Implementing Exact, Approximate and Local Search Algorithms to Solve Minimum Vertex Cover Problem

Arjun Chintapalli
College of Computing
Department of Computational Science and Engineering
Georgia Institute of Technology
arjun.ch@gatech.edu

Xiangyi Yan
College of Computing
Department of Computational Science and Engineering
Georgia Institute of Technology
xyan79@gatech.edu

Sangy Hanumasagar
College of Computing
Civil and Environmental Engineering
Georgia Institute of Technology
sangyh@gatech.edu

ABSTRACT

The decision version of vertex cover is a classical NP-Complete problem extensively studied in computational complexity theory. This paper evaluates the performance of four distinct algorithms in solving the Minimum Vertex Cover problem: Branch and Bound method algorithm, an approximate heuristic algorithm (Greedy Independent Cover) and two local search algorithms (Hill-Climbing and Simulated Annealing).The performance of the four algorithms are evaluated for accuracy and efficiency on a set of graphs. The Local Search algorithm showed very promising results for all the datasets, while the Branch and Bound algorithm was less accurate owing to the running time constraint of 600 seconds and its systematic traversal of search space. The approximate algorithm showed the highest relative error but also very quick, as expected from its theoretical understanding.

KEYWORDS

Vertex Cover, Exact Solution, Approximate Solution, Local Search, Branch and Bound, Greedy Independent Cover, Hill Climbing, Simulated Annealing

1 INTRODUCTION

The Minimum Vertex Cover(MVC) is an extensively studied problem owing to its numerous applications in resource management, computational sciences and operational research. The problem seeks the smallest set of vertices such that every edge in the graph has at-least one endpoint in it, thereby forming the MVC solution. In this paper, four distinct approaches, including an exact solution implemented using the Branch and Bound (BnB) technique, an approximation algorithm using Greedy Independent Cover and two local search algorithms including Hill-Climbing and Simulated Annealing are considered. Their performance is tested on real and random datasets extracted from the 10th DIMACS challenge, and evaluated for accuracy and computational efficiency. As expected from theoretical understanding of the algorithms, we saw that the BnB algorithm was slow in running time while being fairly accurate while the approximation algorithm was quick but with a higher relative error. Interestingly, the Local Search algorithms, which were cleverly designed to accept the heuristic solution as a initial starting point,showed a very good performance within the default cutoff running time of 600 seconds, also finding the optimum solution in most cases.

2 PROBLEM DEFINITION

Consider an undirected graph G=(V,E), where V represents the set of vertices and E represents the set of edges. The vertex cover of G is defined as the subset of vertices, $S \subseteq V$, such that $\forall$ (u,v) $\in$ E: u $\in$ S $\vee$ v $\in$ S. In other words, every edge (u,v) $\in$ E must have at least one end point in S. The MVC problem, which seeks to minimize $|S|$, is a well-known NP-complete problem was included in the list of 21 NP-complete problems in [6].

3 RELATED STUDIES

The Vertex Cover problem belongs to a class of NP-complete graph problems with numerous applications in real life, leading it to be extensively studied over the years. MVC is also closely related to other graph problems like the Independent Set problem and Edge Cover problems. While exact algorithms can perform well within a reasonable time for small graphs, employing them is impractical for large datasets. For this reason, alternate incomplete algorithms that don't entirely search the problem space are used to find a reasonably good solution in a practical amount of time.

The Branch and Bound (BnB) technique systematically enumerated candidate solutions, while employing justifiable bounds to the optimum solution to discard subsets of candidate solutions. While the BnB does produce an exact solution given enough time, it is still exponential in time complexity (O(2^n)). [2] used the BnB technique for solving SAT problems while other studies have been undertaken to create efficient backtracking approaches. [3] summarized the potential improvements that could be achieved by using sequential BnB and parallel BnB algorithms.

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Description. The idea of the BnB algorithm is to investigate smaller subsets of the parent graph, G, for potential solutions (vertex covers), while discarding branches that do not satisfy the bounds to the optimal solution. The bounds to the optimum can be defined in multiple ways, but must be provable to encompass the optimal solution, and are continually updated to tighten the list of possible solutions.
In the current implementation of the BnB algorithm, the upper bound solution is the best solution (vertex cover of least size) encountered at any given stage during exploration, while the lower bound solution for the subset graph is defined as follows:

4.1.2 Algorithm. Given a parent graph G=(V,E), and a partial vertex cover C' to the explored section of G, $LB(subproblem) = |C'| + LB(G')$, where, G' is the unexplored graph and the lower bound to any G' is defined as $LB(G') = \frac{number of edges in G'}{maximum node degree in G'}$. The rationale behind this choice of lower bound solution is that for a graph with $|E|$ edges, maximum node degree k, and a vertex cover of size $v = |VC|$,
$v \times d \geq |E| => v \geq |E|/d, \therefore |E|/d$ forms a lower bound solution to vertex cover.
The algorithm preferentially considers vertices in decreasing order of vertex degree (number of edges connected to it) as the most promising candidates to finding the optimum solution. Each candidate vertex is assigned a state of 1 or 0 to indicate whether it is included in the VC candidate solution or not respectively, and appended to the Frontier Set.

4.1.3 Time and Space Complexity. The Frontier Set contains all candidate vertices to be explored for either possibility of state variable, and therefore, the total number of possible scenarios is $2^{|V|}$. The time complexity of the algorithm is O($2^{|V|}$) which is the number of iterations of the while loop, which each iteration takes O(V+E) time to explore a certain branch of the search space. The space complexity would be O(V+E) as 2V space would be needed to store all candidate vertices in the Frontier Set and E space to find vertex cover solution.
The BnB algorithm is presented in Algorithm 1, while the detailed work-flow description for the BnB implementation can be found in the readme file or the code file.

4.2 Heuristic Algorithm: Greedy Independent Cover

4.2.1 Description.
For the required heuristic algorithms, a Greedy Independent Cover was implemented. This algorithm was implemented based on the description found in [4].

The worst case approximation ratio of this algorithm is is at least $\frac{\sqrt[2]{MaximumDegree}}{2}$.

4.2.2 Algorithm.
This heuristic is very simple in that the vertices of input graph G are iterated through in order of ascending degree and are appended to the graph and the neighbours of the appended node deleted. During the iterations, set C is stored if it is a vertex cover.

4.2.3 Time Complexity and Space Complexity.
Each iteration of the while loop has a the worst case runtime per iteration of $O(maxdegree)$. This computational cost is required to check if after a node is appended, then will any of the connected edges become uncovered after the removal. Since, this method is an iterative method, the time complexity of the overall algorithm isn't defined.

Given a graph G(V,E) where V and E refer to its nodes and edges respectively, the space complexity is O($|V| + |E|$).

Algorithm 1: Branch-and-Bound (G)

```
Input  : Graph G of (V, E), Cutoff Time
Output : Vertex Cover of G
$VC \leftarrow \emptyset$ //will contain tuples of (vertex,state),state=1 if in VC, 0 otherwise
$v \leftarrow$ vertex with highest degree
$FrontierSet = [(v, 0, (-1, -1)), (v, 1, (-1, -1))]$ //contains subproblems, each being tuple of (vertex,state,(parent vertex,parent vertexstate))
//state of a vertex represents whether it is in VC; 1 if yes, 0 otherwise
$UpperBound = |V|$ //initial upper bound=number of nodes in G
while $FrontierSet \neq \emptyset$ do
    (vi,state,parent)=FrontierSet.pop()
    $VC \leftarrow (vi, state)$
    $backtrack = false$
    if $state == 0$ then
        Set state=1 for all neighbors of vi
        $VC \leftarrow$ All neighbors of vi
        $G \leftarrow G\backslash$ (all neighbors of vi)
    else if $state == 1$ then
        $G \Leftarrow G \setminus vi$
    if $G.edges == \emptyset$ then
        //end of exploring
        if $|VC| < UpperBound$ then
            $opt \leftarrow VC$
            $UpperBound \leftarrow |VC|$
        $backtrack = true$
        Move to next subproblem in FrontierSet
    else
        if $LowerBound(G) + |VC| < UpperBound$ then
            //worth exploring
            $vj \leftarrow$ vertex with highest degree among unexplored nodes
            $FrontierSet \leftarrow FrontierSet \cup [(vj, 0, (vi, state)), (vj, 1, (vi, state))]$
        else
            //not worth exploring
            $backtrack = true$
            Move to next subproblem in FrontierSet
        end
    if $backtrack == true$ and $FrontierSet \neq \emptyset$ then
        //backtracking step
        $nextparent \Leftarrow$ last element in FrontierSet
        if $nextparent \in VC$ then
            //remove current node from VC and add back to G
            $index \leftarrow$index of nextparent in VC
            for $i$=index to $|VC|$ do
                remove VC(i) from VC
                add VC(i) and its edges back to G
            end
        else
            reset VC $\leftarrow$ empty
            reset G $\leftarrow$ original G
end
return opt
```

Algorithm 2: Greedy Independent Cover (G)

```
Input  : Graph G of (Vertices, Edges)
Output : Vertex Cover of G
1  $C \Leftarrow \emptyset$
2  while $E \neq 0$ do
3      if $C$ is Vertex Cover of $G$ then
4          $C* \leftarrow C$
5      end
6      $u$ =Minimum Degree Vertex of G
7      $C \Leftarrow C \cup u$
8      $G(V,E) \Leftarrow G(V,E) - n(u)$, where n(u) are neighbors of u
9  end
10 return C*
```

4.3 Local Search 1: Hill Climbing

4.3.1 Description.
For one of the two required local search algorithms, the Hill Climbing Search was implemented. Specifically, the NumVC algorithm was implemented based on the description found at [1].

4.3.2 Algorithm.
The pseudocode for this algorithm is presented in this section. This algorithm was implemented such that to take in the Heuristic Greedy Independent cover algorithm as an initial solution, and then keeps swapping points to change the solution state to explore the neighboring search space till the solution is a vertex cover. The neighbor solution that minimizes the cost function, in this case the number of uncovered edges, is iteratively moved to.

A cost function was used to determine the node to remove such that the node to be removed results in minimal increase in the number of uncovered edges. Then, an endpoint of a random uncovered edge, not already in set C, is added to set C. This swapping drives the search for a smaller vertex set. This swapping is implemented with the addition of taboo-like prohibition implementation, where a configuration array is checked before a new point is added to the solution.

Furthermore, the implementation of the cost function and taboo states help reduce the frequency of this algorithm from getting stuck at local maxima. Whereas the taboo state implementation prevents re-searching the same search space, the cost function implementation guides the algorithm to make the most optimal swaps. But since this algorithm is still a local search algorithm, there is no guarantee that this algorithm will find an optimal solution and might only find a locally optimal solution. In contrast Branch and Bound gives the guarantee of finding a globally optimal solution given enough time.

If a solution is a vertex cover then this solution is saved, and a node is deleted such that this NP optimization problem is recast as a decision problem of finding a vertex cover of $k-1$ nodes, where k is the current number of nodes in the vertex cover. Resultantly, since this decision problem is NP- Hard, this gives further proof that the Minimum Vertex Cover optimization problem is also NP-Hard.

This algorithm was implemented and found to be working accurately, and very fast. After many days of debugging, an implementation that doesn't iterate through the edges of the graph with each iteration dramatically improved performance!This implementation was achieved by simply keeping track of how node swaps affect the number and identity of uncovered edges, cost functions and taboo state. This algorithm was implemented such that the worst case runtime per iteration of the while loop is $O(maxdegree)$, and so the number of swap searches possible within cutoff was dramatically improved so that the overall accuracy of this algorithm matches the Branch and Bound.

Algorithm 3: Local Search: Hill Climbing (G)

```
Input  : Graph G of (Vertices, Edges)
Output : Vertex Cover of G
1  C $\Leftarrow$ $InitialVertexCover$
2  while $(CurrentTime - StartTime) < Cutoff$ do
3      while $C$ is Vertex Cover of $G$ do
4          Store Solution: $C* \leftarrow C$
5          Remove vertex from C leading to Highest Cost Loss
6          Where $Cost(G,C) = \sum w(e)$ for each uncovered $e \in E$
7      end
8      $u$ = vertex from C leading to Highest Cost Loss
9      $C \Leftarrow C \setminus u$
10     $Taboo(u) = 0$
11     $Taboo(x) = 1$ for x in Neighbors(u)
12     $v \Leftarrow$ endpoint of random uncovered edge: S.T $C \setminus v$ and
         $Taboo(v) == 1$
13     $C \Leftarrow C \cup v$
14     //Update edge weights:
15     $w(e) += 1$ for each uncovered $e \in E$
16 end
17 return C*
```

4.3.3 Time Complexity and Space Complexity.
As priorly discussed each iteration of the while loop has a the worst case runtime per iteration of $O(maxdegree)$. Since, this method is an iterative method, the time complexity of the overall algorithm isn't defined.

Given a graph G(V,E) where V and E refer to its nodes and edges respectively, the space complexity is O($|V| + |E|$).

4.4 Local Search: Simulated Annealing

4.4.1 Description.
Simulated Annealing is selected to implement one of the local search algorithm. This approach is based on the process of physical annealing of which the temperature decreases as time passes by. Analogously, after getting a fixed count of iterations, probability of accepting worsening case decreases. [5]. In the above algorithm, I used Greedy Independent Cover algorithm introduced in Algorithm2 to generate initial solution as an input for Simulated

Annealing. In my implementation, the way of searching for a neighbor is randomly removing a vertex that is in the current solution, and randomly take in a vertex that is not covered by vertex cover. The pricing function determining the quality of a given candidate solution is designed as the number of edges that are not covered by the candidate solution. The pseudocode of this approach is shown as Algorithm 4.

Algorithm 4: Local Search(SA)

```
Input  : Graph G of (V, E), initial solution
Output : final solution
T = 0.15
updated solution ← ∅
sol ← initial solution
while execution time < cutoff do
    T = 0.95 * T
    while searching steps not exceeds fixed steps do
        if sol is a vertex cover then
            updated solution ← sol
            randomly remove one vertex from sol
        end
        current solution ← sol
        randomly delete a vertex from sol
        randomly add a vertex that is not in sol
        if current solution is better than sol then
            probabilistically accept sol as current solution
        end
    end
end
return updated solution
```

4.4.2 Algorithm Analysis.
I select simulated annealing algorithm because it is widely used and is also easy to implement.

Parameters and mechanism.
I choose initial temperature as 0.15 and the temperature decreasing rate as 0.95 which are introduced in Holger's book [5]. The fixed number of iterations after which the temperature goes down is $n = (m - l - 1)^2$, where m is the number of nodes in the graph and l is the number of nodes in current solution. The cutoff time is set to be 1000s, which means the iteration breaks at 1000s if it hadn't found the given optimal solution yet. There is an automated tunning mechanism that as the iteration processes, the probability of accepting worse solution decreases. This helps us accepting more possibilities at the beginning of calculations and gradually shrink the scope to reject worse solutions.

Time Complexity and Space Complexity.
Time complexity analyzed here refers to that inside the while loop. Given n as the maximum degree of the graph, time complexity is O(n), since we only scan for all the neighbor of a node within each iteration. Given a graph G(V,E) where V and E refer to its nodes and edges respectively, the space complexity is $O(|V| + |E|)$ since we only store the information of this graph.

Advantage and disadvantage.
The advantage of simulate annealing is that it's a simple and clear algorithm while it can give out relatively good solutions. However, the disadvantages are: the optimal solution cannot be guaranteed and for instances that have a small input size, this algorithm is overly complex.

5 EMPIRICAL EVALUATION

The algorithms were implemented in Python 3.6 taking advantage of the Networkx package priorly used for this class. To ensure accuracy and validity of the results the Python scripts for all the algorithms were tested on the Georgia Tech HP Z230 workstations, which have a 3.6 Ghz Intel 7 processor, 8 GB RAM and 1 TB SSD Storage.

For the final results the algorithms were run on all 11 datasets and if there was a random component in the algorithm, such as the two local search algorithms, then those algorithms were run 10 different random seeds and the performance was averaged of the 10 random seed performance. The resulting performance was evaluated based on time to find the optimal solution as well as the best candidate solution found within a given cut-off. The running times and associated sizes of minimum vertex covers found for all the graph instances are presented in Table 1.

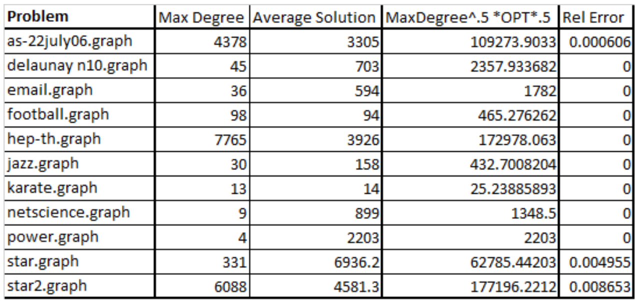
5.1 BnB Results

The BnB algorithm was expected to take a long time to complete traversal of the entire search space (with exponential candidate solutions), and hence run up to an hour each. However, owing to the initial starting point defined such that it begins traversal with the maximum degree node, which represents the most promising vertex to be in vertex cover, the first solution was found very quickly in less than 600 seconds but the second solution took an inordinate amount of time in order of hours. The results presented in Table 1 are all obtained within 600 seconds. The BnB algorithm finds optimal solution for four out of the eleven graphs while the maximum relative error encountered was 0.067 for star.graph, which is acceptable given the time constraint of 600 seconds.

5.2 Approximation Results

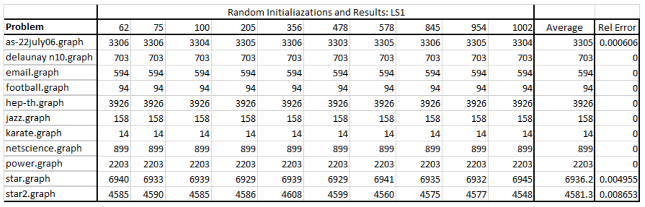
Good results were achieved by the GIC Heuristic Algorithm, as shown in the over all results table in the results section. The worst case relative error from the optimal was only 7% in the star2 graph, which had a relatively high maximum degree of 6088 edges indicating that the worst case approximation is correct. This algorithm was implemented such that the worst case runtime per iteration of the while loop is $O(maxdegree)$, to check if the candidate is a vertex cover, and so this algorithm is extremely fast, taking only at maximum 6 seconds.

As mentioned before the worst case approximation ratio of the implemented Greedy Independent Cover algorithm is is at least $\frac{\sqrt[2]{MaximumDegree}}{2}$. The results in this section demonstrate that the average of randomized runs of this algorithm are within the



Problem	Max Degree	Average Solution	MaxDegree^.5 *OPT*.5	Rel Error
as-22july06.graph	4378	3305	109273.9033	0.000606
delaunay n10.graph	45	703	2357.933682	0
email.graph	36	594	1782	0
football.graph	98	94	465.276262	0
hep-th.graph	7765	3926	172978.063	0
jazz.graph	30	158	432.7008204	0
karate.graph	13	14	25.23885893	0
netscience.graph	9	899	1348.5	0
power.graph	4	2203	2203	0
star.graph	331	6936.2	62785.44203	0.004955
star2.graph	6088	4581.3	177196.2212	0.008653

Figure 1: Approximation Algorithm Results



	Random Initialiazations and Results: LS1											
Problem	62	75	100	205	356	478	578	845	954	1002	Average	Rel Error
as-22july06.graph	3306	3306	3304	3305	3306	3303	3305	3306	3305	3304	3305	0.000606
delaunay n10.graph	703	703	703	703	703	703	703	703	703	703	703	0
email.graph	594	594	594	594	594	594	594	594	594	594	594	0
football.graph	94	94	94	94	94	94	94	94	94	94	94	0
hep-th.graph	3926	3926	3926	3926	3926	3926	3926	3926	3926	3926	3926	0
jazz.graph	158	158	158	158	158	158	158	158	158	158	158	0
karate.graph	14	14	14	14	14	14	14	14	14	14	14	0
netscience.graph	899	899	899	899	899	899	899	899	899	899	899	0
power.graph	2203	2203	2203	2203	2203	2203	2203	2203	2203	2203	2203	0
star.graph	6940	6933	6939	6929	6939	6929	6941	6935	6932	6945	6936.2	0.004955
star2.graph	4585	4590	4585	4586	4608	4599	4560	4575	4577	4548	4581.3	0.008653

Figure 2: LS1: Hill Climb Trials

worst case approximation ratio and are in fact even within $2OPT$ and surprisingly close to the optimal solution.

5.3 LS1: Hill Climbing Results

Extremely good results were achieved by the LS1 Algorithm. As can be seen this algorithm found the optimal number of vertices or was very close for all the graph sets, ranking in the top 3 in the class in terms of accuracy. The average of the results of ten runs with different seeds with a time cutoff of 10 minutes was used for comparison against the non-random methods.

The graphs of Qualified Runtime Distributions for various solution qualities(QRTDs) and Solution Quality Distributions(SQDs) for the Hill Climbing Local Search are shown in Figure 3 and Figure 4.

The QRTD plots are used to show the probability that an algorithm will solve a solution as time progresses for various qualities relative to the optimal. These relative qualities indicate that the candidate solution is at most this percentage away from the optimal. The QRTD plots indicate that within 3 seconds of running the LS1 algorithm will have a 100% sample probability of finding solutions to at most 2% of the optimal solution for the power graph. Likewise for the Star2 graph, it only takes ~ 20 seconds to have a 100% sample probability of finding solutions to at most 2% of the optimal solution. Having runs with more random seeds will better confirm and add more granularity to this algorithmic analysis. Additionally for the Star2 graph, there seems to be an outlier random run that took significantly longer (~ 40 seconds) than the other runs to find the optimal solution. This demonstrates the inherent instability of local searches because they can get stuck at local optima.

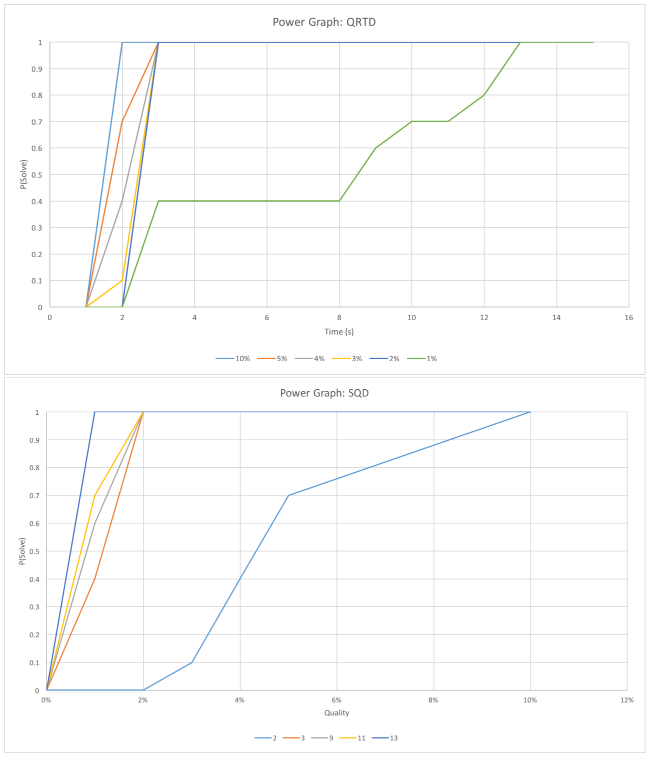


Figure 3: LS1: Power Graph Results

The SQD plots show the probability of achieving a solution as a function of relative quality for various times. The SQDs are plotted the same way with qualities as its x-axis and ratios for solving the problem as y-axis. For example the Power Graph demonstrates that even with a run of 2 seconds, there is a 100% sample probability of finding a solution at most 10% away from the optimal.

5.4 LS2: Simulated Annealing Results

The graphs of Qualified Runtime Distributions for various solution qualities(QRTDs) and Solution Quality Distributions(SQDs) for Simulate Annealing(which refers to LS2)are shown as Figure 7 and Figure 8.
For QRTD, the x-axis represents running times in seconds and the y-axis represents the ratio of algorithm runs that solves the problem to some extent, which is specified as the quality. Running times here count from start of the local search algorithm with an initial solution calculated by heuristic implemented in this project. I selected qualities at 2.0%, 1.0%, 0.5%, 0.3%, 0.2%, 0.1% to plot the result of power.graph and 6.5%, 6.0%, 5.0%, 3.0%, 2.0%, 1.0% to plot the result of star2.graph. We can see that the code performs really well that it takes short time to give a high quality solution at high probability, especially for those graphs with smaller size.
Similarly, SQDs are plotted the same way with qualities as its x-axis and ratios for solving the problem as y-axis. We can see from the graphs that even given a high quality, it has high possibility

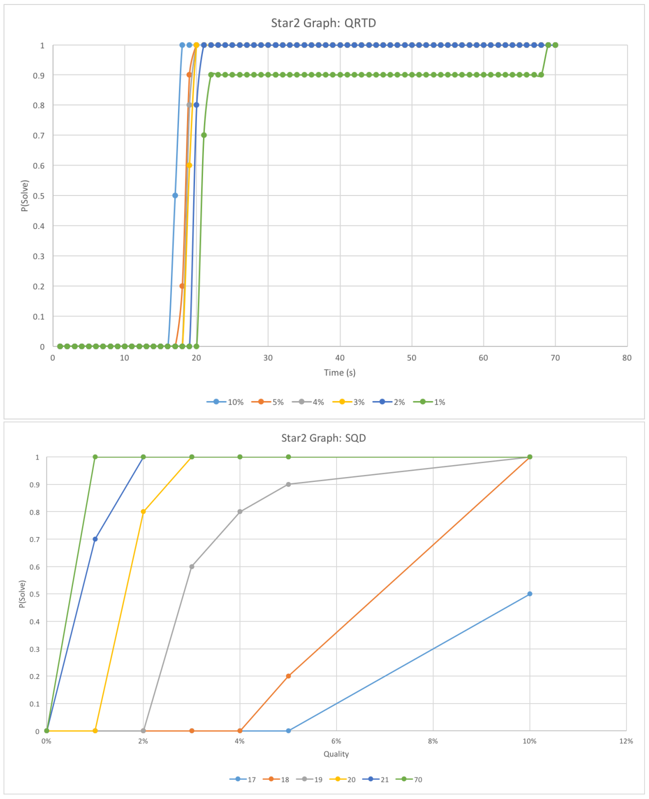


Figure 4: LS1: Star2 Graph Results

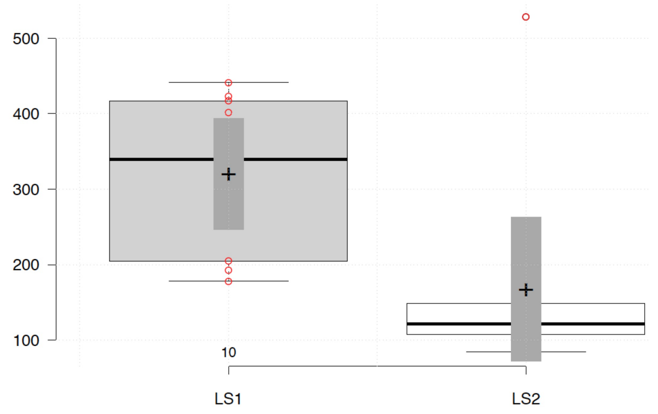
reaching the expected solution while the time spent is not too long.

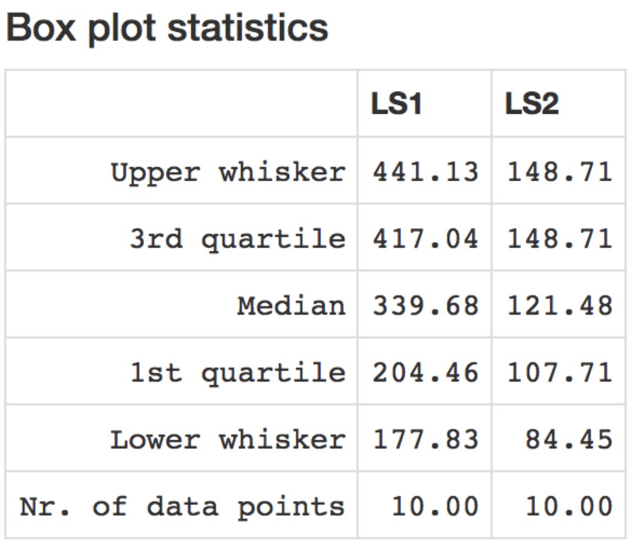
5.5 Comparison between Hill Climbing and Simulated Annealing

The QRTD and SQD plots are very useful for comparison of 2 random algorithms. For example, the QRTD plots on the star2 graphs indicate that for a given relative quality, say 2%, that the Hill Climbing algorithm achieves this performance much faster (20 seconds as opposed to 145 seconds). The SQD plots further demonstrates this in that for a 70 second runtime LS1 has a 100% sample probability of 1% quality solution but LS2 can only give a 60% probability for a lower quality 3% quality solution.

Additionally, the QRTDs indicate that both LS1 and LS2 had a random initialization that wasn't converging to the optimal solution for star2 because they both had quality lines that plateaued below 1. This could possibly be a feature of the graph itself in that there could be a region of the search space in the graph that has a solution especially close to the optimal (a local optima).

The box plots for power.graph and star2.graph are generated as Figure 5 and Figure 6 using running times of LS1, which is Hill Climbing and LS2, which is Simulated Annealing. We can see from the box plot for power.graph that the average value of running




Box plot statistics

	LS1	LS2
Upper whisker	441.13	148.71
3rd quartile	417.04	148.71
Median	339.68	121.48
1st quartile	204.46	107.71
Lower whisker	177.83	84.45
Nr. of data points	10.00	10.00

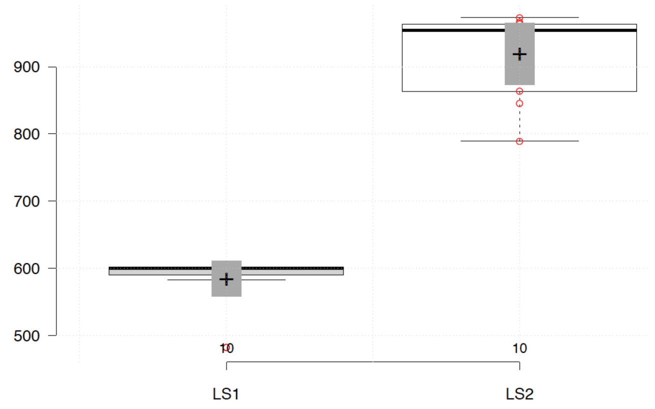
Figure 5: Box Plot for power.graph

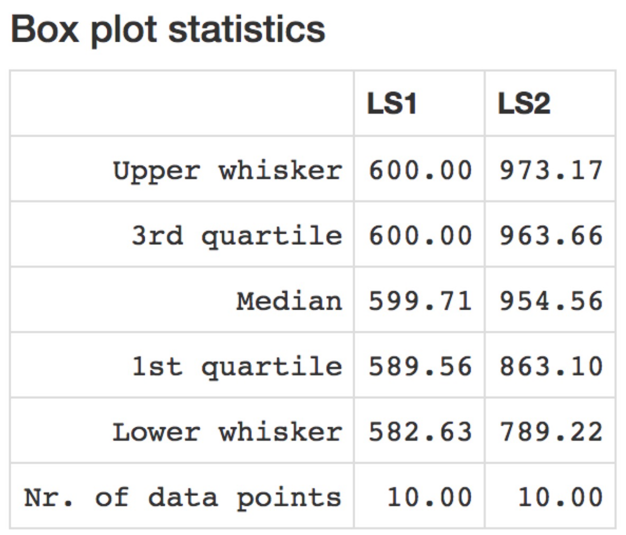
time of LS1 is longer and has larger variation than that of LS2, which means that LS2 is not only faster but also more stable than LS1 when applied to power.graph. Indeed at all percentiles in the time distribution, the LS2 algorithm has smaller run times. This demonstrates effectiveness of box plot statistics to determine the optimal algorithm to use given a graph ahead of time.

As for star2.graph, which has the larger size, since the two algorithms do not find optimal solution at last and LS2 is set longer time cutoff(1000s) than LS1(600s), running time of LS2 is longer than that of LS1.

6 DISCUSSION

The performance of the two Local Search algorithms stood out in terms of accuracy as well as speed, relative to the BnB and Approximation methods. The BnB systematically evaluates a large number of candidate solutions, the size of the which is determined by the tightness of the bounds, but is theoretically the slowest of the four techniques, which was seen in the performances. Given enough time, the BnB should arrive at the exact solution, however a





Box plot statistics

	LS1	LS2
Upper whisker	600.00	973.17
3rd quartile	600.00	963.66
Median	599.71	954.56
1st quartile	589.56	863.10
Lower whisker	582.63	789.22
Nr. of data points	10.00	10.00

Figure 6: Box Plot for star2.graph

cutoff-time of at most 1 hour was used for this project, which is reflected in the results obtained. The approximation method showed the worst accuracy of all the algorithms, but the maximum relative error observed was 0.069, which could be a good starting point for further analysis at a low time cost. The Local Search algorithms were executed using the approximate solution as a starting point, and quickly arrived at a near-optimum or optimum solution as seen in Table 1. Figure 8 presents a bar chart depicting the relative errors for the various graph instances and algorithms.

The BnB algorithm could potentially have been improved if it used the approximate solution as the initial lower bound before exploring the search space.The current implementation estimates the lower bound as the ratio of number remaining edges in the subgraph to maximum node degree, which, although valid, could be a looser bound. However, considering the time bound of 1 hour (5 hours in some cases), the algorithm showed a good performance with finding the optimum solution for 4 graph instances. Other advanced techniques that improve the backtracking or parallel BnB can definitely improve the speed of evaluating the candidates.

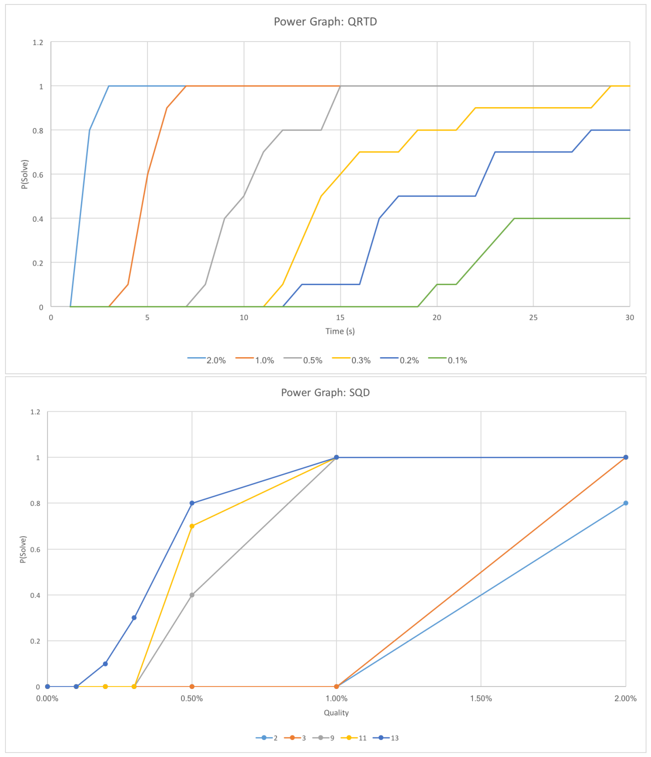


Figure 7: LS2: Power Graph Results

In real world settings, the choice of which algorithm to implement is based on the trade off between accuracy and speed. If time is the primary evaluation criteria the approximation algorithm is the optimal choice because it runs orders of magnitude faster than the other algorithms, otherwise if accuracy is the primary criteria than Branch and Bound is the best choice because it has a guarantee of absolute accuracy and if a balance between these extremes is desired than either of local searches should be chosen.

The time complexity of the algorithms depend dramatically on the actual implementation. In the original implementations, both the local searches and the approximation algorithms ran in $O(E)$ time per iteration due to repeated searches to find uncovered edges but with optimizations that kept track of uncovered edges through the iterations these algorithms then ran in $O(maximumdegree)$ time per iteration. The Branch and Bound algorithm was implemented in O(E+V) time per iteration.

7 CONCLUSION

A summary of various algorithms to solve the vertex cover problem have been presented in this paper. In general, the choice of algorithm to employ to solve the MVC problem should be made based on the resources available and accuracy required. In theory, for an application requiring a high degree of accuracy with no contraint on the running time and computing power, the Branch and Bound method would be the best choice. On the other hand, if

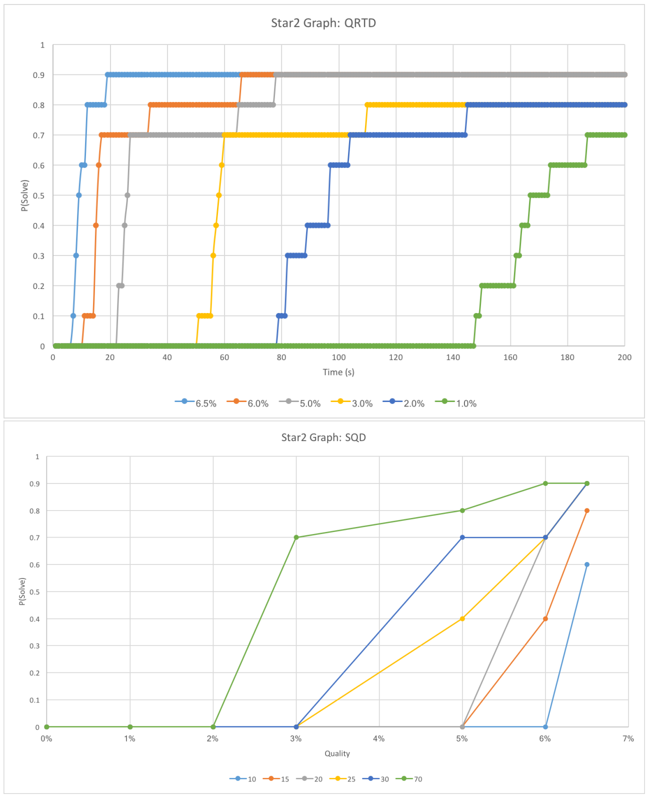


Figure 8: LS2: Star2 Graph Results

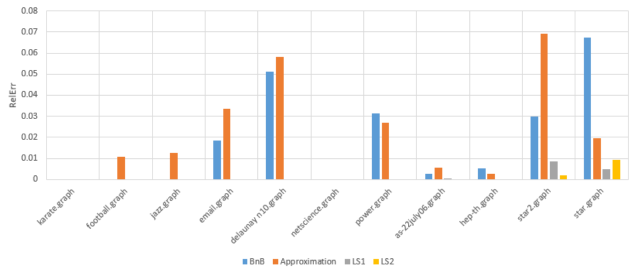


Figure 9: Relative Error Comparison

accuracy can be sacrificed and quick approximations are required, then heurestic approach is the best choice. Finally, the local search algorithms provide a balance between accuracy and speed.

Due to the property of NP-Complete problems, every other NP-Complete problem can be reduced to the Vertex Cover problem in polynomial time. Therefore, the algorithms implemented in this paper enables a programmer to reduce other NP-Complete problems to the Vertex Cover problem and apply these solutions. Finally, the strategies presented here can also be utilized to solve other NP-Complete problems.

Table 1: Summary of Algorithm Performance

	BnB	Approx	LS1	LS2
karate.graph				
Time(s)	0.00001	0.00001	0.00001	0.00
VC Value	14	14	14	14
RelErr	0.00	0.00	0.00	0.00
football.graph				
Time(s)	77.18	0.001	0.004	0.024
VC Value	94	95	94	94
RelErr	0.00	0.011	0.00	0.00
jazz.graph				
Time(s)	308.97	0.0065	0.028	0.076
VC Value	158	160	158	158
RelErr	0.00	0.013	0.00	0.00
email.graph				
Time(s)	0.73	0.047	0.75	10.49
VC Value	605	614	594	594
RelErr	0.019	0.034	0.00	0.00
delaunay_n10.graph				
Time(s)	3.18	0.04	2.79	356.047
VC Value	739	744	703	703
RelErr	0.051	0.059	0.00	0.00
netscience.graph				
Time(s)	1.23	0.05	0.523	0.00
VC Value	899	899	899	899
RelErr	0.00	0.00	0.00	0.00
power.graph				
Time(s)	11.6	0.05	406.7	167.65
VC Value	2272	2262	2203	2203
RelErr	0.031	0.027	0.00	0.00
as-22july06.graph				
Time(s)	94.97	4.26	600.00	73.77
VC Value	3312	3321	3305	3303
RelErr	0.0027	0.0054	0.00061	0.00
hep-th.graph				
Time(s)	30.31	1.14	261.6	132.65
VC Value	3947	3937	3926	3926
RelErr	0.0053	0.0028	0.00	0.00
star2.graph				
Time(s)	3785.94	12.18	600.0	852.79
VC Value	4677	4856	4581	4550
RelErr	0.030	0.069	0.0087	0.0018
star.graph				
Time(s)	78.47	5.92	600.0	934.34
VC Value	7366	7038	6936	6967
RelErr	0.067	0.020	0.005	0.0094

REFERENCES

[1] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. 2013. NuMVC: An efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research* 46 (2013), 687–716.

[2] Zhe Quan Chu-Min Li. 2010. An Efficient Branch-and-Bound Algorithm Based on MaxSAT for the Maximum Clique Problem. *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)* 10 (2010), 128–133.

[3] Jens Clausen. 1999. Branch and Bound Algorithms – Principles And Examples. (1999).

[4] François Delbot and Christian Laforest. 2010. Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover Problem. *J. Exp. Algorithmics* 15, Article 1.4 (Nov. 2010), 1.17 pages. DOI:http://dx.doi.org/10.1145/1865970.1865971

[5] Thomas Stutzle Holger H. Hoos. 2004. *Stochastic Local Search: Foundations and Applications.* Elsevier.

[6] Richard M. Karp. 1972. Reducability Among Combinatorial Problems. *Complexity of Computer Computations. The IBM Research Symposia Series. Springer* (1972). DOI:http://dx.doi.org/https://doi.org/10.1007/978-1-4684-2001-2_9