Minimum Vertex Cover

CX4140 Project Report – Team C

Matthew Berman
Georgia Institute of Technology
mberman3@gatech.edu

Tianxiao Liu
Georgia Institute of Technology
tianxiao@gatech.edu

Yaxiong Liu
Georgia Institute of Technology
yliu710@gatech.edu

Jay Patel
Georgia Institute of Technology
jay11patel@gatech.edu

1. INTRODUCTION

The Minimum Vertex Cover (MVC) problem is a classical NP-complete problem with many practical applications. A minimum vertex cover is the smallest number of vertices necessary to form a vertex cover in a graph. In this paper, the implementation of a branch-and-bound algorithm, an approximation algorithm, and two local search algorithms will be discussed, along with the results returned by each algorithm.

2. PROBLEM DEFINITION

Given an undirected graph $G = (V, E)$, a vertex cover is defined as a subset $C \subseteq V$ such that $\forall (u, v) \in E$: $u \in C \vee v \in C$. The minimum vertex cover problem entails minimizing $|C|$. For this project, four algorithms were implemented to solve the MVC problem: a branch-and-bound algorithm, an approximation algorithm utilizing a modified depth-first-search, a local search with hill climbing, and a local search using simulated annealing.

3. RELATED WORK

The Minimum Vertex Cover problem can be applied to many real-life situations and the many different solutions to these types of problems can be very useful. One of the major areas that this problem is applicable is computational biology. There is particular project that is interesting and it one that is originated in research at Glycodata. This particular problem is involved with areas in glycobiology and bioinformatics. There was a project that involved redeveloping drugs using glycoproteins and the goal is to determine what are the building blocks of the glycans consisting of variations of glycoproteins in a solution, and determining the connectivity of the building blocks was significant also. They used a chip-based technology called GMID (Glycomolecule ID) which uses fingerprints to identify glycomolecules. Through the creation of this problem by creating vertices as the building blocks and the connectivity as the edges, the minimum vertex cover is the prime goal in this problem. The minimized the number of experiments needed to cover the information graph. The results consisted of using a variation of algorithms, including 2-approximation, greedy algorithms, and a primal-dual algorithm. This is one of many applications in which the Minimum Vertex Cover problem can be useful. There are many other applications for the Minimum Vertex Cover problem and various techniques are used in solving these graph theory problems.

4. ALGORITHMS

4.1 Branch and Bound

Description:

The branch-and-bound algorithm begins with an empty set of vertices. The 2-approx is used to establish an initial crude upper bound. The algorithm selects the vertex with the highest degree of uncovered neighbors. The algorithm looks at this vertex and either adds the vertex to the partial solution, or leaves it out and adds its neighbors, depending on which is more promising. If a branch is determined not to be promising, it is not explored any further. Upon finding a vertex cover, the upper bound is updated to the size of the current best cover. The algorithm terminates when no promising branches remain.

Guarantee:

Given enough time, the branch-and-bound algorithm will return an optimal solution, because every possible solution is either explored or intentionally ignored because it cannot be optimal.

Time and Space Complexity analysis:

In a worst-case scenario branch-and-bound simply explores every subset of vertices in *V*. This results in a time complexity of $O(2^n)$ for the algorithm.

Pseudocode:

```
BranchAndBound
    Input = Graph G, int cutoff
    Output = best found vertex cover VC
        if (G.usedVertices > upper)
                return
        if(timeElapsed > cutoff)
                end
        if(G.unusedEdges == 0)
                upper = g.usedVertices.size()
                output = g.UsedVertices
        Vertex v = G.getHighestDegree()
        branch(v)
```

Branch(), mentioned in the pseudocode determines if v or its neighbors are promising by comparing the used vertices and the approximation of the subproblem to the current upper bound.

4.2 Approximation

Description:

The approximation algorithm for finding MVC implemented here is a modification of the DFS search. The algorithm starts with vertex 1, runs DFS on the given graph and outputs the non-leaf vertices in the DFS tree.

Proof of correctness:

This output is a vertex cover if there is no edges between the leaf vertices. By the nature of a DFS tree, if there is an edge between some leaves (u, v), then we could have kept going and traversed that edge when hitting one of u and v, thus one of the is not actually a leaf. So we reach a contradiction, showing that this output has to be a vertex cover.

Proof of 2-approximation:

This algorithm achieves an approximation ratio of 2. That is, relative error should be less than or equal to 1. Consider the graph G and its DFS tree T. Any vertex cover of G must also be a vertex cover of T, which is a subgraph of G. opt(T) <= opt(G). Now consider finding the the optimal vertex cover of T.

We may assume that no leaves are in the optimal vertex cover: for any leaf covered, we may as well cover the parent instead. Also, at least one of the leaf and the parent must be covered. Thus opt(T) covers all parents of leaves. We can then remove all these nodes, and cover the remaining graph iteratively. Let L = P(0) be the leaves of T, P(1) be the parents of P(0), P(2) be the parents of P(1) that are not themselves in P(1), ..., P(i) are the parents of P(i−1) that are not in any previous Pj, etc. Let M be the maximum such index such that P(M) is covered by opt(T). Either P(M) or P(M+1) contains the last few vertices of the tree. Set P(M+1) = 0 if the above enumeration ends at P(M). We observe that $P(0) \geq P(1) \geq ... \geq P(M) \geq P(M+1)$, since multiple children can only have one parent. opt(T) costs P(1) +P(3) + P(5) + ... +P(M), while our algorithm costs:

$$P(1)+P(2)+...+P(M)+P(M+1)$$
$$\leq P(1)+P(1)+P(3)+P(3)+...+P(M)+P(M)$$
$$= 2opt(T)$$
$$\leq 2opt(G)$$

which gives a 2-approximation.

Pseudocode:

```
function DFSvertexCover(Array AdjacencyMatrix[1...n][1...n])
    Array visited[1...n] = false
    vertexCoverSize = 0
    Array vertexCover[1...n] = null
    for i = 1 to n do
        DFS(i, visited, AdjacencyMatrix)
    end
    Return vertexCoverSize, vertexCover

function DFS(int i, Array visited, Array AdjacencyMatrix)
    counter = 1;
    visited[i] = true;
    if i not a leaf then
    //i.e. if all of i's existing neighbors have been visited
        vertexCoverSize++;
        vertexCover[vertexCoverSize] = i;
    end
    for j = 1 to n do
        if (AdjacencyMatrix[i][j] && !visited[j]) then
            DFS(j, visited, AdjacencyMatrix);
        end
    end
```

Time and Space Complexity analysis:

Here we used adjacency matrix to calculate the DFS tree and we start at vertex 1. Classical DFS search would not work here because some of the input graphs are unconnected, and classical DFS has no way to know the unconnected parts of the graph if started from a random source. For every row in the adjacency matrix, which represents the connectivity of a single vertex, we recurse on all its neighbors until it has no unvisited neighbors, which indicates that every element in the adjacency matrix is checked once. Since the matrix is of size $|V|^2$, both time complexity and space complexity is $O(|V|^2)$.

4.3 Local Search – Hill Climbing

The basic idea of Hill climbing is to keep looking at the neighbors of our current solution. The neighbor is also a vertex cover and has exactly one less vertex than the current solution. The algorithm will be automatically ended when the solution reaches the local maxima. The evaluation of a solution is based on the size of the solution and whether it is a vertex cover. The algorithm will not consider any solution that is not a vertex cover, and the value of the evaluation is inversely proportional to the size of the solution. In other words, the smaller the solution, the higher the evaluation value. Cutoff time is used in this algorithm to ensure the algorithm will be ended in given time frame. However, none of the runs had reached the cutoff time (3600s), therefore cutoff was irrelevant. The edge deletion algorithm is utilized to produce an initial solution. The edge deletion algorithm is introduced in the paper *Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover* Problem (François Delbot and Christian Laforest). Edge Deletion iterates over all edges in E. It appends the two vertices of the current edge to the vertex cover, and then removes the current edge as well as any edges that contain either of the two vertices removed. Next, it sorts all vertices in the initial solution from the smallest degree to the largest, because by numbers of experiments conducted, removing a small degree vertex first produces better result. Then, it passes the vertex cover to the next step to perform the hill climbing loop. It iterates over all vertices in the vertex cover and tried to remove them one by one. It would not remove a vertex, if that removal made the vertex cover no longer a vertex cover. In order to do this, before removing any vertex, it first checks if all of the neighbors of that vertex are in the vertex cover, because each edge must have at least one end in the vertex cover. Eventually, it would output the result which is the MVC it found. Technically, this algorithm can be stuck at the local maxima, therefore it does not guarantee to find the optimal solution.

```
LOCAL_SEARCH_HILL_CLIMBING
    Input = graph G
            Cutoff time CUTOFF
    Output = minimum vertex cover VC
        VC←EdgeDeletion(G)
        Sort VC from the smallest degree to the largest degree
        current_vertex←first vertex in VC
        While elapse_time < CUTOFF
            If VC contains all neighbors of current_vertex
                VC.remove(current_vertex)
            If current_vertex is not the last vertex in VC
                current_vertex = next vertex
            Else
                Break the while loop
        End while loop
        Output VC
```

Time and Space Complexity analysis:
First, the Edge Deletion algorithm iterates the entire edge list for each edge. Therefore, in this step, the running time is $O(|E|^2)$. Next, the algorithm uses selection sort for the sorting task. The running time of this part is $O(|V|^2)$. Next, in the while loop the running time of this part is $O(|V|^2)$. Therefore, the overall running time complexity is $O(|V|^2+|E|^2)$.
The algorithm creates a copy of E in Edge deletion and a copy of V in sorting. Therefore, the time complexity is O(|V|+|E|).

4.4 Local Search – Simulated Annealing

Another algorithm that we implemented was the Simulated Annealing algorithm for this problem. This is another Local Search algorithm in which we will be transitioning between good or bad solutions depending on a cooling function. The whole concept of simulated annealing revolves around using a probabilistic technique to find an optimal solution for the problem, in which a cost-decreasing or cost-increasing solution will be chosen based on a function.

The algorithm begins with a base temperature and is constantly cooled after every iteration of the algorithm. The condition used for this particular version of Simulated Annealing was the metropolis condition, which is the following formula : e ^ (-(|delta(cost)| / Temperature). This condition works as long as the temperature is above a certain absolute temperature. The delta cost will be the change in cost between the previous solution and the neighboring solution that we will be randomly selecting. The cost function will the the degrees of the vertex being removed after every iteration of the algorithm. The general idea of using the metropolis condition is to show that at high temperatures, the algorithm is more likely to select worse solutions and jump around more to find an optimal solution. When the temperature becomes lower, the algorithm is more conservative and it will try to pick better solutions only. That is the reason why the temperature will start off high so initially the algorithm will pick random solutions, and then as the temperature gets lower it will begin to pick only the good solutions.

This local search algorithm using simulated annealing has its own unique methodology in finding the optimal solution for the minimum vertex cover problem. First we begin with an initial solution using the maximum degree greedy algorithm, in which we pick a vertex to add to our vertex cover solution with a max degree and then delete all the edges that are adjacent to it. Then we continue to pick the next maximum degree vertex until all the edges are covered and then we have one vertex cover solution. After we get our initial solution we begin our loop in which we use the metropolis condition. In the loop, we first find our next solution by picking a random vertex in the current vertex cover and remove it if it still creates a vertex cover after removal. Then we measure the cost of the removal of that vertex by taking the degree of that vertex. We compare the cost with the previous cost, and if the cost was lower we pick that solution. The end goal is to pick the vertex with the lowest costs. But if the solution returned a higher cost, then we would use the metropolis condition and plug in the delta cost into the following function: e ^ (-(|delta(cost)| / Temperature) . We compare the result of that function with a random double to check if our result was greater than that random double, and if it is then we will use the new solution regardless if it is a better solution or not. This is where the temperature comes into play, since if the temperature is high we will be picking the worse solution to continue with, but the function wouldn't allow a bad solution to be picked. After picking the solution, we cool the temperature using a cooling function and then continue with the algorithm. The following pseudocode will depict this algorithm:

SimulatedAnnealing (Graph G)
Input: Graph G = (V, E)
Output: Minimum Vertex Cover of G

```
VC := new ArrayList<String>;          // init solution list
curr := MaxDegreeAlgo(Graph)          // get initial Vertex Cover
nextSol := new ArrayList<String>      // init next solution

Temp := 10000;                        // set the start Temp
Cooling := 0.9999;                    // cooling function
absTemp := 0.00001;                   // the final absolute Temp

While ( temp < absTemp) {
        nextSol = getNeighborSol(Graph, curr);
        if( cost(nextSol) < cost(curr)) then      // cost is better
                                                  //of new sol then
                curr = nextSol;       // pick the new solution

        else if ( e^(-|deltaCost| / Temp) > randomDouble)  then
                curr = nextSol;       // Metropolis Condition:
                                      // accept worse solution
        endIf
        temp = temp * cooling;        // cooling function
endWhile
Return curr;        // Current output VC
```

Time and Space Complexity Analysis:

This algorithm gets an initial solution through the Max Degree Algorithm which goes through the whole graph through all the vertices and the edges because we need to know the degree of each vertex. Therefore for the initial solution has a O(|E| + |V|) time complexity. For this algorithm, I first sorted the vertices efficiently by their degree, which was in O(n log n) time complexity, instead of finding the max every time.

Then for the main algorithm using the simulated annealing approach, we iterate through the while loop as long as the temperature is cooling which is set value so it will be O(n) iterations. Then we are finding the next solution by removing a random vertex from current solution and looking through its adjacent edges to make sure it will result in a valid vertex cover. Since our hashmap has a O(1) lookup time, it will only take O(|E|) to go through all the adjacent edges to make sure both vertices in every adjacent edge is within the vertex cover still after removal. So each time we are picking a random vertex so in the end we will be going through every vertex which will be O(|V|) time. Overall simulated annealing goes through O(n) temperature steps since it is a set defined amount and it is not in accordance to the edges or vertices. In every temperature step we are going through the adjacent edges of one vertex which is O(|E|) time. So a possible

time complexity can be O(n*|E|), but since this method is a heuristic that can keep picking random solution, then we may fail to reach the global minimum vertex cover so it may result in O(∞) time.

The space complexity for this algorithm is determined first by the max degree greedy algorithm which use O(|E| + |V|) space for storing all the edges and then removing them as they all get used and storing the sorted vertices. Then the main simulated annealing algorithm only uses O(|V|) space for saving the current and next solutions of the vertex covers. Overall the space complexity of the full algorithm would be O(|E| + |V|).

Discussion of Simulated Annealing:

Some of the automated tuning used for the simulated annealing algorithm was based on the size of the graphs. When the algorithm is ran it will measure the size of the initial vertex cover returned by the max degree greedy algorithm, and if it is below 1000 vertices then it will use a smaller starting temperature and a faster cooling method, since we want to pick only the better solutions and it is only a small amount of vertices. But when the size is higher than 1000 vertices, it would be better if some worse solutions are picked to vary the type of solutions so the starting temperature is higher with a slower cooling method. When this was tested the graphs with less than 1000 vertices had larger runtimes and larger solutions, so this methodology was not used.

There were multiple data structures and approaches that were used for this algorithm and tested to find the most optimal solutions. First we will discuss the data structures and then the different methods approached.

Data Structures:

The data structure used to represent the graph was a hashmap in which the vertex was the key and the value was an arraylist of adjacent vertices. Then the edges were stored in an arraylist of edge class, by iterating through through the hashmap of the graph. Then finally our resulting vertex cover was stored as an arraylist of the vertices. The reason the hashmap was implemented was because it has O(1) lookup time and every vertex can be found quickly.

Methodology:

The vertices were sorted efficiently before finding the max degree so it would save time rather than searching for the max degree vertex every time. The Java Collections sort method was used which results in O(n*log(n)) time complexity.

There were many other algorithm approaches taken to test out different methodologies to find an optimal solution. Some approaches were to try using the percentage of vertices uncovered as the cost instead of the degree of the removed vertex. Another approach was to also to put the vertices in order by degree and then measure the position of the vertex being removed and then use that as the cost. These different approaches did not result in better solutions and took a longer time to compute.

5. EMPIRICAL EVALUATIONS

5.1 Branch and Bound

- Platform:
  - CPU: 2.5 GHz Intel Core i5
  - RAM: 8 GB
  - Language: Java
  - Compiler: Java Programming Language Compiler (javac)

All tests were run from the Eclipse IDE, with a cutoff time of 1200s (20 minutes). The results obtained are detailed below. A "-" indicates that no vertex cover was discovered within the time limit.

Cutoff = 1200s	Branch and Bound			
Dataset	Time(s)	VC Value	OPT	Rel Err
jazz	8.2	159	158	0.0063
karate	0.047	14	14	0.0000
football	0.484	94	94	0.0000
as-22july06	1200	-	3303	-
hep-th	1200	-	3926	-
star	1200	-	6902	-
star2	1200	-	4542	-
netscience	1200	933	899	0.0378
email	1200	605	594	0.0182
delaunay_n10	1200	739	703	0.0487
power	1200	-	2203	-

For the small datasets, branch-and-bound effectively finds an optimal solution. For medium-sized datasets, it gets near an optimal solution but runs out of time before reaching the optimal. On the large sets, it does not even find a vertex cover within the time limit. In addition to the long running time of the algorithm itself, the algorithm clogs up memory, creating a new graph at each branch. In hindsight, the Vertex class is unnecessary, as is much of the background cloning of data structures. Removing these issues would improve the efficiency of the algorithm.

5.2 Approximation

- Platform:
  - CPU: 2.9 GHz Intel Core i7
  - RAM: 8 GB 1600 MHz DDR3
  - Language: Java
  - Compiler: Java Programming Language Compiler (javac)

Theoretically, this approximation algorithm is guaranteed to have 2-approximation ratio. Based on the relative error calculation formula $RelErr = (Alg - OPT)/OPT)$, relative error should be less than 1. Below is the table showing execution results for the given dataset based on the approximation algorithm.

Cutoff =10s	Approximation			
Dataset	Time(s)	VC Value	OPT	RelErr
jazz	0.002654	184	158	0.16
karate	0.000121	17	14	0.21
football	0.001079	107	94	0.14
as-22july06	1.879186	4283	3303	0.30
hep-th	0.262661	4407	3926	0.12
star	0.466081	8795	6902	0.27
star2	0.788725	5688	4542	0.25
netscience	0.035966	948	899	0.055
email	0.047218	728	594	0.23
delaunay_n10	0.022326	905	703	0.29
power	0.128560	3127	2203	0.42

Vertex cover sizes calculated here by approximation algorithm provide a lower bound of the optimum solution. From the table above, we observe that the highest relative error is 0.42, which is less than half of the theoretical relative error 1. This may imply that approximation algorithms actually can produce a tighter lower bound for the optimum solution than expected.

5.3 Local Search – Hill Climbing

- Platform:
  - CPU: 2.70 GHz 2.71GHz Intel Core i5-6400
  - RAM: 8 GB
  - Language: Java
  - Compiler: Java Programming Language Compiler (javac)

The highest relative error is 6.0%, and the lowest relative error is 0.0%, which indicates an optimal solution. The algorithm found a solution with the highest relative error on delauanay_n10. Optimal solutions were found for the graphs karate and netscience.

	Local Search (Hill Climbing)						
	Phase 1: Edge Deletion			Phase 2: Hill Climbing			
Dataset	Time (s)	VC Value	RelErr	Time(s)	VC Value	RelErr	Rand Seed
jazz	0.025	188	0.190	0.092	159	0.006	256
karate	0.000	22	0.571	0.007	14	0.000	4
football	0.006	108	0.149	0.022	95	0.011	4096
as-22july06	2.193	5996	0.815	20.921	3310	0.002	64
hep-th	0.552	5728	0.459	3.887	3956	0.008	1024
star	3.548	10686	0.548	48.094	7173	0.039	1
star2	21.382	6606	0.454	105.897	4750	0.045	256
netscience	0.048	1210	0.346	0.191	899	0.000	4096
email	0.088	878	0.478	0.404	614	0.034	1
delaunay_n10	0.053	928	0.320	0.198	745	0.060	16384
power	0.144	3754	0.704	0.910	2265	0.028	16

Hill Climbing Evaluation Table

The evaluation plotting does not look like we expected. Based on analysis of these diagrams and the trace file data, the reason could be: 1) Results were collected by manipulating the random seed. It did help the algorithm to find optimal solutions or improve the performance, however all results did not show large differences. 2) Only ten runs were performed for each graph, which might not have given enough data for accurate plotting.

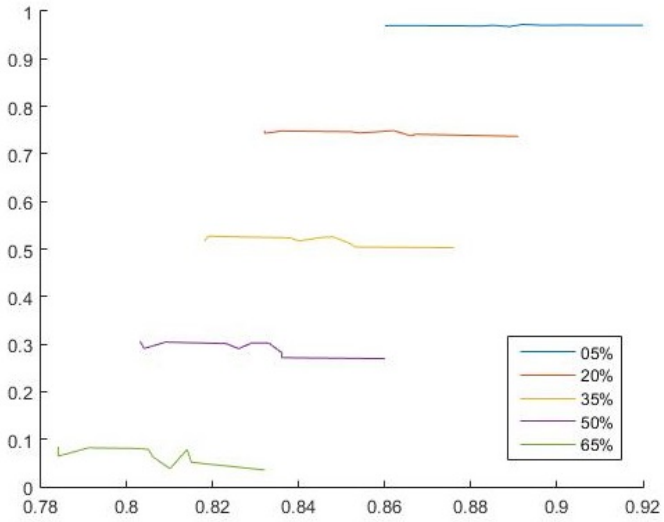


Figure 1. Hill Climbing Power QRTD

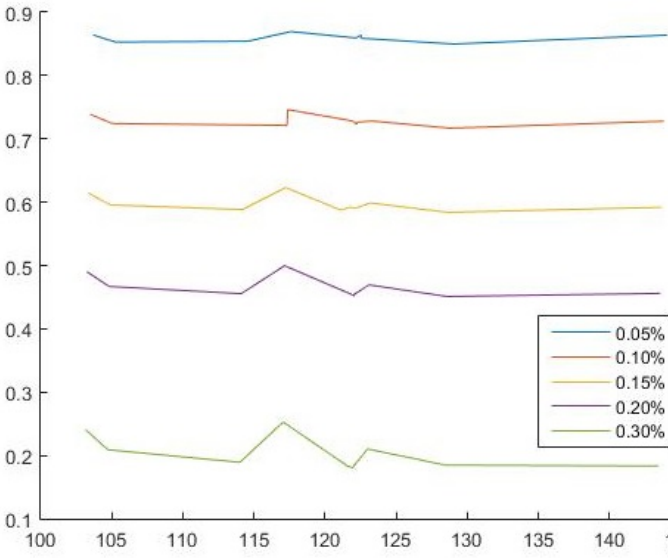


Figure 2. Hill Climbing Star2 QRTD

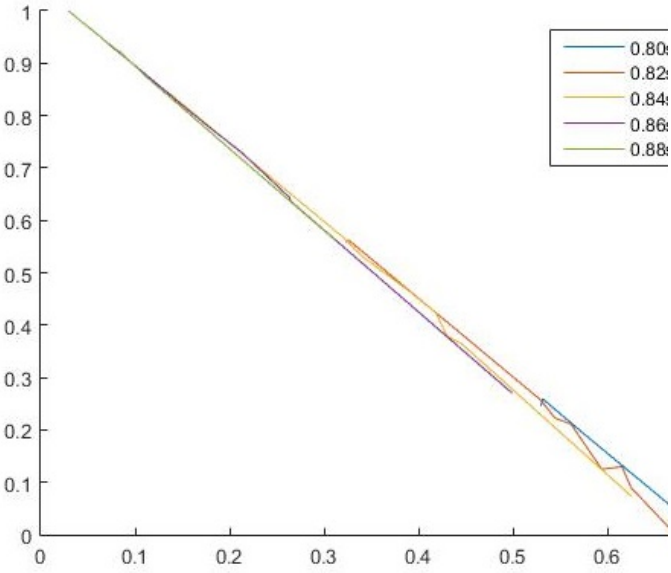


Figure 3. Hill Climbing Power SQD

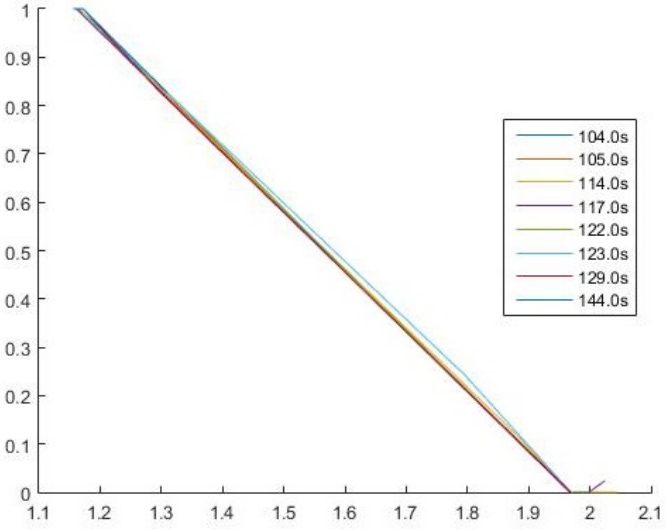


Figure 4. Hill Climbing Star2 SQD

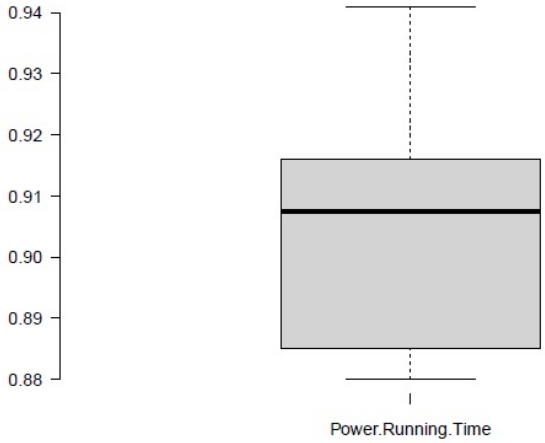


Figure 5. Hill Climbing Power BOX

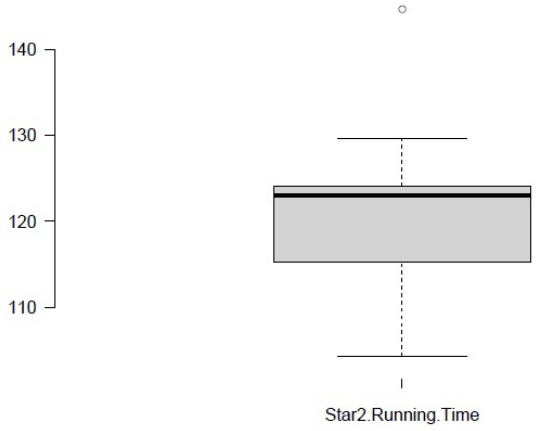


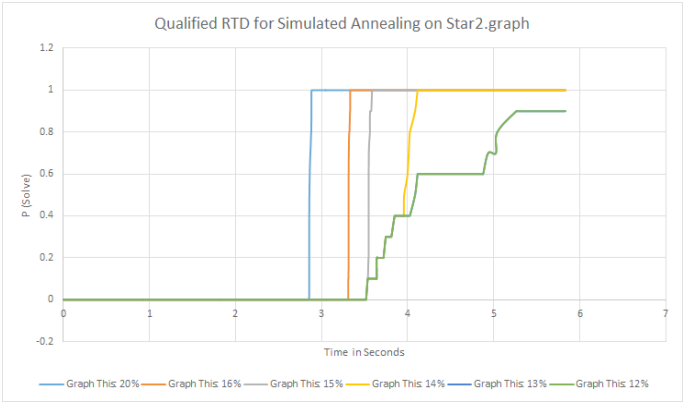
Figure 6. Hill Climbing Star2 BOX

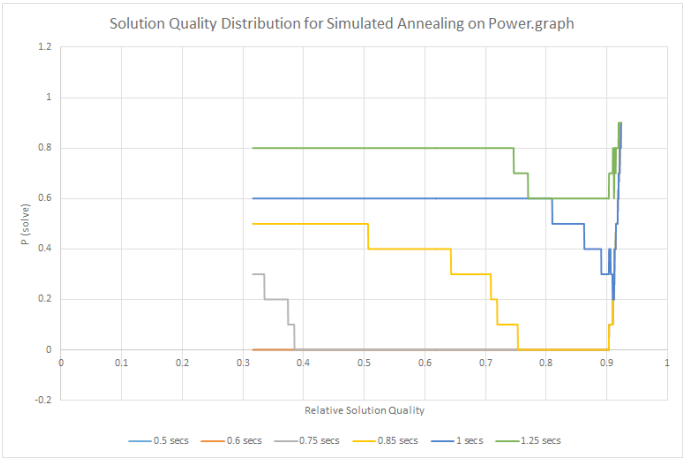
5.4 Local Search – Simulated Annealing

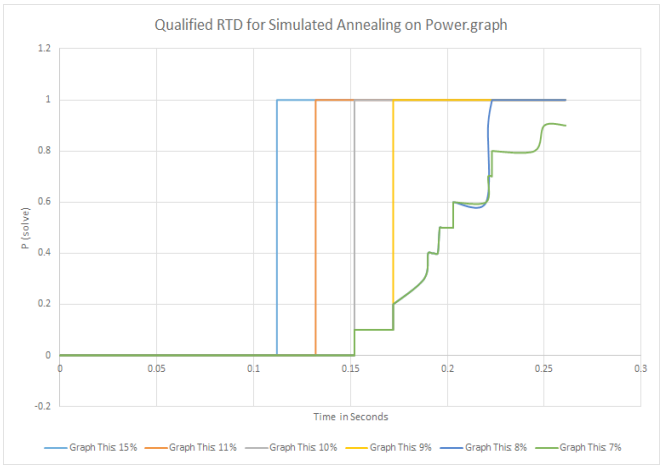
- Platform:
  - CPU: 2.00GHz Intel Core i7
  - RAM: 8 GB
  - Language: Java
  - Compiler: Java Programming Language Compiler (javac)

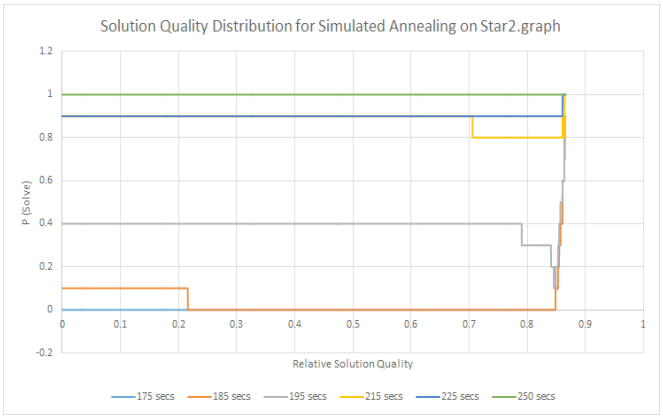
There were different classes made for each algorithm. There is also an executable file that can run any algorithm depending on the inputs.

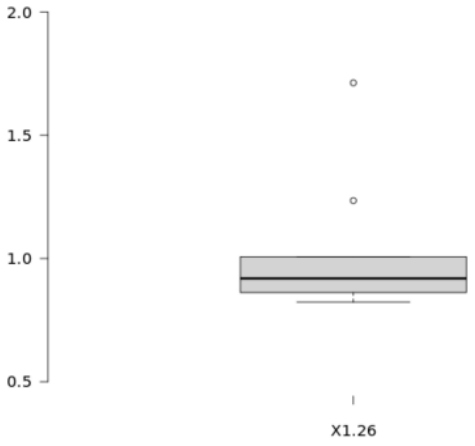
	Local Search (Simulated Annealing) - Averages							
	Phase 1: Max Degree Greedy			Phase 2: Simulated Annealing			Extra Data	
Dataset	Time(s)	VC Value	RelErr	Time(s)	VC Value	RelErr	RandSeed	OPT
jazz	0.0801	190	0.20253	0.0834	172.6	0.09241		
karate	0.0008	20	0.428571	0.0029	14.8	0.057143		
football	0.0053	113	0.202128	0.0085	96.9	0.030851		
as-22july06	27.6278	15115	3.576143	29.8269	3553.9	0.075961		
hep-th	4.1809	8353	1.127611	5.9356	5134.3	0.307769		
star	59.4014	8265	0.197479	66.3909	7338.1	0.063185		
star2	202.7917	13578	1.989432	208.7759	5180.9	0.140665		
netscience	0.1301	1588	0.766407	0.1885	1074.4	0.195106		
email	0.4482	960	0.616162	0.4983	653.9	0.100842		
delaunay_n10	0.1486	1006	0.43101	0.1914	768.2	0.092745		
power	0.7081	3709	0.683613	1.0477	2387.9	0.083931		



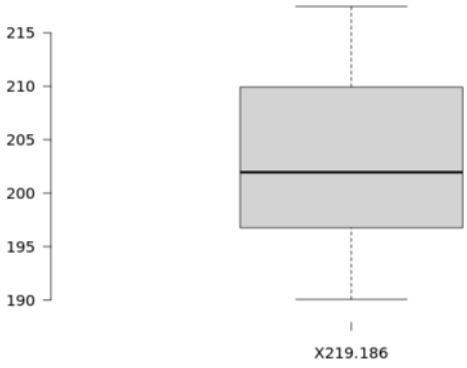








Power Box-plot for Simulated Annealing



Star2 Box-Plot for Simulated Annealing

6. DISCUSSION

Several algorithms were implemented to solve the MVC problem, each with pros and cons. The modified DFS approximation algorithm produces by far the most consistently inaccurate results. However, it runs almost instantaneously for even the largest of the given graphs. In addition, the most inaccurate result of the approximation has a relative error of 42%, which is impressive considering the only guarantee is that relative error will be at most 100%.

Conversely, the branch-and-bound algorithm produced the most accurate results, but at the cost of time efficiency. For the smaller datasets (*karate, football, jazz*), branch-and-bound quickly found an optimal solution. The algorithm successfully returned optimal solutions on some of the medium-sized graphs but failed to finish solving others within the cutoff time. On the larger graphs, branch-and-bound to find any vertex cover, let alone a minimum, within the cutoff time.

The two local search algorithms sacrificed the exactness of branch-and-bound for a significant increase in speed. Its longest runtime was 21.4s (*star2*), and the highest relative error it recorded was 6% (*delaunay_n10*) and it produced two optimal solutions (*karate, netscience*). The simulated annealing method took slightly longer, and was significantly more inaccurate than the hill climbing search. In some cases (*hep-th, netscience*), it resulted in a higher relative error than the approximation algorithm. On the whole, when considering relative accuracy and speed, the hill climbing local search delivered the best, most consistent performance.

7. CONCLUSION

All four algorithms were successfully implemented, and correctly outputted the results. However, among all of the runs for the 11 graphs, very few of them found the optimal solution. Local Search Hill Climbing found optimal solutions for the graphs karate and netscience. Local Search Simulated Annealing found an optimal solution for the graph karate. Branch and Bound found optimal solutions for graphs football and karate, but it failed to find solutions for some graphs. Overall, the Local Search with Hill Climbing was the most efficient, providing high levels of accuracy and speed. Both of the local searches did not plot correctly, but the issue is more obvious for the hill climbing algorithm.

8. REFERENCES

[1] 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.

[2] "Simulated Annealing - Solving the Travelling Salesman Problem (TSP)." - *CodeProject*. Web. 6 Dec. 2015.

[3] Xinshun Xu, Jun Ma, “An efficient simulated annealing algorithm for the minimum vertex cover problem”, *Neurocomputing*, Volume 69, Issues 7–9, March 2006, Pages 913-916, ISSN 0925-2312, http://dx.doi.org/10.1016/j.neucom.2005.12.016.

[4] Guha, Sudipto, Refael Hassin, Samir Khuller, and Einat Or. "Capacitated Vertex Covering." *Journal of Algorithms*: 257-70.

[5] Sangeeta Bansal, Ajay Rana. “Analysis of Various Algorithms to Solve Vertex Cover Problem.” *International Journal of Innovative Technology and Exploring Engineering* (IJITEE) ISSN: 2278-3075, Volume-3, Issue-12, May 2014