Approximation Algorithms: Minimum Vertex Cover

Li, Ryan
Georgia Institute of Technology
Atlanta, USA

Morioka, Hatsune
Georgia Institute of Technology
Atlanta, USA

Shin, Bomm
Georgia Institute of Technology
Atlanta, USA

Wang, Gabriel
Georgia Institute of Technology
Atlanta, USA

ACM Reference Format:
Li, Ryan, Morioka, Hatsune, Shin, Bomm, and Wang, Gabriel. 2021. Approximation Algorithms: Minimum Vertex Cover. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In this paper, we discuss and implement different algorithms for building a Minimum Vertex Cover (MVC) to evaluate the accuracy and speed of each implementation. The MVC problem is defined by finding the minimum size set of vertices so that for all the edges in the graph at least one vertex per edge is within the output set of vertices. The implementations we chose were Branch and Bound, Approximation, Genetic Algorithms, and Stochastic Local Search. For the Approximation algorithm, it found a vertex cover in a short amount of time, but the effect was an inaccuracy of up to 2 times the amount of vertices chosen as the optimal solution. For the Branch and Bound algorithm, we formulated a binary state-space tree to search for an exact solution with upperbound and lowerbound for pruning purposes. While accuracy is guaranteed, the running time is exponential to the input size. Hence, a time limit of 600 seconds were imposed on the algorithm to run on each input. For the Stochastic Local Search, it found a relatively accurate result, however, it takes a large amount of time until it reaches to the local minima for dense graphs. Setting a cut-off time to reduce running time can lead to a considerable drop in accuracy in this algorithm, so it is hard to predict a reasonable cut-off time. Finally, Genetic Algorithms found relatively accurate results for smaller graphs, but tended to get caught in local minimas for larger graphs once a valid vertex cover was found. While it could probably find better solutions, it would take an infeasible amount of time.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 Related Studies

The Vertex Cover Problem is an NP-Complete graph problem which is similar to problems like the Independent Set Problem and Edge Cover Problem. And the Vertex Cover Problems can be re-formulated as a half-integral linear programs such as the Maximum Matching Problems.

The Vertex Cover Optimization Problem serves as a model for many real-life problems. For example, an establishment that wants to find the fewest possible camera installment covering the hallways (edges) connecting all rooms (nodes) on a building of a floor might be the objective of the Minimum Vertex Cover Problem.

3 Branch and Bound

Let C' be a partial vertex cover of G and $G' = (V', E') \subseteq G$, subgraph that is not covered by C'. Also, let $V' = V - C' - \{v \in V | \forall (v,u) \in E, u \in C'\}$ and $E' = \{(u,v) \in E; u, v \in V'\}$. We find the MVC using Branch and Bound by exploring nodes in decreasing order of degrees, since max degree node is the most likely to belong to a Vertex Cover. Each node is expanded in binary choice of either belonging to Vertex Cover or not. Then each choice is evaluated with the LowerBound= $|C'| + \frac{\text{number of edges in partial graph } (G')}{\text{max node degree in } G'}$. The reason behind this choice is that suppose a partial graph has total of m edges, and max node degree, k, then the vertex cover for that subgraph should be at most $\frac{m}{k}$. The initial upper bound for the root node is the total number of nodes in G and is updated with the best vertex cover found during the exploration. Each expansion is evaluated with the LowerBound and UpperBound mentioned previously to either prune or continue the expansion. The algorithm ends once all the nodes have been explored.

The Frontier set contains all the candidate nodes to be explored with either possibility of either belonging to the Vertex Cover or not. Hence the time complexity of this Algorithm is $O(2^{|V|})$. And the space complexity is $O(|V| + |E|)$.

```
CurrentGraph= InputGraph
CurrentVC= []
Frontier=[]
v_i= node with max degree(CurrentGraph)
UpperBound= Number of nodes in InputGraph
TimeList=[]
while FrontierSet ≠ ∅ and t < cut-off time do
    (v_i, state, parent) = vertex with max degree;
    if state == 1 then
        Remove v_i from CurrentGraph ;
        Append v_i to CurrentVC ;
    else if state == 0 then
        Remove v_i from CurrentGraph ;
        Add all incident nodes to CurrentVC ;
    if CurrentGraph is empty then
        if length of CurrentVC < UpperBound then
            Upperbound = length of CurrentVC;
            append (length of CurrentVC, time
              passed) to TimeList ;
        end
        if Frontier ≠ ∅ and
          parentNode ∈ CurrentVC then
            iterate CurrentVC and remove nodes;
            add the removed nodes back to
              CurrentGraph;
        else
            Reset VC as empty set;
            reset G as the original Graph;
        end
    else
        LowerBound= numberofedgesinCurrentGraph / maxdegreenodeinCurrentGraph;
        LowerBoundUpdate= int(LowerBound) +
          length of CurrentVC;
        if UpperBound > LowerBoundUpdate then
            v_j= max degree of CurrentGraph;
            Append (v_j, state=0, parent vertex);
            Append (v_j, state=1, parent vertex) ;
        else
            if Frontier ≠ ∅ and
              parentNode ∈ CurrentVC then
                iterate CurrentVC and remove nodes;
                add the removed nodes back to
                  CurrentGraph;
            else
                Reset VC as empty set;
                reset G as the original Graph;
            end
        end
    end
end
Optimal Vertex Cover, TimeList
```

Algorithm 1: Branch and Bound

3.1 BnB Pseudocode

4 Approximation

To create a MVC using a construction heuristic with approximation guarantees, the approximation algorithm runs through the graph to add vertices based on unused edges. The code has two sets to keep track of: the set of edges and vertices. The edges set is initialized as the entire set of edges in the graph which the vertices is set to an empty set. The code then runs through and randomly selects edges to add both vertices to the vertices set. We used a random seed of 100. Once those two vertices are added, all edges that contained either vertex will be removed from the edges set. Once there is no more edges left, the algorithm ends and returns the set of vertices.

This algorithm is a poly-time 2-approximate algorithm meaning that the returned set of vertices is at most twice the size of the optimal MVC.

The Time and Space Complexity are O(E*E) and O(E + V) respectively.

The strengths of this algorithm is it's simplicity. The code is small and easy to understand, but this results is a inaccurate output. The clear weakness of this algorithm is that it typically doesn't provide the optimal solution due to the fact that it adds both vertices when adding a new edge meaning that that edge has been covered twice.

4.1 Approx Pseudocode

```
Data: G = (V, E)
V' = empty set;
E'= E;
while E' is not empty do
    x, y = random element in E';
    V' add x and y;
    remove any element in E' that contains x or y;
end
return V';
```

5 Local Search (Genetic Algorithms)

To create a MVC using genetic algorithms, we have to define a few terms. First, how will an individual in a population be represented? In our case, we define an individual as a subset of vertices from the original graph. By definition, a MVC is a subset of vertices that connects to every edge in the graph. Therefore, we can use this set representation as our individual, and our population will simply be a list of sets, where each set is an individual that can be evaluated. Individuals can be initialized randomly this way very easily as well. We can simply create a random subset of vertices to act as an individual, then add it to the population. Since we now have an individual and population, next we need to

define how we are evaluating the individuals.
We can use the cost function defined in the description, where $Cost(G, C')$ = number of edges not covered by C'. We want to minimize the number of edges not covered by C', as a valid vertex cover would have a cost of 0 as it covers every edge. However, we also want to minimize the number of vertices in the cover, as the definition of an MVC is the vertex cover with the minimum number of vertices possible. Therefore, we can use multi-objective optimization here. We want to minimize both our $Cost$, as defined above, and minimize N, the number of vertices in the individual. We can consider our total "fitness" function as the simple sum of those two objectives. $Fitness(G, C') = Cost(G, C') + N$. However, we always want to produce a valid vertex cover, so we want to encourage solutions that are valid. Invalid solutions will always have a $Cost(G, C') > 0$, therefore if $Cost(G, C') > 0$ we add a high constant to its fitness to encourage finding more valid individuals.
Next we must define how we will actually search for the optimal solution. As we mentioned, we randomly initialize individuals by creating random subsets. We have implemented two ways to help the population search the space and find the best solution.

Method 1: Mating
We have implemented something similar to traditional crossover methods. In traditional genetic programming, individuals are represented as trees and we exchange subtrees between individuals. In our case, since our individuals are sets, we look for the difference between the two sets. Given the individuals A and B, we can form two lists of the differences between the sets. Let us create 2 lists, L_1 and L_2. L_1 contains all the nodes in A but not B and L_2 contains all the nodes in B but not A. Given these two lists, we randomly select half of the nodes, and exchange them between the individuals. This way, each individual produces a new "child" with a new set of vertices for the next generation.

Method 2: Mutation
Since our individuals are sets, we can randomly add, remove, or replace a node in a set to act as a mutation. For remove, we randomly remove a vertex from the set. For add, we add a random possible vertex from the set of all possible vertices. Finally, if we replace a node, we randomly remove a vertex from the set then add a random vertex from the possible vertices. So mutating an individual will change the solution in some way which we can then evaluate.

Finally, we discuss how we select individuals for the next generation. We can evaluate individuals using our two objectives stated above. So in order to select individuals for the next generation, we use a simple tournament selection method where we randomly split the population into groups of 3, then of those groups of 3, we select the individual with the best combined fitness from our objective functions. The selected individuals are then used for the next generation. We also introduce the idea of elitism and pareto dominant individuals. Unlike other search algorithms, genetic algorithms keeps a population of possible solutions, therefore we can store and persist good individuals for multiple generations. We keep the best individuals from each generation, and select them to continue to the next generation. This is the concept of elitism. However, since we have multiple objectives that we are trying to minimize, there is not a single "best" individual per generation. But rather, multiple solutions that perform better at a certain objective compared to other individuals. This produces the concept of pareto dominant individuals, where we can create a boundary between individuals who have "optimal" fitness for objectives combinations and individuals who cannot perform better than the boundary individuals at any objective. So combining the concepts of elitism and pareto dominance, at each generation, we select the pareto dominant individuals and allow them to continue to the next generation.

5.1 Genetic Algorithm Pseudocode

```
population = [] // A simple list ;
population size = 50 // Set an initial population size ;
// Initialize the population ;
while length(population) < population size do
    initializeIndividual();
end
for every generation do
    for individual in population do
        evaluateIndividual();
        mate();
        mutate();
    end
    selectIndividuals();
end
return max(individualFitness);
```

5.2 Plots

In order to plot QRTDs, SQDs, and Box plots, we used ten different results obtained with random seed = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 for each of power.graph and star2.graph. Unfortunately, most of the solutions found by genetic algorithms for larger graphs were very large vertex covers. Since the quality of the solutions produced were very poor, our plots reflect that. Unfortunately, since our solutions did not reach the minimum thresholds for quality, there were 0 quality solutions that met our criteria. Because of this, our QRTDs and SQDs are horizontal lines at 0. If the genetic algorithm was given more time, it likely would have

found a solution that met our minimum thresholds. We could have also changed our $q*$ value to better reflect our results, however, we aren't sure what would be a reasonable $q*$ in this scenario. Look at our Stochastic Local Search section for more plots that better represents a scenario with good solutions that are found.

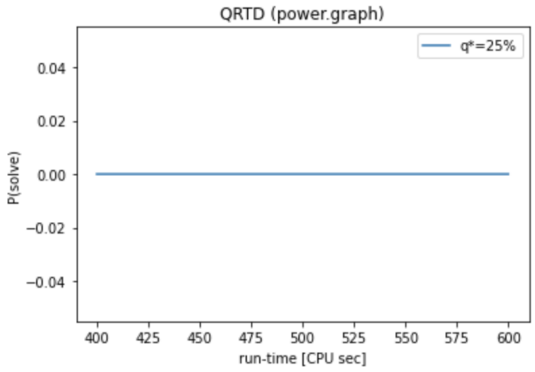

Figure 1. QRTD for power.graph

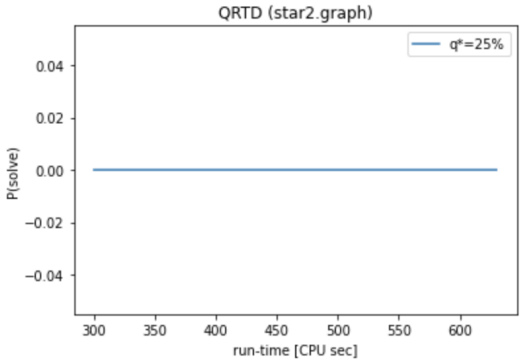

Figure 2. QRTD for star2.graph

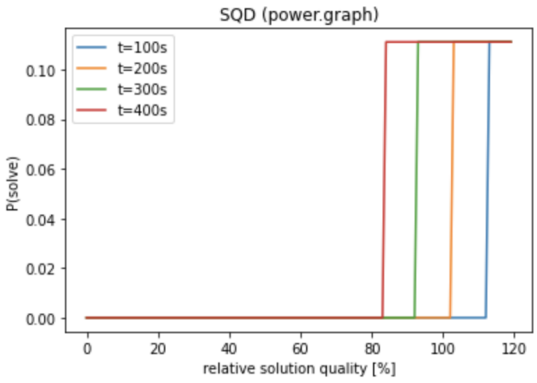

Figure 3. SQD for power.graph

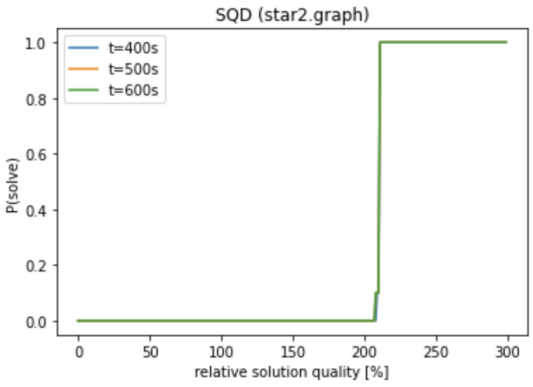

Figure 4. SQD for star2.graph

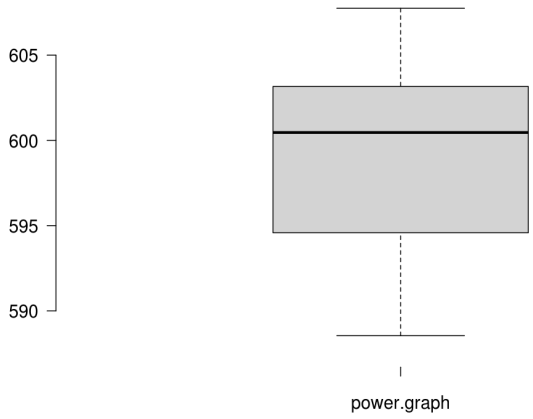

Figure 5. Box plot for power.graph (axis=run-time)

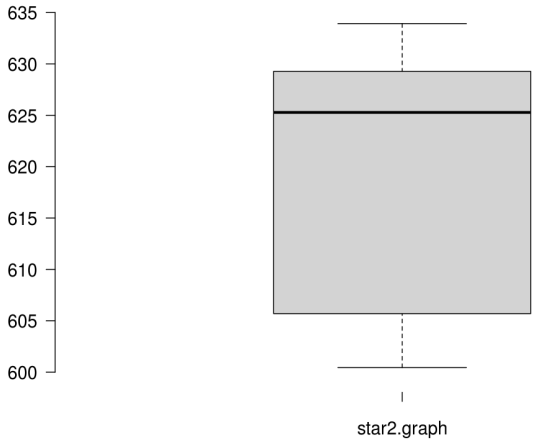

Figure 6. Box plot for star2.graph (axis=run-time)

6 Local Search (Stochastic Local Search)

6.1 Algorithm Description

Stochastic local search is an algorithm which uses a randomized initial state and randomized steps in a searching

process. In our algorithm, we used a whole set of nodes V of a given graph $G = (V, E)$ instead of a random sample of nodes as an initial state. In addition, we used an enhanced way of randomized selection to choose the next state to test rather than a completely randomized selection to make a more efficient selection at each step.

First of all, our initial state is a whole set of nodes V of a given graph $G = (V, E)$. A set of all nodes in a graph is guaranteed to be a vertex cover, so it is a good candidate solution to start with. Let C denote this candidate solution.

Second of all, we search through each node in G one by one in order of increasing number of its neighbors (degree) to determine whether it should be included in a vertex cover or not. Clearly, a node of higher degree is more likely to be in a vertex cover because it covers more edges, so it is safer to remove a node of lower degree from our candidate solution C. Taking this into account, we can achieve a more efficient method of a randomized selection of a next candidate solution to test as follows:

1. Sort a set of nodes V in a graph G in order of increasing degree. Let Q denote this sorted list of nodes.
2. Pick up a node i from Q to test whether it should be included in a vertex cover in order of increasing degree. If we decide not to include i in C, then remove i from C. If there are more than one node of the same degree, then pick up a node randomly from the set of nodes of the same degree. We test each node only once, so we will not test the same node multiple times.

Third of all, we will consider how to determine whether we should remove a node i from C or not. In our local search steps, we only remove nodes from our candidate solution without adding any nodes, so we want to make sure a current state of our algorithm is a vertex cover at any time. The objective function to determine whether we step forward or not can be represented as:

$minZ$ = (the number of edges not covered by C)+(the size of C),

where C represents a candidate solution. At each step, we want to make sure Z does not become worse. Because we only remove at most one node at each step, we essentially want to check the number of edges not covered by C remains to be zero, which is also equivalent to C remaining to be a vertex cover.

This algorithm helps us avoid searching through a huge number of candidate solutions to find a next move because it uses a sorted list Q to pick up a next candidate solution which is most likely to improve a current solution. It narrows down the searching space at each step. In addition, this algorithm prevents a solution from hitting local minima by using a sort of randomized selection of a next state. However, it is time-consuming to remove a picked-up node from the sorted list Q at each step inside a while-loop because we implemented Q using a list data structure in python.

```
C= V of G = (V, E) //initial state;
Q = V of G = (V, E);
Sort Q in order of increasing degree;
while Q is not empty and t < cut-off time do
    v = random node in Q of the lowest degree;
    remove v from Q;
    remove v from C;
    if C is not a vertex cover then
        add v back to C;
    end
end
return C;
```

Algorithm 2: Stochastic Local Search Pseudocode

6.2 Time and Space Complexities

Our Stochastic Local Search implementation has three major processes:

1. Sorting V of $G = (V, E)$ in order of increasing degree
2. Counting the number of edges in $G = (V, E)$ not covered by a candidate solution C
3. Actual local searching process

- Time Complexity:
  Firstly, sorting takes $O(nlogn)$. A graph $G = (V, E)$ is represented as a dictionary in our implementation, where the key is a set of all nodes V and the value is a set of neighbors corresponding to each key node. Thus, sorting V of $G = (V, E)$ in order of increasing degree is equivalent to sorting keys of a dictionary by the size of corresponding sets. Secondly, counting the number of edges in $G = (V, E)$ not covered by a candidate solution C takes $O(|V| * |E|)$. Finally, our actual local search steps take $O(|V|^2 * |E|)$ because we search through each of nodes V of $G = (V, E)$, and for each of those, check the number of edges in $G = (V, E)$ not covered by a candidate solution C.
- Space Complexity:
  Space complexity of this algorithm is $O(|V|)$, which is the size of the set of nodes Q, sorted in order of increasing degree.

6.3 Plots

In order to plot QRTDs, SQDs, and Box plots, we used ten different results obtained with random seed = 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 for each of power.graph and star2.graph.

7 Empirical Evaluation

7.1 Branch and Bound Algorithm

This data was performed using Intel Core i5 Processor, 8GB RAM and Python 3. The Branch and Bound Algorithm was expected to take longest time out of other algorithms. However, the search space started with a maximum degree node,

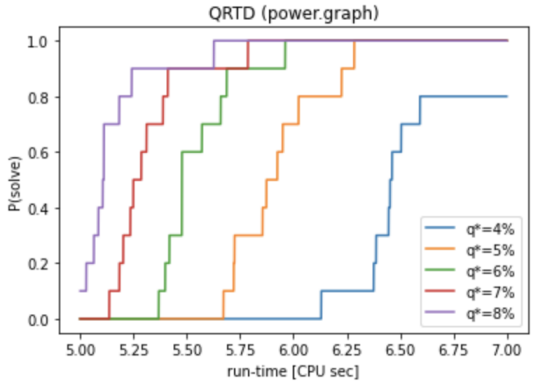

Figure 7. QRTD for power.graph

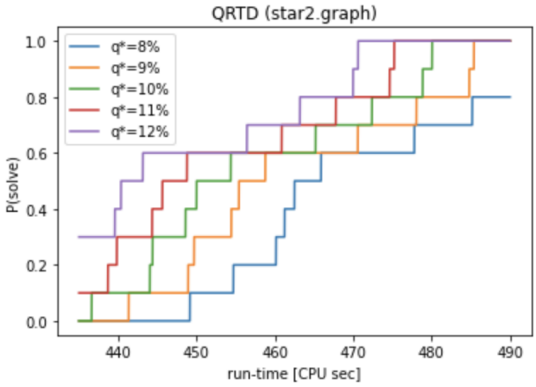

Figure 8. QRTD for star2.graph

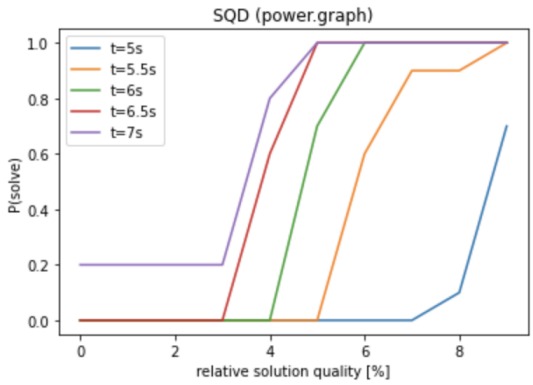

Figure 9. SQD for power.graph

the most "promising" node, the total time of traversal is reduced. The results shown in Table 3 were all found within 600 second time cut-off.

7.2 Approximation Algorithms

This data was performed using an Intel Core i7 processor, 8GB of RAM, and Python 3.

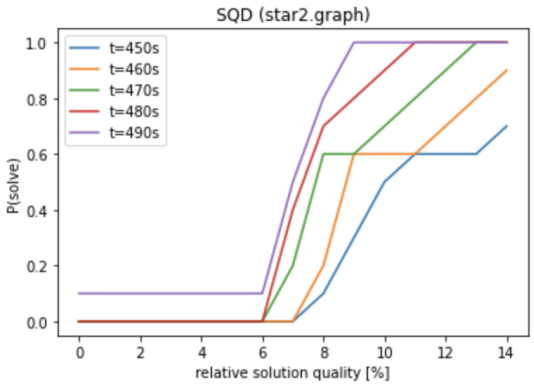

Figure 10. SQD for star2.graph

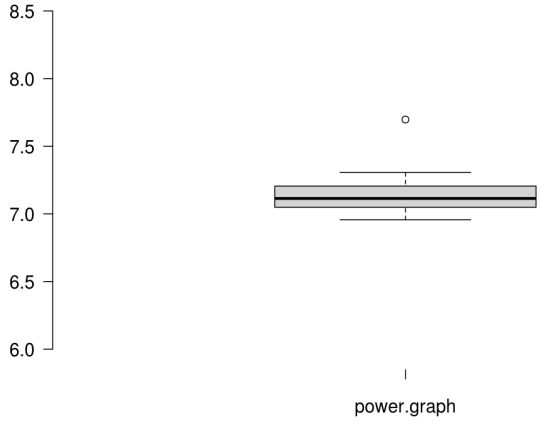

Figure 11. Box plot for power.graph (axis=run-time)

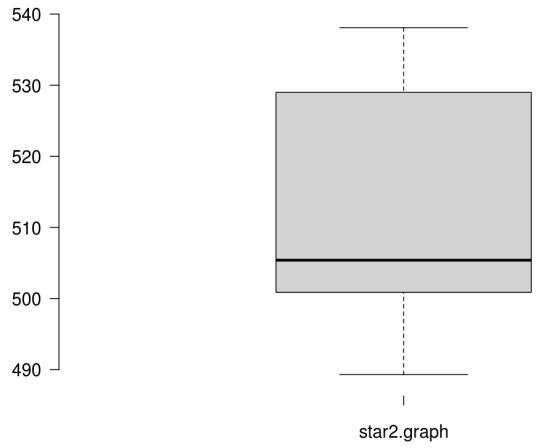

Figure 12. Box plot for star2.graph (axis=run-time)

7.3 Local Search (Stochastic Local Search)

This data was performed using an Intel Core i5 processor, 8GB of RAM, and Python 3. *RandomSeed* = 100 was used for all the graphs.

7.4 Local Search (Genetic Algorithms)

This data was performed using an Intel Core i5 processor, 8GB of RAM, and Python 3. *RandomSeed* = 100 was used for all the graphs. Genetic Algorithms tends to have a longer run time, as it depends on the size of the initial population and the time required to evaluate all those individuals. It also tends to find solutions slower due to crossover and mutation, however, it does tend to find some novel solutions that other methods don't.

8 Discussion

8.1 Approximation Algorithm

The Approximation Algorithm seemed to follow in line with our expectations of the time complexity besides the MVC for "Star2". "Star2" had the largest amount of edges, but completed faster than "Star". This difference could have been correlated to a significant amount of edges being removed early in "Star2" relative to "Star". However, besides this set, the rest of the graphs seemed to be aligned with the notion that the time complexity is O(E*E).

When it came our expectation of the evaluation criteria, the approximation algorithm did follow the poly-time 2-approximation as no output had a MVC of over 2 times the optimal. Each algorithm finished in a relatively quick time span.

8.2 Genetic Algorithms

The Genetic Algorithm approach results are in line with our expectations for relative error for smaller graphs, however for larger graphs such as "Star2" and "Power" it performed very poorly. This may be due to how genetic algorithms tend to maintain and exploit existing valid solutions using our elite pool and prevents too much negative change. For example, a valid vertex cover could contain every possible node in the graph. However, it obviously is far from the Minimum vertex cover. But, since it is a valid solution, the genetic algorithm fitness function would rate it higher than an invalid solution. Because of this, when large individuals are initialized in the first generation, they are more favored as they are more likely to be valid. It then takes a very long time for the large individuals to start shrinking down to approach the true minimum vertex cover. This may be why genetic algorithms are performing poorly on these larger graphs. Another hypothesis could be that the initial population size is too small, and because of this, individuals are too similar in later generations, causing it to get caught in local minima that it cannot escape. Because of the very poor results on larger graphs, the Stochastic Local Search would likely be better. As the run times and the relative quality of the solutions found by genetic algorithms were very poor, the QRTD and SQD plots for genetic algorithms were horizontal lines at 0, as the necessary relative quality would be extremely high in order to show the solutions, so in most cases, none

Table 1. Summary of Algorithms

	BnB	Approx	GA	SLS
as-22july06.graph				
Time	184.54	19.44	598.16	404.81
VC Value	3312	6008	22761	3325
RelError	0.0027	0.819	5.89	0.0067
delaunay_n10.graph				
Time	2.12	0.35	65.91	0.63
VC Value	739	912	777	744
RelError	0.051	0.297	0.105	0.058
email.graph				
Time	0.62	0.42	288.45	1.12
VC Value	605	818	659	612
RelError	0.019	0.377	0.11	0.030
football.graph				
Time	368.66	0.007	77.43	0.01
VC Value	94	110	94	97
RelError	0	0.17	0	0.032
jazz.graph				
Time	0.11	0.075	345.57	0.08
VC Value	159	182	160	160
RelError	0.0063	0.152	0.013	0.013
karate.graph				
Time	0	0.0006	0.213	0
VC Value	14	20	14	14
RelError	0	0.429	0	0
netscience.graph				
Time	1.04	0.37	179.79	0.85
VC Value	899	1200	950	899
RelError	0	0.335	0.057	0
power.graph				
Time	11.11	3.21	599.67	7.21
VC Value	2272	3626	3662	2272
RelError	0.031	0.646	0.66	0.031
star.graph				
Time	92.58	53.81	596.27	221.21
VC Value	7366	10262	10793	7227
RelError	0.067	0.487	0.56	0.047
star2.graph				
Time	129.5	46.22	598.26	489.30
VC Value	4677	6886	13963	4859
RelError	0.030	0.516	2.07	0.070
hep-th.graph				
Time	35.46	9.34	599.79	34.31
VC Value	3947	5760	7641	3937
RelError	0.0053	0.467	0.95	0.0028

of the solutions produced by genetic algorithms had met the quality threshold because none of them had good relative error for larger graphs.

8.3 Stochastic Local Search

Our implementation of Stochastic Local Search randomly picks up a next state from a set of nodes of the same degree to move to at each step, so the running time varies especially for dense or large graph because it has more choices of states we can move to at each step. The variance of running time is visualized by the box plots. Additionally, the running time of computing the objective function used in our implementation is O(|V| * |E|), so the running time is significantly affected by dense graphs. However, the result we obtained in terms of the accuracy was satisfactory without being stuck in a low-quality local minima.

9 Conclusion

In general, the choice of algorithm to solve MVC problem should be based on the required accuracy and available computing resources. If there is no limit to computing resources with high accuracy requirement, Branch and Bound algorithm is suitable. However, if quick approximation is required in spite of sacrificing the accuracy, heuristic algorithms such as the Local Search Algorithm or Approximation Algorithm should be used. Genetic Algorithms appear to find promising novel solutions in smaller graphs that other search algorithms don't find as easily, however, it tends to overexploit the valid solutions it already found and get caught in local minima, causing it to perform poorly on larger graphs. Due to nature of NP-Complete problem, if one can reduce any particular NP-Complete problem into a MVC, any algorithm outlined in this report can be applied.