Minimum Vertex Cover Algorithms

Nikhita Sagar
Undergrad student of CSE
Georgia Institute of Technology
nsagar3@gatech.edu

Sim Kieu
Master student of CS
Georgia Institute of Technology
simkieu@gatech.edu

Zongwan Cao
Master student of CSE
Georgia Institute of Technology
zongwanc@gmail.com

1. INTRODUCTION

In this report, we would like to introduce the three different methods for finding the minimum vertex cover problem. A minimum vertex cover problem is a problem where given an undirected graph, we want to find the smallest set of vertices such that every edge in the graph has at least one end from that set. This problem is known to be an NP-complete problem, which means it does not have the polynomial time algorithm to solve this problem. Many efforts have been put to solve these kinds of NP-complete problem including Approximation algorithm, Local Search algorithm. Each method has different advantages and disadvantages. The approximation methods like Approximation algorithm and Local Search algorithm can solve the problem faster but will not give the optimal result. The exact algorithm, on the other hand, can give the optimal solution but run really slow and almost impossible to get the result with large graphs.

2. PROBLEM DEFINTION

A vertex cover V' of an undirected graph G with a set of vertices V and a set of edges E is a subset of V such that for all edge (u, v) in E, either u or v must belong to V'. A minimum vertex cover is a vertex cover with smallest possible size, and we need to find such a vertex cover.

3. RELATED WORK

For the approximation algorithm, Chvatal proposed an algorithm to find the minimum vertex cover by selecting a random vertex in the graph, then removing all adjacent edges to this vertex. Then the process continues until all edges are removed [1]. The set of vertices we have selected will be our vertex cover. Then Clarkson improved this method by proposing a method to pick the vertex based on its degree rather than picking a random vertex [2].

Later on, Avis and Imamura invented another approximation algorithm with tighter bound [3]. Let A be the quality of the solution, the bound of their solution is OPT<A<k*OPT, with $\sqrt{(\Delta/2)} < k < \sqrt{(\Delta/2+3/2)}$. In their approach, vertices are sorted in the decreasing order of their degrees (number of edges adjacent to it).

Then Delbot and Laforest improved Avis's method by reducing the worst case factor to Δ. They called their algorithm List Right [4].

Xu and Ma proposed a Simulated Annealing algorithm for finding vertex cover where they introduced a new accepting factor [5].

Asgeirsson and Stein proposed in their paper a divide and conquer method for solving the vertex cover problem. First the graph is divided into two subgraphs, and the recursion function is called on those two subgraphs. Then a combine algorithm is called to find the minimum vertex cover for the combined graph. However, the algorithm they proposed in their paper was not clear as how to divide the graph and they did not even provide the quality bound of the solution [6].

Balaji proposed a new method using a new parameter called support of vertex and his algorithm gives optimal result on most experiments with average running time of O(EV^2). Support of a vertex is the sum of the degrees of all the vertices that are connected to that vertex. He called this VSA algorithm [7]. Imran khan later on devised AVSA which is an improved version of this VSA algorithm and AVSA has been shown to give better result than VSA [8].

4. ALGORITHM

4.1 Branch and Bound algorithm

4.1.1 Algorithm Description

The Branch and Bound algorithm runs on the principle of an exhaustive state space tree search. We evaluate a partial solution at every level of the tree and prune the branches according to our set lower bound and upper bound criteria. In our project we used the 2-approximate vertex cover as our upper bound initially and update it to a valid vertex cover with lesser number of vertices. Our lower bound was the sum of the included vertices and the approximate vertex cover of the remaining graph of undecided vertices. We implemented a depth-first search tree using a stack data structure. The vertices we make a decision on are sorted by their degree (i.e the vertices connected to the maximum number of vertices are decided on first).

Our algorithm pushes a set of the included vertices, remaining vertices not yet visited and a remaining graph without the included vertices. While the stack is not empty, we pop every stack element and branch out into a decision about the next vertex in the list of remaining vertices. If the vertex is not included, then all the neighbours of that vertex and appended to the included vertices list and the neighbours are removed from the list of undecided vertices and the remaining graph.

4.1.2 Pseudocode

```
1. BnB(G):
    a. bestsolution_sofar = approxVC(G)
    b. V_notdecided = vertices in G sorted by degree (descending)
    c. V_included = [] G_remaining = G
    d. stack.push(V_included, V_notdecided, G_remaining)
    e. While (stack is not empty & elapsed time < cutoff):
        i. included, notdecided, G_rem = stack.pop()
    // DFS traversal
        ii. if no edges are remaining in G_rem:
            1. if number of included vertices < bestsolution_sofar:
                a. bestsolution_sofar = included vertices
                b. VC = included vertices
        iii. else:
```

1. v = notdecided.pop()
2. if included + approxVC(G_rem) <= bestsolution_sofar:
a. stack.push(included, notdecided, G_rem) //not include
b. G_rem.remove(v)
c. included.add(v)
d. stack.push(included, notdecided, G_rem) // include vertex

f. return VC

We tried using recursion instead of a physical stack. This lead to memory errors because a recursive stack uses a lot more memory than a stack data structure. Additionally we used a depth-first-search method instead of breadth-first-search because with a level search we would have taken even longer to reach a leaf node.

4.1.3 Complexity

The algorithm has an O(2^n) time complexity. This is because it branches into a binary tree.

Space complexity: The algorithm uses O(V) space to store the vertices, O(V+E) to store the graph and O(V+E) for the stack.

4.1.4 Results

Table 1. Table result of the branch and bound

Graph	Time (s)	VC Value	OPT	Relative Error
karate	600	14	14	0.0000
football	600	98	94	0.0425
jazz	600	171	158	0.0822
delaunay_10	600	925	703	0.3157
email	600	900	594	0.5151
netscience	600	1438	899	0.5995
power	600	3792	2203	0.7212
hep-th	600	5800	3926	0.4773
star	600	10514	6902	0.5233
star2	600	6838	4542	0.5055
as-22july06	600	6052	3303	0.8322

4.2 Approximation algorithm

4.2.1 Algorithm Description

Our approximation algorithm does not alter the graph. It calculates the vertex cover by changing the 'visited' attribute of every node from true to false. We used the built-in Networkx library in python to create the graph using the graph data structure and add the 'visited' attribute to the nodes.

Initially we set the 'visited' attribute of all vertices to false. Next, we only pick an edge to include if both its endpoints are set to false. Everytime an edge is picked in the algorithm, the algorithm changes the 'visited' attribute of the vertices to true and includes them in the vertex cover. The algorithm thus produces a 2-Approximate vertex cover.

4.2.2 Algorithm Description

Approx_VC(G):

a. VC←∅

b. while E != ∅

pick any {u, v} ∈ E

VC ← VC ∪ {u, v}

delete all edges incident to either u or v

return VC

4.2.3 Complexity

This algorithm is a polynomial time 2-appoximate algorithm. The returned set VC is a vertex cover. If A is the set of edges picked by the algorithm and OPT is the optimal vertex cover.

- OPT must include at least endpoint from each edge in A.
- Since edges in A do not share endpoints (because of the algorithm design) no 2 edges in A are covered by the same vertex in OPT.
- Thus: |OPT| >= |A| also |VC| = 2|A| (Since the number of vertices are twice the number of selected edges in A)
- |VC| = 2|A| <= 2|OPT|
- |VC| <= 2|OPT|

3. Runtime: The runtime of this algorithm is O(V+E)

It takes O(V) time to create the graph from the file

It takes O(V+E) time to traverse the graph and pick all the edges

4. Space Complexity:

The graph is stored in an adjacency list in O(V+E) space

The vertex cover is stored in O(V) space

Overall space complexity is O(V+E)

4.2.4 Results

Table 2. Table result of the approximation algorithm

Graph	Time (s)	VC Value	OPT	Relative Error
karate	0.32	20	14	0.43
football	2.74	108	94	0.15
jazz	15.62	190	158	0.20
delaunay_10	26.84	966	703	0.37
email	47.33	838	594	0.41
netscience	15.83	1226	899	0.36
power	0.05	3792	2203	0.72
hep-th	141.94	5800	3926	0.48
star	0.19	10514	6902	0.52
star2	0.16	6838	4542	0.51
as-22july06	186.37	6052	3303	0.83

*Note: The running time for bigger graphs is sometimes faster than for the smaller ones. It was because the state of the machine when the graphs were run.

4.3 Local Search algorithm

4.3.1 Algorithm Description

Local search is a metaheuristic method for solving computationally hard optimization problems. It can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions, thus we can apply it on our vertex cover problem. The basic idea of local search is moving from solution to solution in the space of candidate solution by applying local search, until a solution deemed optimal is found or a time bound is elapsed.

To generate a proper selection scheme, we define two scoring properties for each vertex v :1) loss(v) shows the number of covered edges that would become uncovered by removing v out of current solution C; 2) gain(v) is the number of uncovered edges that would become covered by adding v into current solution C.

1. Construct an initial vertex cover
   For each edge e in E, if it is uncovered, we add the endpoint of e with higher degree into C. Now C must be a valid vertex cover, we calculate loss function loss(v) for each vertex v in our solution C : if v's adjacent vertex is not in C, then loss(v)++. Then we shrink our initial vertex cover to make its size closer to minimum vertex cover by checking loss(v) for each v in C. If loss(v) equals to zero, v will be removed from our initial solution C.
2. Remove vertex from current vertex cover
   Starting from current solution C, we keep removing a vertex with minimum loss from C until C is no longer a valid vertex cover
3. Select exiting and entering vertex
   We implement two different selection scheme here
   a. Local Search A : The first one chooses a vertex pair randomly from candidate vertex pairs, exchanges them and update loss and gain functions. We sample k (k is set as 20 in our implementation) times from candidate vertex pairs, so we have k potential exiting vertices. Among them, the vertex with minimum loss is our final exiting vertex. Correspondingly, entering vertex is its peer.

Local Search B: The second one is called two-steps exchange. It first chooses a vertex in vertex cover and removes it, and updates the loss and gain functions. Then it picks up a vertex outside cover and adds, and again updates loss and gain functions. Basically, k (k is set as 20 in our implementation) vertex are chosen from C randomly and the exiting vertex is the one with minimum loss.

To obtain the entering vertex, a random uncovered edge e is checked. We put the entering vertex as the endpoint of e with greater gain breaking ties with choosing one of those two randomly.

Now by removing exiting vertex and adding entering vertex, we generate a new vertex set. If the new set is a valid vertex cover: update our current solution C as this new set and back to step (2). If the new set is not a valid vertex cover: back to step (3) to find a new pair of exiting-entering vertex.

4.3.2 Pseudocode

1. Local search 1
   Here we implemented Local search A with one main function and five other functions
   a. LocalSearchA(G, cutoff, random seed) : takes graph G, cutoff time and random seed to return final vertex cover of G

```
Input: graph G = (V,E), cutoff time, random seed
Output: vertex cover of G
(c, loss) := ConstructVC(graph)
while time < cutoff do
        if IsVertexCover(c) then
                opt = c
                pick up a node  in c with min loss as rm
                UpdateRm(c,rm)
                continue
        u:=ChooseRmVertex(c)
        UpdateRm(c,u)
        Pick up an uncovered adjacent node of u as v randomly
        UpdateAdd(c,v)
```

   b. ConstructVC(graph): takes graph G to construct an initialized vertex cover

```
Input: graph G = (V,E)
Output: vertex cover of G, loss function of nodes in the vertex cover
c:= empty
foreach e in E do
        if e is uncovered then
                add the endpoint of e with higher degree into c
foreach v in C do
        loss(v):=0
foreach e in E do
        if only one endpoint of e belongs to c then
                for the endpoint v in c, loss(v)++
foreach v in C do
        if loss(v) =0 then
                UpdateRm(c,v)
return c, loss
```

   c. IsVertexCover(c): takes current solution to check if it is a valid vertex cover

```
Input: current solution c
Output: true if c is a vertex cover, false if c is not a vertex cover
foreach v in V do
        if v not in c then
                foreach u in v's adjacent nodes do
                        if u is not in c then
                                return false
return true
```

   d. UpdateRm(c,rm): updates loss function, uncovered edge and solution c after removing rm from c

Input: current solution c, vertex rm need to be removed
Output: updated solution c, loss function loss, uncovered edges uncovered
remove rm from c
remove rm from loss
foreach v in both rm's adjacent nodes and c do
loss(v)++
if there is some v in rm's adjacent nodes but not in c then
foreach v in rm's adjacent nodes but not in c do
make uncovered cover (rm,v)

e. UpdateAdd(c,ad): updates loss function, uncovered edge and solution c after adding ad into c
Input: current solution c, vertex ad need to be added
Output: updated solution c, loss function loss, uncovered edges uncovered
add ad to c
foreach v in both rm's adjacent nodes and c do
loss(v)--
if there is some v in rm's adjacent nodes but not in c then
foreach v in rm's adjacent nodes but not in c do
remove edge (rm,v) from uncovered

f. ChooseRmVertex(c): picks up a vertex from c randomly
Input: current solution c
Output: vertex rm
Select k nodes randomly
rm:= vertex in k with min loss

2. Local search 2
The difference between Local search A and B is the neigborhood strategy. So we still have one main function and five other functions. Except for main function, the remaining five are the same as Local search A

a. LocalSearchB(G, cutoff, random seed) : takes graph G, cutoff time and random seed to return final vertex cover of G
Input: graph G = (V,E), cutoff time, random seed
Output: vertex cover of G
(c, loss) := ConstructVC(graph)
while time < cutoff do
if IsVertexCover(c) then
opt = c
pick up a node in c with min loss as rm
UpdateRm(c,rm)
continue
u:=ChooseRmVertex(c)
UpdateRm(c,u)
e:=a random uncovered edge
v:=the endpoint of e with greater gain, breaking ties by randomly choosing
UpdateAdd(c,v)

4.3.3 Complexity

For time: Since
(1) UpdateRm(c,v): For each adjacent node of v, find which are inside c and which are outside c. This is O(|c|*deg|v|)
(2) UpdateAdd(c,v): same as UpdateAdd O(|c|*deg|v|)
(3) ConstructVC : Visit each edge to see if it is covered, so the time is O(|E|)
(4) IsVertexCover(c): Check two endpoints of each edge to see if at least one of them is in current solution c. O(|c|*|E|)
Basically, LS's overall time complexity is hard to analyze theoretically since it is randomized. Here we use heuristic ways to evaluate its time complexity. Our experiments on sample graphs show it is a fast and efficient method to solve Vertex Cover problem.
For space, we have four lists:

1. c = list() current solution may be a valid vertex cover, may be not O(|V|)
2. opt = list() current optimal vertex cover O(|V|)
3. loss = dict() key: vertex in current solution; value: corresponding loss value for each key O(|V|*|V|)
4. uncover = dict() key: vertex not in current solution; value: adjacent nodes which are also not in current solution O(|V|*|V|)

Overall Space Complexity O(|V|*|V|).

4.3.4 Results

1. Comprehensive Table
cutoff time = 900s. For those graph that has already arrived at optimal solution at some time t before cutoff time, we record the time t and cutoff it manually.
Here we can see both LS1 and LS2 perform quite well to arrive at optimal. For small graphs, we can obtain optimal within several seconds and medium graphs need 1-2 mins. For those large graphs (thousands of vertices), with our cutoff time, though they do not get their optimal, their RelErr values are pretty small.

Table 3. Table result of the Local Search 1 algorithm

Graph	Time (s)	VC/OPT Value	Initial Size	Relative Error
karate	0.01	14/14	14	0.0000
football	0.38	94/94	96	0.0000
jazz	0.36	158/158	160	0.0000
delaunay_10	900.00	713/704	747	0.0128
email	900.00	597/594	615	0.0051
netscience	0.14	899/899	899	0.0000
power	900.00	2247/2203	2276	0.0200
hep-th	900.00	3945/3926	3945	0.0048
star	900.00	7049/6902	7209	0.0213
star2	900.00	4796/4542	4868	0.0559
as-22july06	900.00	3325/3303	3325	0.0067

Table 4. Table result of the Local Search 2 algorithm

Graph	Time (s)	VC/OPT Value	Initial Size	Relative Error

karate	0.00	14/14	14	0.0000
football	0.02	94/94	96	0.0000
jazz	0.22	158/158	160	0.0000
delaunay_10	900.00	704/704	747	0.0000
email	900.00	594/594	615	0.0000
netscience	0.15	899/899	899	0.0000
power	900.00	2209/2203	2276	0.0027
hep-th	900.00	3938/3926	3945	0.0030
star	900.00	7043/6902	7209	0.0204
star2	900.00	4783/542	4868	0.0530
as-22july06	900.00	3314/3303	3325	0.0033

For graph power.graph and star2.graph, we ran each of them 10 times with different seeds(seeds are 1,2,4,8,16,32,64,128,256,512) and got the following three estimations. When we did LS1 for power.graph, we found it got stuck at some time stamps and never move to any neighbor any more. Different seeds have different stuck time points and most of them are around 20 seconds. From QRTD and SQD, we can see LS2 is more stable than LS1. Since in LS1 exiting node and entering node must be on the same edge, it is likely that at some point, we cannot find any node pair satisfied this requirement. So our local search will stop at that point. In this view, we can conclude LS2 performs better than LS1.

2. Qualified Runtime for various solution qualities (QRTDs)

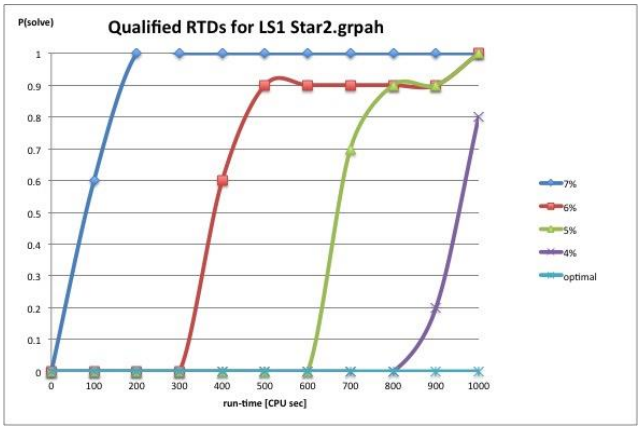

Figure 1. Qualified RTDs for LS1 Star2.graph.

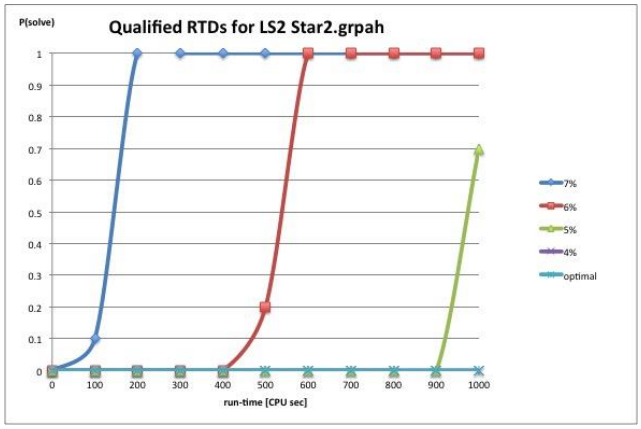

Figure 2. Qualified RTDs for LS2 Star2.graph.

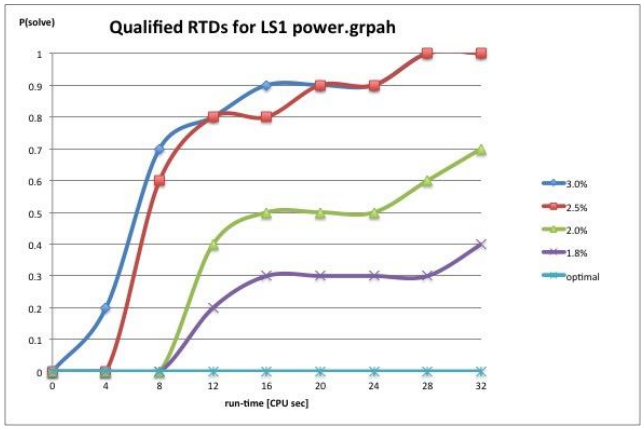

Figure 3. Qualified RTDs for LS1 power.graph.

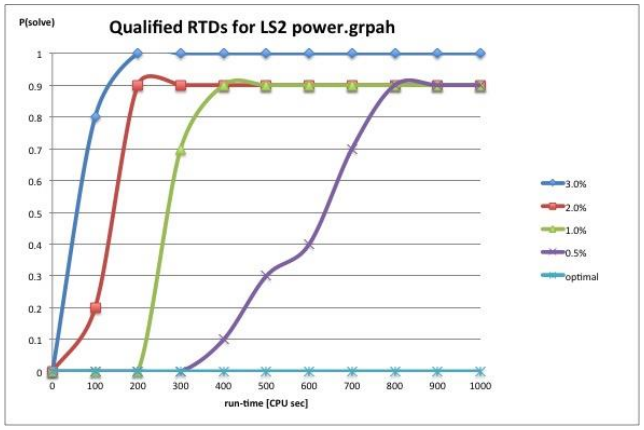

Figure 4. Qualified RTDs for LS2 power.graph.

3. Solution Quality Distributions for various run-times (SQDs):

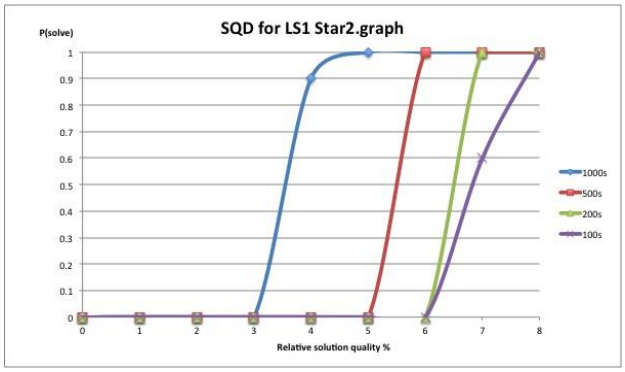

Figure 5. Solution Quality Distribution (SQD) for LS1 Star2.graph.

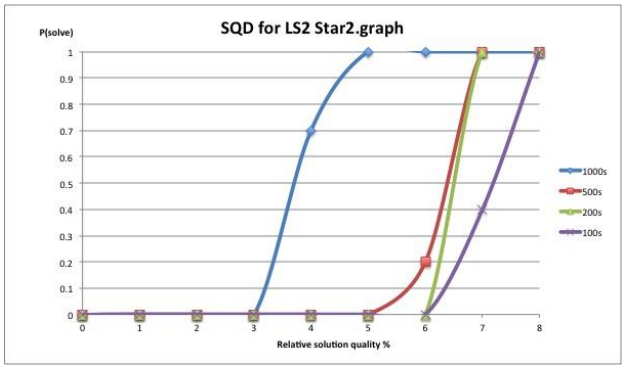

Figure 6. Solution Quality Distribution (SQD) for LS2 Star2.graph.

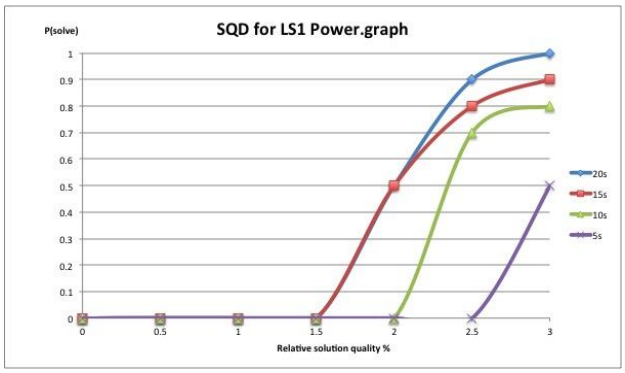

Figure 7. Solution Quality Distribution (SQD) for LS1 power.graph.

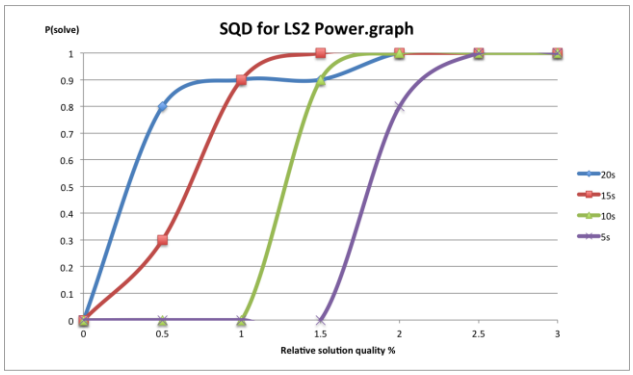

Figure 8. Solution Quality Distribution (SQD) for LS2 power.graph.

4. Box plots for running times:

Center lines show the medians; box limits indicate the 25th and 75th percentiles as determined by R software; whiskers extend 1.5 times the interquartile range from the 25th and 75th percentiles, outliers are represented by dots. n = 9, 10, 10, 10 sample points.

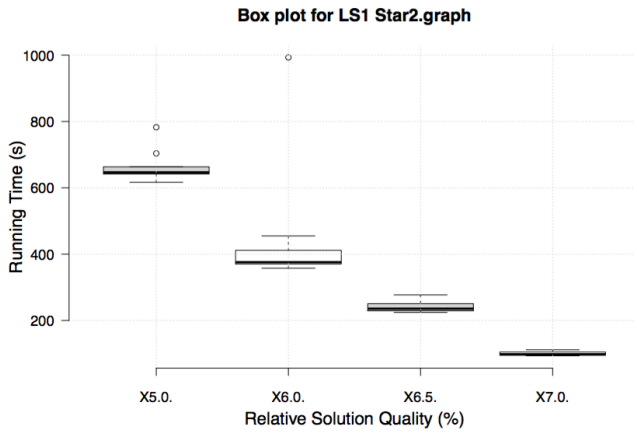

Figure 9. Box plot for LS1 Star2.graph.

n = 7, 10, 10, 10 sample points.

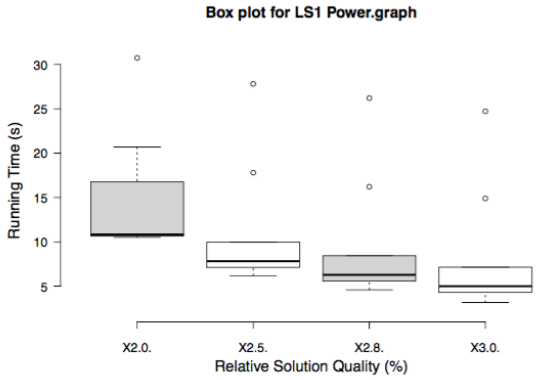

Figure 10. Box plot for LS1 power.graph.

n = 7, 10, 10, 10 sample points.

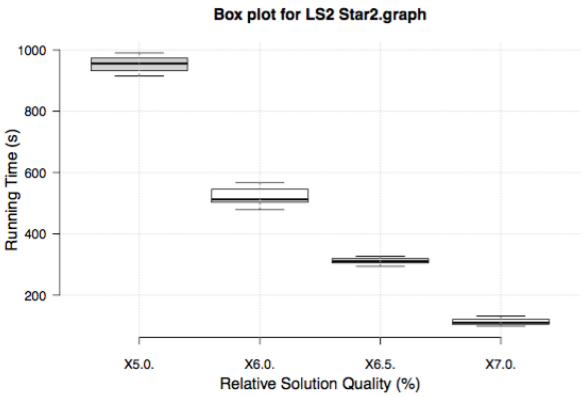

Figure 11. Box plot for LS2 Star2.graph.

n = 9, 9, 10, 10 sample points.

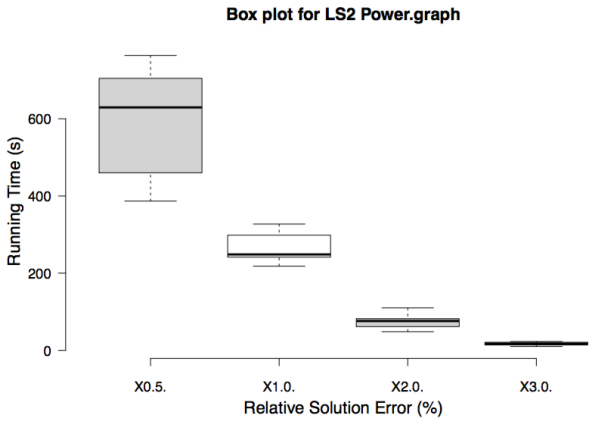

Figure 12. Box plot for LS2 power.graph.

5. EMPIRICAL EVALUATION

Comparison of different algorithms:

We used Python to code our algorithms. We realized later in the project that algorithms written in C++ ran much faster than those written in Python and thus produced better results for the Branch and Bound algorithm. Even though we used a machine with 8GB RAM to run our code, we realized that Python only allotted 2GB RAM to run code. Thus we kept getting memory errors for bigger graphs while running Branch and Bound. Additionally, we used the Sublime Text Editor as an IDE. This text editor uses to Cython compiler to compile. We used the built-in NetworkX library in Python to create our graphs from the given data files. We used the built-in graph data structure to store the graph and add additional attributes to the nodes.

We evaluated our algorithms on the basis on relative error and time. The relative error plot in the figure below shows that the local search algorithms produced the solutions closest to the optimal solution. The branch and bound algorithm worked well for smaller graphs, however, due to the exponential time complexity, didn't run well for larger graphs.

Our lower bound on the optimal solutions quality was not a very tight bound. Both the Branch and Bound and Approximation algorithms resulted in maximum 70% relative error.

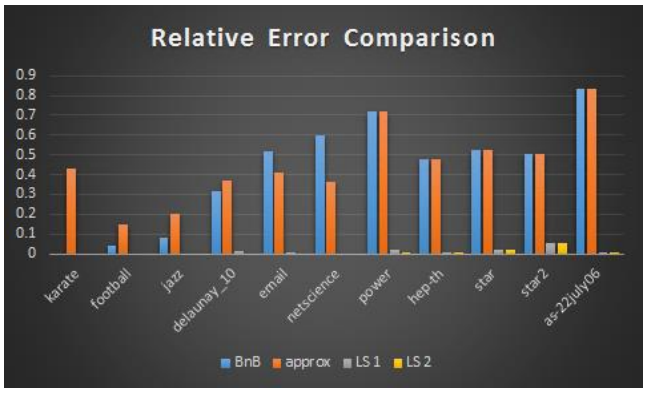

Figure 13. Insert caption to place caption below figure.

6. DISCUSSION

The approximation algorithm is the fastest among the algorithm being analyzed in this report. Its running time complexity is linear to the number of edges. The slowest algorithm is the branch and bound algorithm with its worst case time complexity is still exponential. There's an advantage and disadvantage between these two algorithms. The branch and bound trade its slow running time to guarantee to find the optimal solution whereas the approximation is much faster but most likely will not give the optimal solution. Local search can be considered between the spectrum of these two algorithms in terms of solution quality and running time. The running time of Local Search is considerably much faster than the branch and bound algorithm and gives relatively better result than the approximation algorithm, especially with random restart (in simulated annealing).

7. CONCLUSION

Through this project, we have learned to solve one of the NP-complete problems using different approaches. NP-complete problems are intractable and hard to solve but we see them a lot in our daily life's optimization problems. Being able to solve them is definitely a good skill to learn and we can see that many researchers and scientists have spent a lot of time and effort to come up with better ways to solve these types of problems. In this project, we have learned three different methods with their unique properties. Branch and bound is an exact algorithm, which always gives optimal result but extremely slow although it is faster than the naive brute-force. Approximation is an algorithm which is fast and can guarantee to give not too bad result. Local search is a fast algorithm with no guarantee in the quality of the result but often give very good result, and if lucky can lead to optimal solution. Through this project, we know the advantages and disadvantages of each algorithm and know when to use it based on the condition and the requirement. Apart from learning the algorithms, we also learn how to develop our thoughts and discuss it with teammates. This project helped us feel more confident in our problem solving skills and teamwork as well as gave us the opportunity to challenge our thinking ability.

8. REFERENCES

[1] Chvatal, V. 1979. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*. 4 (1979).

[2] Clarkson, K. 1983. A modification to the greedy algorithm for vertex cover problem. *Information Processing Letters.* 16 (1983).

[3] Avis, D. and Imamura, T. 2007. A List Heuristic for Vertex Cover. *Operations research letters.* 35 (2007).

[4] Delbot, F. and Laforest, C. 2008. A better list heuristic for vertex covers. *Information Processing Letters*. 107 (2008).

[5] Xu, X. and Ma, J. 2006. An efficient simulated annealing algorithm for the minimum vertex cover problem. *Neurocomputing*. 69, 7 (2006), 913-916.

[6] Asgeirsson, E. I. and Stein, C. 2009. Divide-and-Conquer Approximation Algorithm for Vertex Cover. *SIAM Journal on Discrete Mathematics.* 23, 3 (2009), 1261-1280.

[7] Balaji, S., Swaminathan, V., and Kannan, K. 2010. Optimization of Un-weighted Minimum Vertex Cover. *World Academy of Science, Engineering and Technology*. 67 (2010).

[8] Khan, I., Ahmad, I., and Khan, M. 2014. AVSA, Modified Vertex Support Algorithm for Approximation of MVC. *International Journal of Advanced Science and Technology*. 67 (2014), 71-78.