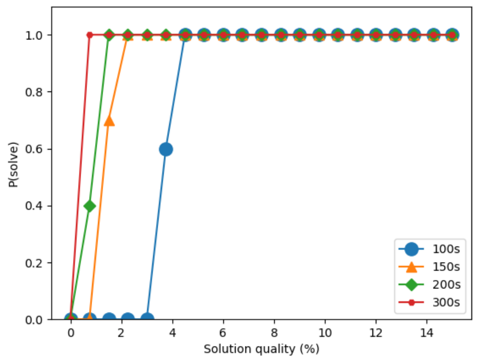
Likewise, the SQD plots shown in Figure 2 also indicate that the algorithm results are very close to the optimum within 10s of operation, for both cases. From these plots, it can also be inferred the SA algorithm appears to outperform the HC algorithm in terms of the attainable quality within a certain time. For instance, the HC algorithm was not able to reach even a 15% accuracy after a runtime of 6s. This indicates that the SA algorithm converges faster than the HC algorithm for this instance.



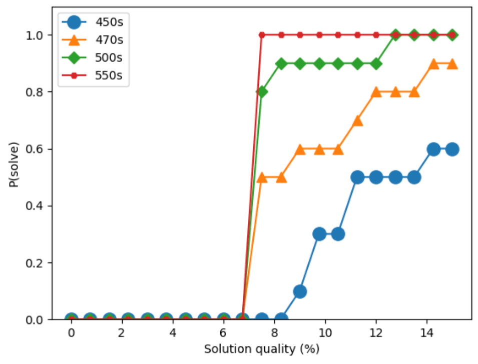
(a) Simulated annealing algorithm (b) Hill-climbing algorithm

Fig. 3. The QRTD plots for the two local search algorithms implemented in this study. Both figures correspond to the star2.graph instance.

Analyzing performance on the star2.graph instance is a better indicator of how both methods would perform on a complicated graph, since this instance has a much larger number of initial nodes and optimal VC size. In this case, the difference in speed between the two algorithms was stark; while the SA algorithm achieved all four tested accuracies within 120s, it is clear that the HC algorithm did not manage to obtain a 6% accuracy throughout the test. In fact, even the higher (worse) target solutions were obtained only after a much longer duration of time. This indicates the clear superiority of the SA algorithm in this task.



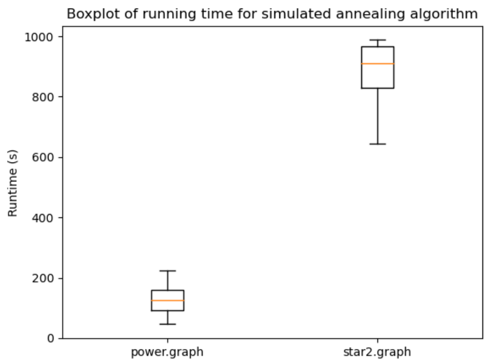
(a) Simulated annealing algorithm



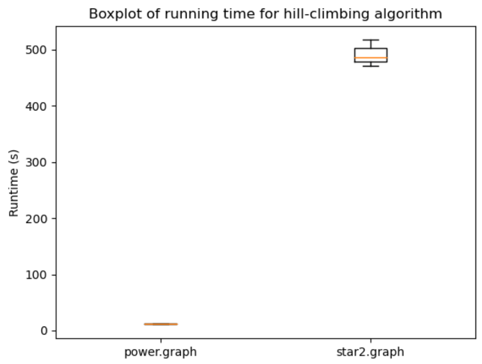
(b) Hill-climbing algorithm

Fig. 4. The SQD plots for the two local search algorithms implemented in this study. Both figures correspond to the star2.graph instance.

The SQD plot corresponding to the star2.graph instance indicates a similar inference. While all runs were able to achieve a solution within 6% of the optimal VC throughout the run in case of the SA algorithm, the HC algorithm only has all instances reaching a maximum quality of around 7%, with a much longer time taken (at least 500s). As inferred from all runtime plots presented, we see that the SA algorithm outperforms the HC algorithm on both instances mentioned. However, we observed a higher degree of variability in the total runtime for each of the SA runs when compared with the HC runs. As seen in Figure 5, the outliers for the SA algorithm in terms of runtime are much more drastic. We believe that this is observed due to the probabilistic acceptance of worse solutions, in the design of the SA algorithm. This likely introduces a higher degree of randomness in the result obtained. However, it can be observed that the outliers in terms of runtime for the star2.graph instance actually performed better in terms of runtime, reinforcing the belief that the acceptance of worse solutions at an intermediate stage could lead to potentially more efficient runs of the algorithm.



(a) Simulated annealing algorithm



(b) Hill-climbing algorithm

Fig. 5. The boxplots of the runtime of both local search algorithms implemented in this project

7 DISCUSSION

In terms of accuracy of result and running times, the local search algorithms did a good job relative to BnB and approximation approach. The BnB approach systematically evaluates a large number of candidate solutions which is dependent on the bounds and hence takes significantly more time to get the optimal solution. Our observation is that considering that this approach can give us an exact solution, it is the slowest out of the four techniques. Considering the results for the approximation method, we obtain a solution is a few seconds for all the graphs, however with compromised accuracy, going upto twice the optimal value. To that effect, for some graphs such as *jazz* and *football* wee achieve accuracy close to 20% which is reasonable. Considering these promising results, it may be a good idea to use an approximate algorithm to make an initial prediction for the solution. This initial prediction can then be used as a lower bound for the BnB in contrast to the one used for this study (ratio of edges to maximum node degree). Though this is a good choice, it may be a loose bound and the can be improved to better performance. Moreover, as mentioned earlier, there are many established studies in the literature that employ improvements to traditional BnB and can be used if accuracy of the solution is utmost important with not very high running times.

As stated before, local search did end up being a good compromise for good results as we achieved approximately 3% and less than 1% error (average for all graphs) for hill climbing and simulated annealing respectively. Moreover, in many graphs an exact solution was achieved within the stipulated time limit. However, it is important to note that these local search approaches are sensitive to the seeds and it can be helpful to run multiple cases to analyse the speed. Again, for the purpose of this study we used the entire graph as the starting point for the local search approaches, which can be easily modified to start from the solution given by approximate approach. This may lead to better and quicker solutions.

Finally, to summarize, through these experiments we observe and infer that the time complexity of the algorithms depend dramatically on the actual implementation. Hence, for real world applications, the choice of which algorithm to implement depends not only on the type of algorithm but also on implementation technique. Then based on booth these parameters, the selection can be made based on a trade off between speed and accuracy.

8 CONCLUSION

Various algorithms to solve the MVC problem have been implemented and analysed for the purpose of this project. A summary with the algorithm description, implementation details and results analysis has been presented in this report. Since MVC is an important problem and holds significant application importance, the choice of algorithm to solve a particular problem depends on the accuracy required and computational resources available. In general, for applications/ problems that need a high amount of accuracy, Branch and bound method is the best choice. However, it is important to note that the time to run such an algorithm can be significantly high as also seen in the report for the given datasets. Hence, it should be used only when there is no constraint on the running time and computing power. Alternatively, if accuracy can be sacrificed, approximation/ heuristic algorithms can be used for quicker results. Moreover, such algorithms also provide a bound on the accuracy of the result, and if that is acceptable, it can be a good choice. It was also presented that local search algorithms such as hill climbing and simulated annealing can provide results with good accuracy and less time than BnB. The accuracy is better than approximation in most of the cases, however such methods does not provide a bound on accuracy. Hence, it provides a good balance between accuracy and computational efficiency. Overall, this report provides means to understand and analyse different approaches to the MVC problem. Finally, since other NP-Complete problems can be reduced to the vertex cover problem and it helps using the algorithms presented in this report to find solutions to those other problems.

REFERENCES

[1] David Avis and Tomokazu Imamura. 2007. A list heuristic for vertex cover. *Operations research letters* 35, 2 (2007), 201–204.

[2] Jianer Chen, Iyad A Kanj, and Ge Xia. 2010. Improved upper bounds for vertex cover. *Theoretical Computer Science* 411, 40-42 (2010), 3736–3756.

[3] Jens Clausen. 1999. Branch and bound algorithms-principles and examples. *Department of Computer Science, University of Copenhagen* (1999), 1–30.

[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.

[6] François Delbot and Christian Laforest. 2008. A better list heuristic for vertex cover. (2008).

[7] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.

[8] Vasily V Gusev. 2020. The vertex cover game: Application to transport networks. *Omega* 97 (2020), 102102.

[9] Holger H Hoos and Thomas Stützle. 2004. *Stochastic local search: Foundations and applications*. Elsevier.

[10] Xiuzhen Huang, Jing Lai, and Steven F Jennings. 2006. Maximum common subgraph: some upper bound and lower bound results. *BMC bioinformatics* 7, 4 (2006), 1–9.

[11] Chu Min Li and Zhe Quan. 2010. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem.. In *AAAI*, Vol. 10. 128–133.

[12] Carla Savage. 1982. Depth-first search and the vertex cover problem. *Information processing letters* 14, 5 (1982), 233–235.

[13] Carsten Witt. 2012. Analysis of an iterated local search algorithm for vertex cover in sparse random graphs. *Theoretical Computer Science* 425 (2012), 117–125.