CSE 6140 Fall 2020 Project

Minimum Vertex Cover

Kiarash Ahmadi
Georgia Institute of Technology
Atlanta, Georgia, United States of America
kahmadi3@gatech.edu

Jason Lu
Georgia Institute of Technology
Atlanta, Georgia, United States of America
jlu367@gatech.edu

Kipp Morris
Georgia Institute of Technology
Atlanta, Georgia, United States of America
kmorris9@gatech.edu

Sonia Sargolzaei
Georgia Institute of Technology
Atlanta, Georgia, United States of America
ssargolzaei7@gatech.edu

Introduction

This report outlines the different approaches that were taken in solving the minimum vertex cover problem where the minimum set of vertices that includes at least one endpoint of every edge of the graph is to be found. To solve this problem, four different approaches were taken: a branch-and-bound algorithm, an approximation algorithm, and two local search algorithms. The quality of each approach was then evaluated by assessing the relative error from the given optimal solution and the runtime. Our results have shown that different approaches yield a variety of results regarding efficiency and quality. Different approaches yielded different results in regard to these heuristics. The brand-and-bound approach sacrifices runtime in order to guarantee optimality, however sometimes the cutoff time is reached without any solutions discovered. On the other hand, the approximation and local search approaches do not guarantee optimality but do find solutions with small relative error in a reasonable amount of time. Between these latter three algorithms, it is evidenced that generally an increased runtime leads to a better-quality solution, and vice versa. Quantitative information on the results obtained can be found under Empirical Evaluation.

Problem Definition

The minimum vertex cover (MVC) problem is a basic combinatorial optimization problem. A vertex cover of a graph is a set of vertices that includes at least one endpoint of every edge of the graph. Thus, what the MVC problem attempts to find is the minimum set of vertices possible that include at least one endpoint of every edge in the graph. This problem has numerous applications in many fields from biology, supply chain engineering, to network security. The process of finding the minimum vertex cover is a combinatorial optimization problem because it consists of finding the different combinations of vertices that satisfy the vertex cover condition. This can be found through a myriad of algorithms leading to solutions of different levels of quality. Quality can be measured and evaluated via runtime and relative error from the optimal solution.

Related Work

Due to the prominence of the MVC problem, there are various existing discussions on the different ways this problem can be solved. One method consisted of performing a local search where the problem is viewed from the perspective of its dual problem: the Maximum Independent Set, where an independent set of graph G with vertices (V) and edges (E) is a subset of the vertices whose elements are non-adjacent [1]. This was done in two ways. The first approach determined in linear time whether the optimal solution can be improved by replacing a single vertex with two others that are non-adjacent to the vertex being removed. The second approach determined in $O(m\Delta)$ time (Δ is the highest degree in the graph) whether there are two vertices in the candidate solution that can be replaced by a set of 3 vertices. Another approach worked on designing more efficient local search algorithms by addressing two drawbacks that local search approaches often have: selecting a pair of vertices that are exchanged simultaneously and not having a strategy to decrease edge weighting techniques [2]. Thus, to combat this, the paper proposes a 2-stage exchange strategy and incorporates edge weighting without forgetting. In addition, the edge weights were decreased periodically. These minor adjustments helped to decrease the run time for local search algorithms. Another paper consisted of 6 different approximation algorithms for solving the MVC problem. These approaches are mostly greedy based and range from using commonly known approaches such as maximum degree greedy, depth first search, edge deletion, and greedy independent cover [3]. The benefit of these approaches is that an approximation ratio for each approach can be calculated which gives one a quality guarantee of the respective algorithm.

Algorithms

Approach #1: Branch-and-Bound

The branch-and-bound approach is generally used for solving combinatorial optimization problems. Due to the nature of these problems, they are often exponential in terms of time complexity because of the worst-case scenario that would require one to explore all permutations. In the branch-and-bound approach, we

search through the exponentially sized search space of all possible solutions, but we maintain a global upper bound on the cost of the optimal solution in addition to lower bounds on the costs of the solutions that can come from each partial configuration/search node. We then use those bounds to prune the search space of partial solutions that are no longer worth considering, with the hope that in practice the algorithm can work reasonably well on small problem instances. For our algorithm for the MVC problem, we calculate the bounds as follows:

- Global upper bound: The number of vertices in the smallest vertex cover that has been found so far. Initialized to the number of vertices in the graph.
- Lower bound for each partial solution's possible outcome: The number of vertices in the partial solution plus the size of a maximal matching on the remaining uncovered subgraph. The maximal matching is calculated greedily in O(|E|) time.

Pseudocode for the general branch-and-bound approach is shown in Figure 1. In addition to calculating bounds, we also had to clearly define subproblems, partial solutions, and the "Choose", "Expand", and "Check" portions of the pseudocode from the class lectures. Those five key things are described below:

- Partial solution: A set of vertices that is not yet a vertex cover.
- Subproblem: A subgraph of the input graph that is not yet covered by the partial solution.
- "Choose": The partial solution with the highest lower bound; in our case, this is the partial solution that has the largest size when combined with the size of the maximal matching on the remaining uncovered subgraph.
- "Expand": Given a partial solution, we expand it by selecting a new vertex we have not yet looked at and choose whether to include that vertex in the vertex cover. This gives two new partial solutions nodes: one where the new vertex is included and one where it is not included.
- "Check": Every subset of the vertices in the graph is a valid partial solution, so there is no work to be done at the "Check" step.

One additional thing to note is that we consider adding vertices to the vertex cover starting with the highest degree vertices first since they are likely to cover more uncovered edges.

```
function BNB( G )
Given graph G = ( V, E )
    F ← { (∅ , G) }
    B ←  (∞ , F)
    While F ≠ ∅ do
        Choose ( X, Y) in F – the most promising solution
        Expand (X, Y)
        Make new configurations  from (X, Y) → (Xi, Yi)
        For each new configuration in (Xi, Yi) do
                Check (Xi, Yi)
                If solution found then
                    If cost(Xi) < B cost then
                                B ←  (cost(Xi), (Xi, Yi))
                Else
                    If lb(Xi) < B cost then
                                F  ← F ∪ {( Xi, Yi)}
Return B
```

Figure 1: Pseudocode of Branch-and-Bound Approach

Time Complexity Analysis: Even though we can prune unpromising search nodes, the number of possible configurations explored is still asymptotically exponential in the number of vertices in the graph. In addition, for each configuration explored, we have to determine the remaining uncovered subgraph given the vertices that make up the partial solution, which takes O(|V| + |E|) time, and we also calculate a maximal matching on that subgraph, which takes O(|E|) time. Thus, the runtime is O((|V| + |E|) * 2^|V|).

Space Complexity Analysis: For each search node, we store an integer representing the quality of the solution (the number of vertices in the vertex cover) and a dictionary of size O(|V|) that tells us which vertices are in the vertex cover. The graph itself is only loaded into memory once, and the subgraphs that we create when exploring each search node are actually just views of the subgraphs based on the original graph object, saving a lot of memory space. Thus, the space complexity is O(|V| * 2^|V|).

The benefit of using a branch-and-bound algorithm is that it guarantees an optimal solution so long as we run it for long enough. We set an upper time limit of runtime to 10 minutes, so after this time has elapsed the program is terminated and output the best solution that we have found so far. The downfall is that we are not necessarily able to find the optimal solution (or even any solution for some of the larger graphs) within 10 minutes. If we do not set a time limit to the program, we will eventually find the optimal solution. However, if quality is a top priority, the branch-and-bound algorithm is a viable option.

One interesting implementation detail worth mentioning is that to avoid storing a graph object representing the remaining uncovered subgraph in each frontier search node, our algorithm calculates the uncovered subgraph upon visiting each node as explained in the space complexity analysis section. We tried storing the subgraphs in the frontier search nodes at first. Even though it made the algorithm faster, we found that on a machine with 8 GB of RAM we often ran out of RAM when running the algorithm on the

larger graphs. Not storing those extra graph objects prevented the algorithm from using up all of the 8 GB of RAM, but it also slowed it down, so this was an interesting learning experience and a good example of a time-space trade-off in algorithm design.

Approach #2: Approximation Algorithm - Maximum Degree Greedy

The maximum degree greedy approach consists taking an input that is a graph object. From there, one iteratively selects a vertex with the graph with the maximum degree. Then one removes the vertex from the from the graph in question and adds such vertex to the solution. This process continues until there are no remaining vertices in the graph of question. The approximation factor, A, a constant ratio bound, for this algorithm is the harmonic series $\Sigma 1/i$ from i = 1 to n, where n is the maximum degree of all vertices of the graph. So, $C \leq A*C'$, where C' is the optimal value of vertex cover. Below is the pseudocode for such approach.

```
function MDG( G )
Given graph G = ( V, E )
    C ← ∅
    While E ≠ ∅ do
        Select vertex u with a maximum degree
        V ← V – u
        C ← V ∪ u
Return C
```

Figure 2: Pseudocode of Maximum Degree Approach

The Maximum Degree Approach algorithm doesn't guarantee optimality under any circumstances. However, it does guarantee a solution that is within the approximation factor A of the optimal solution. Furthermore, a benefit of an approximation algorithm is that it runs much faster compared to other methods, as it can terminate on its own in a relatively short amount of time. This shows how quality is sacrificed for the sake of efficiency in time, which depending on the situation at hand could be a better option. The algorithm worst-case approximation ratio is $H(\Delta)$, with H(n) being the harmonic series, ($H(n) = \ln(n) + 0.57$) and Δ being the maximum degree of the graph.

Approach #3: Stochastic Local Search

Stochastic Local search is one of the more commonly used methods to solve combinatorial optimization problems. The general approach of the local search is to first initiate search space for the problem in question. From there, various methods are used to create an initial solution. Then, one evaluates the initial solution using an evaluation function. This then allows you to iteratively move among potential solutions to others and evaluate the performance of every solution. The pseudocode for this approach can be seen below.

```
function StochasticLocalSearch( G )
Given graph G = ( V, E )
    C ← MDG( G )
    While elapsed time < cutoff do
        While C is a vertex cover then
            C* ← C
            Randomly remove one vertex from C
        u ← Select exiting vertex from C
        C ← C \ {u}
        v ← Select entering vertex from V \ C
        C ← C ∪ {v}
    Return C*
```

Figure 3: Pseudocode of Stochastic Local Search Approach

The initial construction of the vertex cover for this approach was created using the Maximum Degree Greedy approach. Once an initial solution is found, the local search approach iterates through this initial vertex cover and randomly removes vertices until it is no longer a vertex cover. Then, an exit vertex from C is selected in which the vertex is one that produces the minimum increase in cost. The way cost was calculated was that given the total graph G and candidate solution in question C, calculate the total number of edges that are not covered by C. Once this exiting vertex is identified, it is removed, and an entering vertex is put into the candidate solution C. This entering vertex is randomly selected from the set of vertices that are not in the candidate solution C. This process is then repeated until either the elapsed time is past the cutoff or the approach reaches the optimum. The advantage with this approach is that it works relatively fast and well for most of the graphs given. However, when the graph size is larger there is a chance that this approach can get stuck at local optimums. In order to combat this, the search step was randomized by selecting a random entering vertex which allows for worser steps to occur. This in turn helps to improve the robustness and performance of the approach despite still not guaranteeing the optimal result.

Time Complexity Analysis: The time complexity based on our input G = (V, E) would largely be dominated by the initial construction of the VC through a maximum degree greedy approach, which would, at worst case, go through each vertex and sort each iteration, resulting in a O((|V|^2)log|V|) time complexity. Selecting an exiting vertex from the VC would take O(|V||E|) to analyze the cost of removing each possible vertex from the VC. Selecting an entering vertex and checking the validity of the VC takes O(|E|) to check each edge. Thus, initial construction of the VC dominates time complexity. However, for the majority of cases, the overall time complexity of the Stochastic Local Search algorithm will be defined by the cutoff time, which across different graph instances should remain constant (for our experimentation this was set to 600 seconds).

Space Complexity Analysis: In our code, we originally created the graph data structure with |V| vertices and |E| edges resulting in O(|V|+ |E|) space taken up. The vertex cover was stored in a set data structure and occasionally copied to test exiting vertices, so it

took up an addition O(|V|) space. So overall, space is dominated by the graph itself of O(|V||E|), as no major additional storage is present that would affect the space.

Approach #4: Simulated Annealing Local Search

In this approach, we implement an efficient simulated annealing algorithm inspired from the paper [4] for the Minimum Vertex Cover problem. Simulated Annealing can be considered as a version of an iterative improvement algorithm, which allows various types of transitions that may be in the opposite direction of the goal. Generally, in Simulated Annealing implementation, the temperature progressively decreases from an initial positive value to zero. At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions, which during the search respectively remain at 1 (or positive) and decrease towards zero. [5]

In our problem, the goal is minimizing the size of vertex cover set. So, in each iteration, we surely step toward decreasing the cost function, and sometimes with a probability that depends on the temperature and the difference in cost, we may step toward increasing the cost function, hoping that our algorithm can get out of a possible local optimum. The pseudocode for this approach can be seen in the figure below:

```
function SA Local Search( G )
Given graph G = ( V, E )
    Temp← 1
    Final_temp← .0000001
    alpha ← .999
    C ← MDG( G )
    While temp < final_temp do
        u ← Select random vertex from G
        if u is not in C then
                delta ← degree of u
                p ← e^(-delta/current_temp)
                if  p > random number between 0 and 1
                    C ← C ∪ {u}
        Else
                C* ← C \ {u}
                If C* is a vertex cover
                    C ← C*
        Temp ← Temp * alpha
    Return C
```

Figure 4: Pseudocode of SA Local Search Approach

In our implementation, the state of node *i* can be determined by whether it is added to the vertex cover or not. We show this state of nodes with a dictionary, containing the node numbers as keys and 0 or 1 as values. First, we create our initial solution by randomly selecting the nodes in the graph and them to our solution until we have a vertex cover. Then, we specify the parameters of the Simulated Annealing algorithm, e.g., initial and final temperature, etc. At each time, we select a random node from the graph and check if it is in vertex cover or not. If it is not present in VC, adding it would increase the cost function by the degree of the node, and we add it with the probability $e^{-\text{cost\_difference} / T}$. If it is present in VC, we delete it only if the deletion of the node doesn't cause any edges to get uncovered. We stop the algorithm if the temperature is lower than a threshold, if we exceed the cutoff-time specified for running the algorithm, or if we do more than 10000 iterations without improving the solution. Throughout the running process, we keep the improved solutions and their time in the '.trace' file of the graph. We repeat this process for 10 random seeds and get the average of our results as seen in table 5. As for the parameters of SA algorithm, since we had a good initial solution, we wanted to penalize adding too many nodes to the graph, and since removing each node from the solution was safe, we wanted to encourage the algorithm to find ways to remove nodes while still maintaining the vertex cover. Therefore, we started from relatively low temperature, which helped our algorithm avoid adding too many nodes to vertex cover set by bringing down the probability of going towards worse cost.

Time Complexity Analysis: Again, the time complexity of this algorithm would be dominated by the construction of the initial solution with a maximum degree greedy approach $O(|V|^2 \log |V|)$. Also, since we have some thresholds on time and number of iterations, those would impact our running time. But in the worst case, at each iteration, we pick a random node from our graph G, and either add it to the VC or not. In case we add it, the operation is simply O(1), and if we want to remove it, we need to do a check on the VC set to make sure it covers all the edges. For that, we loop on all the edges on the graph. Therefore, the overall cost for the algorithm is $O(|V|^2 \log |V|) + O(|V| \cdot |E|)$.

Space Complexity Analysis: For keeping track of VC nodes, we keep a dictionary of size |V|, and we need to iterate over all the edges of the graph. Thus, the space complexity of this algorithm is O(|V| + |E|).

Empirical Evaluation

The following table helps to show the different platforms that were used for the different approaches.

Table 1: Platform Overview

Processor	RAM	Platform approaches used
2.6 GHz 6-Core Intel Core i7	16 GB	3
1.4 GHz Quad-Core Intel Core i5	8 GB	1
2.3 GHz Quad-Core Intel Core i5	8 GB	3
2.6 GHz Intel Core i5	8 GB	2,4

This table helps to give perspective when evaluating the results obtained by different approaches as the platform these approaches are being used on can affect the solution due to the cutoff time constraint. These four approaches were evaluated in terms of runtime and relative error where relative error is the percent difference of each respective approach and the given optimal value. For the case of the local search approaches, 10 separate runs with different random seeds were performed for each respective graph. These results were then averaged for each graph. Below one can see the results that were obtained with each approach.

Table 2: Branch-and-Bound Approach Results

Branch-and-Bound Results			
Dataset	Time (sec)	VC Value	Relative Error
jazz.graph	195.37	182	0.15
karate.graph	1.23	14	0.00
football.graph	51.47	105	0.12
as-22july06.graph	>600	-	1.00
hep-th.graph	>600	-	1.00
star.graph	>600	-	1.00
star2.graph	>600	-	1.00
netscience.graph	307.17	1453	0.62
email.graph	588.73	873	0.47
delaunay n10.graph	260.62	935	0.33
power.graph	>600	-	1.00

Table 3: Maximum Degree Greedy Approach Results

Approximation Algorithm – Maximum Degree Greedy Results			
Dataset	Time (sec)	VC Value	Relative Error
jazz.graph	0.018	160	0.013
karate.graph	0.00051	14	0.00
football.graph	0.017	96	0.021
as-22july06.graph	88.063	3312	0.0027
hep-th.graph	24.48	3947	0.0053
star.graph	122.42	7366	0.067
star2.graph	74.55	4677	0.030
netscience.graph	0.82	899	0.00
email.graph	0.46	605	0.019
delaunay n10.graph	0.24	740	0.053
power.graph	8.26	2272	0.031

Table 4: Stochastic Local Search Approach Results

Local Search Approach #1 (Stochastic) Results			
Dataset	Time (sec)	VC Value	Relative Error
jazz.graph	12.74	158.10	0.00063
karate.graph	0.00	14	0.00
football.graph	1.97	95	0.011
as-22july06.graph	92.25	3309.00	0.0018
hep-th.graph	162.32	3937.60	0.0030
star.graph	554.68	7035.30	0.019
star2.graph	566.64	4665.90	0.027
netscience.graph	0.58	899	0.00
email.graph	107.86	598.90	0.0082
delaunay n10.graph	129.67	717.67	0.021
power.graph	531.49	2239.70	0.017

Table 5: Simulated Annealing Local Search Approach Results

Local Search Approach #2 (Simulated Annealing) Results			
Dataset	Time (sec)	VC Value	Relative Error
jazz.graph	0.53	160	0.012
karate.graph	0.25	14	0.000
football.graph	0.40	96	0.021
as-22july06.graph	105.01	3387.66	0.035
hep-th.graph	50.51	4235	0.078
star.graph	144.18	7060.1	0.022
star2.graph	117.23	4643.5	0.022
netscience.graph	4.17	899.3	0.000
email.graph	4.17	602	0.013
delaunay n10.graph	3.64	730.1	0.038
power.graph	37.03	2251.8	0.022

One interesting point that can be drawn from these results are that one can obtain a lower bound on the solution quality by looking for the highest relative error in approximation approach results. This would be a lower bound because one can rule out any solution with a relative error higher than the one obtained from implementing the approaches. Furthermore, quality runtime distribution plots (QRTD), solution quality distribution plots (SQD) ,and boxplots were created for the local search approaches to help convey the behavior of the respective approaches. These plots were created for two graphs in question: ‘star2.graph’ and ‘power.graph’. The plots can be seen in the subsequent figures below.

Figure 5: QRTD Plot of Power Graph Using Stochastic Local Search over 10 Runs

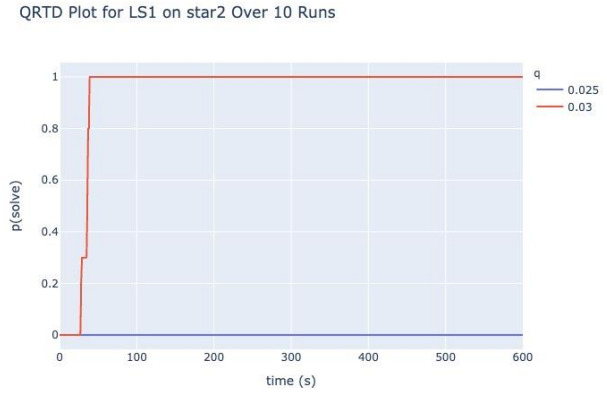

Figure 6: QRTD Plot of Star2 Graph Using Stochastic Local Search over 10 Runs

Figure 7: SQD Plot of Power Graph Using Stochastic Local Search over 10 Runs

Figure 8: SQD Plot of Star2 Graph Using Stochastic Local Search over 10 Runs

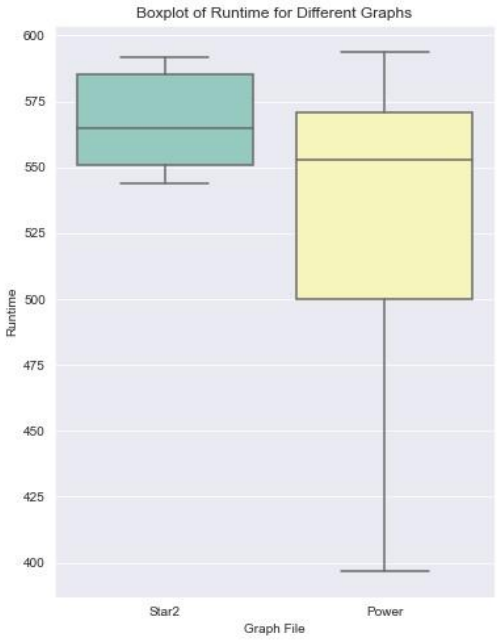

Figure 9: Boxplot of Runtime for Star2 and Power Graphs Using Stochastic Local Search Approach over 10 Runs

Figure 10: QRTD Plot of Power Graph Using SA Local Search over 10 Runs

Figure 11: QRTD Plot of Star2 Graph Using SA Local Search over 10 Runs

Figure 12: SQD Plot of Power Graph Using SA Local Search over 10 Runs

Figure 13: SQD Plot of Star2 Graph Using SA Local Search over 10 Runs

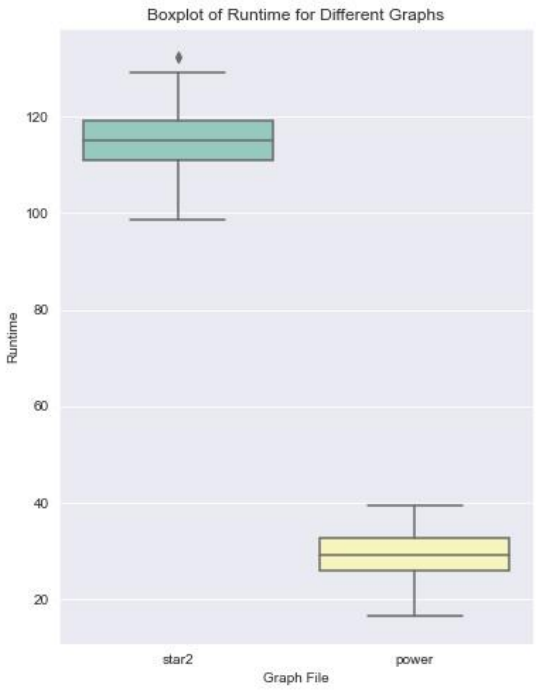

Figure 14: Boxplot of Runtime for Star2 and Power Graphs Using Simulated-Annealing Local Search Approach over 10 Runs

These figures help to illustrate that the two local search approaches were able to get close the optimal solution while still not reaching the cutoff time of 10 minutes. Furthermore, they help to show the variation in runtimes for different runs (and random seeds) as seen by the boxplots.

Discussion

From our experimentation, it is shown that different algorithms yield different results in regard to runtime efficiency and optimal quality. The branch-and-bound approach results help to confirm and challenge prior knowledge about the advantages and disadvantages of such an approach. For several of the larger graphs, the approach failed to find any solution before the cutoff time of 10 minutes elapsed. However, for the graphs for which the algorithm found a solution before the cutoff time, the algorithm took longer than the approximation and local search approaches. The approximation approach yielded quick results as it operates in polynomial time. However, the VC sizes it produced were far from the optimum in comparison to the local search approaches. This result is expected, as approximation algorithms work to find a solution quickly but do not guarantee the optimal solution.

In the stochastic local search approach, one can see that the solution performs best in terms of relative error but at the expense of a longer runtime. In addition, as one can see in the results the larger graphs tend to produce a higher relative error. This can be due to the increase in combinatorial possibilities because of a larger search space. Although the entering vertex was selected at random, the approach often would plateau at a local optimum. An approach in fixing this problem would be to introduce a probability factor when deciding an entering vertex so we would have the possibility of exploring other optimums. In addition, a use of a different data structure that evaluates whether the candidate solution is a vertex cover would decrease the time complexity of the approach, providing more simplicity in the analyzation through the program and more efficiency in the runtime of the code. While the algorithm will terminate only based on the cutoff time we set for the code, we will yield better results with the progression of time.

In the Simulated Annealing implementation results, we can see that our solution can take a longer time to converge for large graphs. This is normal due to the huge size of the search space. However, the good side of this approach is the relatively shorter time to return a solution compared to other approaches which can take much longer, and the solutions created in this short period of time also look good. Giving the algorithm a good enough initial solution was important in reaching the local optimum fast. We still can improve our solution, especially for the large graphs, if we find a way to check if we still have a vertex cover after removing a node efficiently. Also, since there is a limit on removing nodes and improving the solution, sometimes the algorithm doesn't even change the initial solution. This is fair because as mentioned, our initial solution is relatively close to the optimal solution.

Conclusion

From the development of our algorithms, it is evidenced that different approaches to solve combinatorial optimization problems lead to different results in efficiency and quality. These different approaches serve as the general methods to cope with NP-Complete problems like the MVC. The Branch and Bound algorithm sacrifices runtime for guaranteeing optimality as long as the algorithm finishes, while the Approximation and Local Search algorithms show improvements by sacrificing quality for the solution, with the Approximation algorithm guaranteeing a bound on the solution. Depending on circumstances and goals of an objective, certain approaches would be preferrable over others in different cases. However, when just evaluating the performance of the approaches as to which one has the best relative error, the stochastic local search approach performs best. The MVC optimization problem serves as one of the most applicable problems in our society today, with several applications in modern industry and research. In the future, this will only become a more relevant scenario for more applications. The approaches we've presented to solve the MVC problem are imperative towards solving complex combinatorial optimization problems in the world today and in the future.

References

[1] Diogo V. Andrade, Mauricio G.C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. Journal of Heuristics, 18(4):525–547, 2012.

[2] Shaowei Cai1 Kaile Su and Abdul Sattar. Two new local search strategies for minimum vertex cover. 2012.

[3] Francois Delbot and Christian Laforest. Analytical and experimental comparison of six algorithms for the vertex cover problem. Journal of Experimental Algorithmics (JEA), 15:1–4, 2010.

[4] Xinshun Xu, Jun Ma, An efficient simulated annealing algorithm for the minimum vertex cover problem, Neurocomputing, Volume 69, Issues 7–9, 2006

[5] https://en.wikipedia.org/wiki/Simulated_annealing