CSE 6140 Final Project Report

Tiancheng Ye
Department of Computer Science and Engineering
Georgia Institute of Technology
tye97@gatech.edu

Chenying Liu
Department of Computer Science and Engineering
Georgia Institute of Technology
cliu662@gatech.edu

Kai Qu
Department of Computer Science and Engineering
Georgia Institute of Technology
kqu30@gatech.edu

1 Introduction

The minimum vertex cover (MVC) problem is a well-known NP-Hard problem in graph theory. The problem is to find a minimum subset of vertices such that every edge in the original graph has at least one endpoint in the subset of vertex cover. In this project, we develop 4 algorithms to solve the MVC problem. The branch and bound algorithm finds the exact solution of the minimum vertex cover by recursively examining each branch of the search tree. The approximate algorithm finds an approximate solution with a fixed approximate ratio. We also develop two local search algorithms based on the modification of hill-climbing algorithm. The results show that the performance of approximate algorithm is desired given the stringent computational time constraints and large solution quality tolerance, while the branch and bound algorithm takes too long to find the exact solution and the solution quality of local search algorithms strongly depends on the randomization.

2 Problem Definition

In graph theory, a vertex cover is a set of vertices which includes at least one endpoint of every single edge of the graph. Formally, given an undirected graph $G = (V, E)$, a vertex cover of G is the subset $V' \subseteq V$ such that for every edge $(i, j) \in E$, either $i \in V'$ or $j \in V'$. There are two versions of vertex cover problem. The decision version determines if there exists a vertex cover with size smaller than or equal to a given integer k for an undirected graph, which is an NP-Complete problem. The optimization version, which is NP-hard, finds the vertex cover with the minimum number of vertices.

3 Related Work

The vertex cover problem has received lots of attentions in the past few years and many algorithms have been proposed to solve this problem. However, due to its NP-Completeness, there is no efficient algorithm to find the exact solution in polynomial time. Hence, many algorithms solve the problem approximately with or without optimality guaranteed. For example, the Maximum Degree Greedy (MDG) algorithm, which is adapted from the classical greedy algorithm [2], solves the vertex cover problem with an approximate ratio of $H(\Delta) = 1 + \frac{1}{2} + \cdots + \frac{1}{\Delta}$, where Δ is the maximum degree of the graph. This algorithm initializes an empty set of vertex cover and adds the next vertex with maximum degree step by step until the edges are empty. Another algorithm which is derived based on the greedy algorithm for independent set problem by [5], is known as The Greedy Independent Cover (GIC) algorithm. GIC is very similar to MDG except that it adds the neighboring nodes of the vertex with the minimum degree. However, GIC does not have an upper bound for the worst-case approximate ratio. In other words, its approximate ratio is at least $\frac{\sqrt{\Delta}}{2}$. [6] proposed another algorithm using Depth-First-Search (DFS) which returns a connected vertex cover from the non-leaf nodes of a DFS spanning tree. [6]'s algorithm has a fixed approximate ratio of 2 independent of the structure and size of the graph. Another algorithm with an approximate ratio of 2 is the Edge Deletion algorithm developed by [4], which is also the algorithm presented in the lecture notes. The ListLeft algorithm by [1] and ListRight algorithm by [3] are another type of algorithms which combines approximation algorithms with list heuristics. In both algorithms, the vertices are scanned one by one in a fixed sequence and include or exclude current scanned vertex based on predefined criteria. The ListLeft algorithm has an approximate ratio of $\frac{\sqrt{\Delta}}{2} + \frac{3}{2}$ while the ListRight algorithm has a approximate ratio of Δ. However, ListRight usually returns a solution that is smaller than ListLeft algorithm.

4 Input & Output

To read the graph instances, we define a *RunExperiment* class which has a *read_graph* method. The argument of this method is the filename of the graph instance. Inside this method, an empty NetworkX graph is initialized. After opening the file, each line is split, mapped to the *int* type, and added as edges to the currently processing node. For some nodes, there are no edges incident to them. In this case, we only record the nodes in the graph. The output of this method is a NetworkX graph with all the information of node and edge read from the input file.

5 Algorithms

5.1 Approximation Algorithm

The approximate algorithm outlined in the lecture notes is essentially the Edge Deletion algorithm by [4]. One limitation with this algorithm is that it randomly removes the endpoints of next edge without any selection criteria. Hence for very large graph, this algorithm may start to remove

the edge with low degree which reduce the computational speed. Another previously mentioned algorithm (i.e. MDG) removes the next vertex with the maximum degree. However, MGD only considers a single vertex each time hence may not represent the connectivity of the edge. In this project, we use a greedy heuristic that combines both the Egde Deletion and the MDG algorithms. This novel algorithm first determines the edge with the largest sum of degrees of both end nodes. Then, it removes the associated nodes and all their incident edges until the edge set is empty. In this pattern, we do not choose edges arbitrarily but remove edges whose nodes have higher coverage of edges first, which could speed up the algorithm for graphs with many high-connectivity edges. The pseudo code for this algorithm is presented below:

Algorithm 1: Approximate algorithm to find the minimum vertex cover

```
Data: a graph G(V, E)
Result: a vertex cover C
Initialize C = ∅;
while E ≠ ∅ do
    select an edge (i, j) with maximum sum of
      degrees for node i and j;
    V = V − {i, j};
    C = C ∪ {i, j};
end
return C
```

Since the algorithm is a combination of both MDG and Edge Deletion, it is straightforward to derive that the worst-case approximate ratio of this algorithm is minimum of the MDG and Edge Deletion, in other words, $\min(H(\Delta), 2)$, where $H(\cdot)$ is the harmonic series and Δ is the maximum degree of the graph. The algorithm will run number of edges times and each time we need to compute the degree of each vertex and find the maximum edge, hence the time complexity is $O(EV^2 \log(V))$ and the space complexity is $O(V + E)$.

One potential drawback of this algorithm is that if lots of the edges in a graph shares similar degrees, then this algorithm will give the similar result as the regular Edge Deletion algorithm but the time complexity is much higher.

5.2 Branch & Bound Algorithm

This implementation of branch & bound algorithm is a modification to a recursion in which a recursion represents a branch in the search tree, and this branch will be pruned if its lower bound is larger than the best upper bound recorded. When deciding how to expand the subproblem, we apply a simple greedy heuristic that chooses the node with the maximal degrees in the current graph first. The intuition of this heuristic lies in the belief that nodes with larger degrees have higher potential to cover more edges.

For Lower bound calculation, assuming the current graph has edges E, and the number of maximal degree is denoted as n, we use the following formula to calculate the lower bound: $l = \lceil \frac{|E|}{n} \rceil$. This is a valid lower because suppose we have an optimal vertex cover O, then it follows that $|O|n \geq |E|$, which is equivalent to $|O| \geq |E|/n$. Thus, the $\lceil |E|/n \rceil$ is a lower bound for the current graph.

The pseudocode of branch & bound algorithm is presented below.

Algorithm 2: Branch & Bound Algorithm

```
Data: a graph G(V, E)
Result: a vertex cover C
Initialize C = ∅;
start_nodes = sort(V) by the decreasing order of
  node degrees;
for node ∈ start_nodes do
    graph_copy = deepcopy(G);
    recurse(graph_copy, node, C, [])
end
return C;
```

Algorithm 3: lower_bound: the helper function that calculates the lower bound of a given graph

```
Data: a graph G(V, E)
Result: the calculated lower bound for G
n = len(G.nodes);
max_degree = the max node degree of G;
return max_degree/n;
```

Algorithm 4: recurse: the helper function that recursively explore the search space and conduct pruning

```
Data: a partial graph G′(V, E)
Data: current node n
Data: The current best vertex cover best_solution
Data: The current partial solution s
Result: None, as this function only updates
        best_solution
if G′ is empty then
    if best_solution is empty or
      len(s) < len(best_solution) then
        update the best solution;
    end
end
lower_bound = lower_bound(G′);
if lower_bound > len(best_solution) then
    return;
end
candidates = sort(G′.nodes) by the decreasing order
  of node degrees;
for c ∈ candidates do
    copy_graph = deepcopy(G′);
    recurse(copy_graph, c, best_solution, s + [c]);
end
```

For time complexity, since there are $2^{|V|}$ possible states in the search space, and for each search space, we have to sort the nodes in the graph by the decreasing order of node degrees to calculate the lower bound and apply the greedy heuristics. Thus, the overall time complexity is $O(|V|log(|V|)2^{|V|})$. The space complexity is $O((|V|+|E|)2^{|V|})$ because there are $2^{|V|}$ possible recursions and in each recursion we have to copy the current graph, which takes $O(|V|+|E|)$.

5.3 Local Search Algorithm 1

This local search is a hill-climbing algorithm. In each iteration, we randomly remove 5 points and try to find a few other points to replace them. The metrics of this algorithm is whether the amount of replacing points is below 5 and whether new vertex cover is valid. The pseudo code for this algorithm is presented below:

Algorithm 5: Local Search algorithm 1

```
Initial valid vertex cover: output → {x_1, x_2, ..., x_n}
while elapsed time<cutoff do
    Randomly remove 5 vertex from output:
     R → {A_1, A_2, ..., A_5}
    Find the vertexes which connect with the R but is
     not included in the output:
     S → {B_1, B_2, ..., B_n} where B_i ∉ output \ R and
     exist (B_i, A_k) ∈ edges where k ∈ {1, 2, 3, 4, 5}
    if |S| < 5 then
        output → output \ R
        output → output ∪ S
    end
end
return output
```

Since this algorithm only has one loop and restricted by the cutoff, the time complexity will be the $O(n)$. Meanwhile, we don't need to record any intermediate result, so the space complexity will be $O(1)$.

5.4 Local Search Algorithm 2

This local search algorithm is a hill climbing algorithm. At first, we try to remove every vertex and check whether the vertex cover is still valid. If this strategy not works, we try to randomly replace one vertex to another until the vertex cover is valid. A threshold is set in this step since there may be a local optimal. Hence, we will randomly replace two vertexes to another if the threshold is achieved. Finally, we can go back to first step to check every vertex again. The pseudo code for this algorithm is presented below:

Algorithm 6: Local Search algorithm 2

```
Initial valid vertex cover: output → {x_1, x_2, ..., x_n}
while elapsed time<cutoff do
    flag = False
    for each in output do
        if output − each is vertex cover then
            output = output − each
            flag = True
            break
        end
    end
    if flag=True then
        continue
    end
    if elapsed time<threshold then
        output → exchange one vertex
        if output is vertex cover then
            break
        end
        else
        output → exchange two vertexes
        reset elapsed time=0
        if output is vertex cover then
            break
        end
    end
end
return output
```

Since this algorithm is restricted by the out loop, the time complexity will be the $O(n)$. Meanwhile, we don't need to record any intermediate result, so the space complexity will be $O(1)$.

6 Empirical Evaluation

The experimentation can be summarized into two subparts. The first one is the lateral comparison of performances among the four algorithms across all graph instances. The time limit for each graph instance is 10 minutes, and the random seed set for local search algorithm 1 and local search algorithm 2 is 42. As Branch & Bound Algorithm and Approximation algorithm implemented do not involve stochasticity, the next experiment focus on the comparative evaluation between Local Search 1 and Local Search 2 on two large graph instances power and star2, where we perform 20 independent runs for each cut-off run time and solution quality. The processors and platform for running the algorithms are 2.9 GHz Quad-Core Intel i7 and MacOS Catalina 10.15.3 respectively.

6.1 Approximation Algorithm

The comprehensive table, presented as table 1, includes four fields, where Instance Name refers is the name of the graph instance, Running Time is the time for a certain algorithm to

Comprehensive table with Approximation Algorithm			
Instance Name	Running Time (s)	Vertex Cover Size	Relative Error
karate	0.001	16	0.14
football	0.03	102	0.09
jazz	0.16	174	0.10
email	1.31	728	0.23
delaunay_n10	1.25	858	0.22
netscience	1.16	1144	0.27
power	12.20	3112	0.41
as-22july06	63.42	4956	0.5
hep-th	33.55	5200	0.33
star	139.91	10096	0.46
star2	85.34	5900	0.31
Comprehensive table with Branch & Bound Algorithm			
karate	0.003	14	0.00
football	0.14	96	0.02
jazz	0.05	160	0.01
email	0.67	605	0.02
delaunay_n10	0.55	740	0.05
netscience	1.07	899	0.00
power	10.76	2272	0.03
as-22july06	104.04	3312	0.00
hep-th	44.31	3947	0.01
star	69.71	7366	0.07
star2	76.73	4677	0.03
Comprehensive table with Local Search (1) Algorithm			
karate	0.02	14	0.00
football	0.01	95	0.01
jazz	27.34	158	0.00
email	24.26	603	0.02
delaunay_n10	69.08	741	0.05
netscience	15.36	899	0.00
power	37.11	2304	0.05
as-22july06	47.69	3344	0.01
hep-th	77.83	4023	0.02
star	7.11	7013	0.02
star2	189.89	4710	0.04
Comprehensive table with Local Search (2) Algorithm			
karate	0.002	14	0.00
football	8.83	96	0.02
jazz	1.0	163	0.03
email	58.43	638	0.07
delaunay_n10	43.67	766	0.09
netscience	22.89	914	0.02
power	596.90	2614	0.19
as-22july06	592.94	4317	0.307
hep-th	599.83	4376	0.10
star	596.91	8459	0.23
star2	595.15	5694	0.25

Table 1. The Comprehensive Table of All Algorithms

find its best solution, Vertex Cover Size is the size of the best vertex cover found by the algorithm, and Relative Error is the ratio calculated as $(|S| - |O|)/|O|$, where S is the vertex cover found by our algorithm and O is the optimal vertex cover. From table 1, we can observe that the average relative error increases as the size of the graph instance increases. In section 5.1, the bound of the approximation algorithm is $\min(H(\Delta), 2)$, where $H(\cdot)$ is the harmonic series and Δ is the maximum degree of the graph. The result in table 1 is consistent with the theoretical estimation as none of the relative error exceeds 1.

6.2 Branch & Bound Algorithm

From table 1, we can see that Branch & Bound (BnB) performs relatively well across all graph instances. In the *expand* step, we apply the greedy heuristic where we select the vertex with the largest degree first to expand first. It turns out that this greedy heuristic works surprisingly well as from the solution trace file of all graph instances with BnB, we can only see one or two records, which indicates that in the limited time, the first found solution is the best one. Since the solution space is exponential and the initial solution is reasonably well, it might take hours for BnB to further optimize the current best answer under the current hardware settings.

6.3 Local Search Algorithm 1

Local Search Algorithm 1 (LS1) has the best result in terms of relative error. It is worth noting that the run time of star instance is particularly small, which is only 7.11 seconds. This is due to the detail of implementation where we set an additional 20 seconds time limit for the algorithm to perform the random exchange step. If this step takes longer than 20 seconds and there is no better solution found, we terminate the algorithm and return the best result got at this point. This is to ensure the balance between run time and performance and to increase the variability of run time so that the run time of the algorithm would not be the time limit for every single run.

The Qualifed Runtime Distributions (QRTD) and the Solution Quality Distributions (SQD) of LS1 with respect to power and star2 graph instances are presented from figure 1 to figure 4 respectively. The QRTD plots are used to show the probability that an algorithm will solve a solution as time progresses for various solution qualities. The SQD plots show the probability of obtaining a solution as a function of solution qualities for various times. The solution qualities selected for LS1 with power instance and star2 instance are 5%, 10%, 15%. The time limits are 30 seconds and 110 seconds for the two instances respectively. For SQD plots, The cut-off time limits selected for LS1 are 10, 20, 30 seconds for power instance and 20, 40, 60 seconds for star2 instance.

In Figure 1, the QRTD of Local Search 1 with power Instance shows that it takes essentially same time (about 3 seconds) for LS1 with 15% and 10% solution quality to have 100% probability of finding a solution, whereas it takes additional 20 seconds run time for LS1 with 5% to get the same probability. This shows that it is increasingly difficult to get an improvement of quality as run time increases for LS1. This is reasonable because initially there are many available vertices for the random exchange step in LS1 to get a vertex cover with less vertices. As time goes, the quality of our vertex cover increases and the available vertices that can further improve the current cover decreases simultaneously, so the time taken to find a better solution increases. Similar analysis and conclusions can be obtained based on QRTD and SQD of LS1 with star2 instance, as shown in Figure 3 and Figure 4.

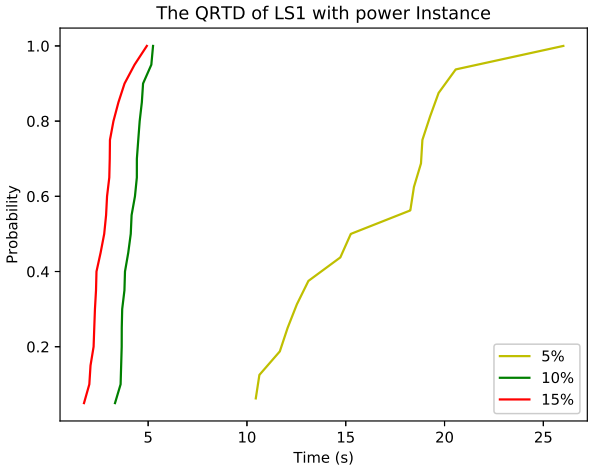

Figure 1. The QRTD of Local Search 1 with Power Instance

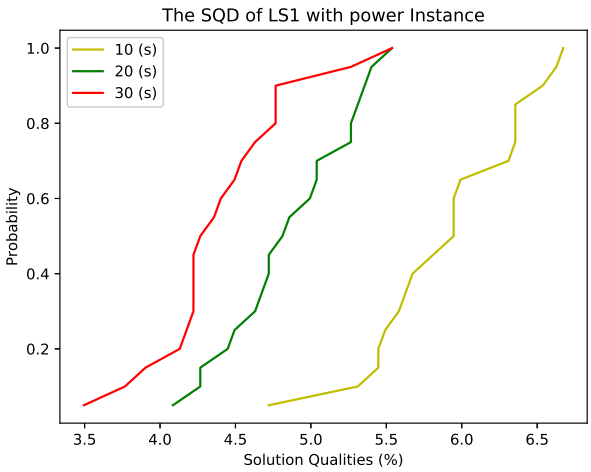

Figure 2. The SQD of Local Search 1 with Power Instance

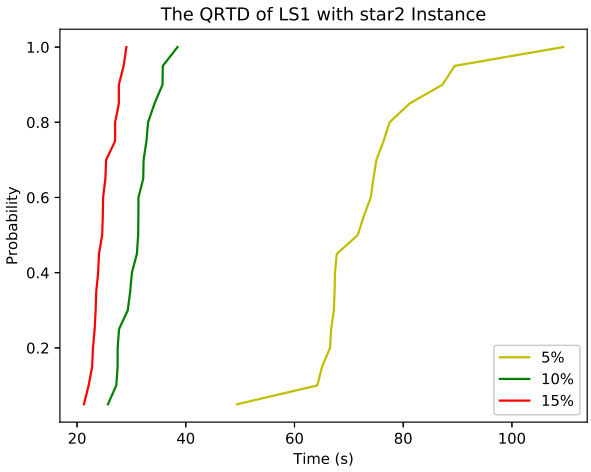

Figure 3. The QRTD of Local Search 1 with star2 Instance

6.4 Local Search Algorithm 2

As the size of the graph instance increases, Local Search Algorithm 2 (LS2) performs worse in terms of running time and relative error from the results in table 1. The QRTD and SQD distributions of LS2 with respect to power and star2 graph instances are presented from figure 5 to figure 8. The solution qualities selected for LS2 with power instance and star2 instance are 30%, 26%, 24% and 29%, 28%, 27% respectively. The time limit is 400 seconds for both instances. For SQD plots, the cut-off time limits selected for LS2 are 10, 60, 180 seconds for both instances.

From figure 5, the QRTD plot of LS2 with power instance indicates that with at most 30% of solution quality, LS2 will have 100% probability of finding a solution in about 25 seconds, whereas for the solution quality of 26% and 24%, the corresponding time to achieve probability of one are about 300 seconds and 400 seconds. This indicates that inferiority of LS2 because getting a smaller improvement in solution quality is becoming more and more difficult as time increases significantly. The SQD of LS2 with power instance shows the congruent result as the improvement of solution quality differs by only2% when we increase the cut-off run time from 60 seconds to 120 seconds. In figure 8, the SQD of Local Search 2 with star2 instance shows a counter-intuitive result where we can get the higher probability of obtaining a solution when running LS2 for 10 seconds than when running it for 60 seconds below the approximate solution quality of 26.6%. This can be due to the stochasticity of LS2 where the initialization and the random exchange of vertices between the cover set and its complement can have an impact on the final solution quality.

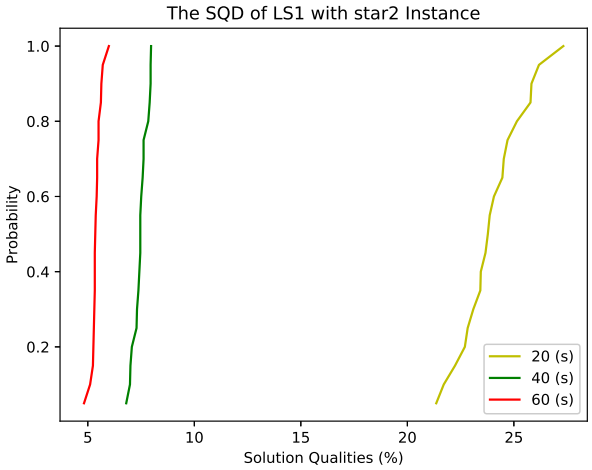

Figure 4. The SQD of Local Search 2 with star2 Instance

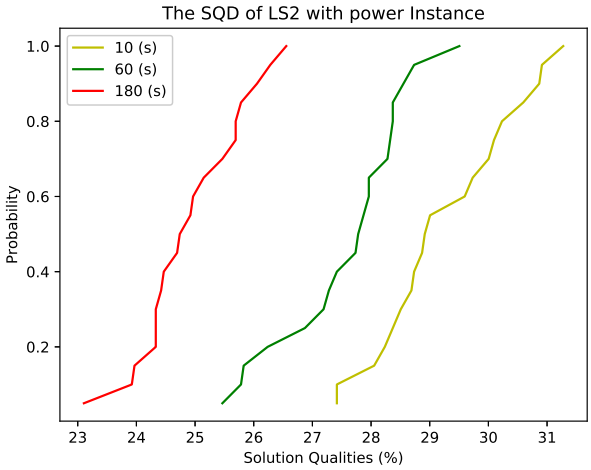

Figure 6. The SQD of Local Search 2 with Power Instance

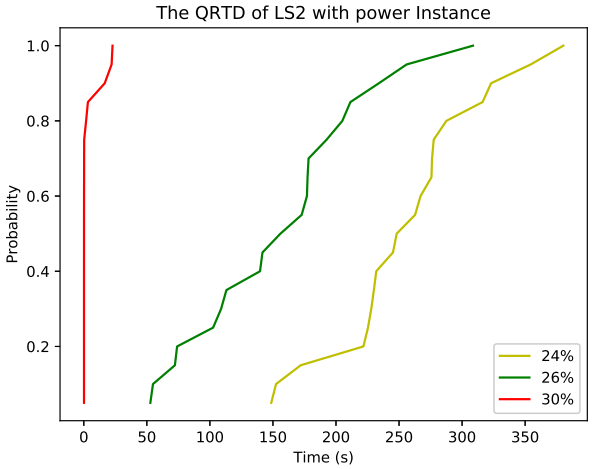

Figure 5. The QRTD of Local Search 2 with Power Instance

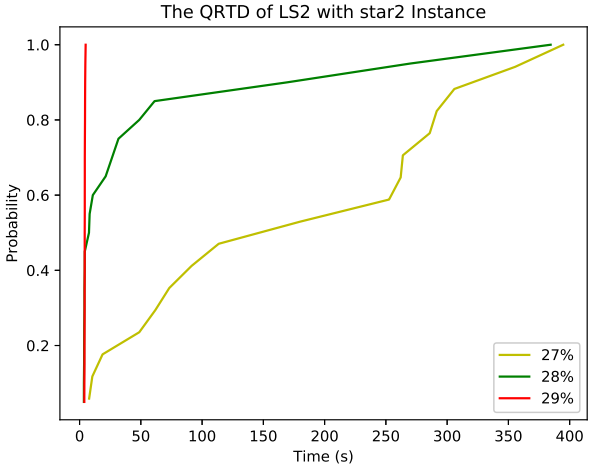

Figure 7. The QRTD of Local Search 2 with star2 Instance

7 Discussion

Based on the experiments and the empirical evaluation, we find that approximation algorithm can terminate quickly compared to the other three algorithms and produce solution within in the bound of 2 relative to the optimal solution. Nonetheless, it performs the worst in terms of the relative errors, especially for large graph instances such as star, for which the relative error is 46%. The time and space complexity are $O(EV^2 \log(V))$ and $O(V + E)$ as explained in section 5.1. The branch & bound algorithm is able to explore the whole search tree and guarantees to find the optimal solution given sufficient run time and memory. However, since the search space is exponential in terms of the number of vertices, it is unrealistic to be able to find the optimal vertex cover for every large graph instances. What we do is to apply the greedy heuristic to let the algorithm explore the promising nodes first and be able to return the relatively better upper bound in the time limit. The time and space complexity for branch & bound are $O(|V|log(|V|)2^{|V|})$ and $O((|V| + |E|)2^{|V|})$, as shown in section 5.2.

Local Search 1 performs better than Local Search 2 in both relative error to the optimal solution and run time. Figure 9 and Table 2 presents the box plots of run time for LS1 and LS2 on power Instance and its associated statistics. The number of independent trails is 10, and the time limit is 180 seconds for both algorithms. In Figure 9, we can observe that the median of running time of Local Search 1 is less than that of Local Search 2 by about 130 seconds. Based on the same plot, it is also apparent that Local Search 2 always

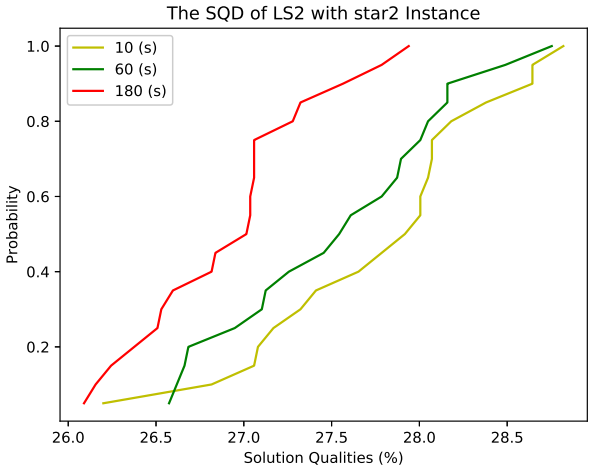

Figure 8. The SQD of Local Search 2 with star2 Instance

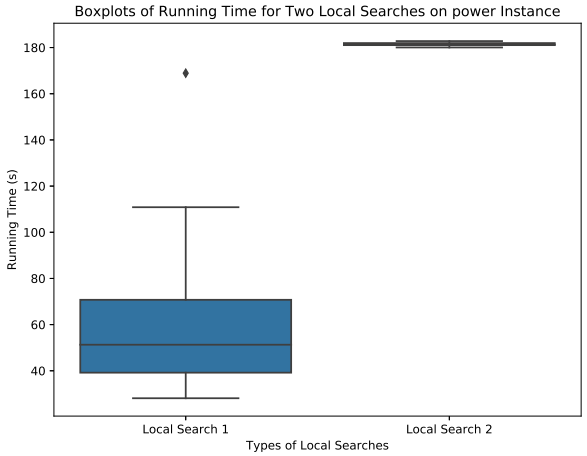

Figure 9. The Box Plots of Run Time for LS1 and LS2 on power Instance

runs to completion of the time limit. This is possibly due to the inefficiency of one exchange in which a same vertex can be randomly exchanged multiple times. To address this issue, we think Tabu Search may be a worthy attempt as it prevents the scenario of repeatedly accessing the same vertex in certain recency. The time complexities for LS1 and LS2 are both $O(N)$ and the space complexities are both $O(1)$, as explained in section 5.3 and 5.4.

Run Time Statistics (s) for LS1 and LS2 on power instance		
Quantile	LS1	LS2
max	168.91	182.91
75%	70.74	181.94
50%	51.29	181.31
25%	39.18	181.06
min	28.11	180.05

Table 2. Run Time Statistics (s) for LS1 and LS2 on power instance

8 Conclusion

In this project, we implement approximation, branch & bound, and two local search algorithms to solve the optimization version of the Vertex Cover problem. With the analysis of the comprehensive table, QRTD and SQD plots, we conclude that in general, if running time is the primary concern and the solution can be relaxed to a certain bound, then the approximation algorithm is mostly desired. Branch & bound can guarantee to find the optimal solution, but due to the time limit and computational constraints, the promise of optimality can be seldom expected in real life. Local search provides a balance between accuracy and speed. However, the performance of the local search depends largely on implementation and randomization, as shown by the run time and relative error between local search 1 and local search 2 implemented in this project. Therefore, in the future, we expect to try out more local search ideas such as simulated annealing to have an integrated view of local searches in terms of their performance and run time distribution.

References

[1] David Avis and Tomokazu Imamura. 2007. A list heuristic for vertex cover. *Operations research letters* 35, 2 (2007), 201–204.
[2] Thomas H Cormen. 2009. Introduction to algorithms, Chapter 8.2.
[3] François Delbot and Christian Laforest. 2008. A better list heuristic for vertex cover. (2008).
[4] Michael R Garey and David S Johnson. 1979. *Computers and intractability*. Vol. 174. freeman San Francisco.
[5] Magnus M. Halldorsson and Jaikumar Radhakrishnan. 1994. Improved approximations of independent sets in bounded-degree graphs via subgraph removal. *Nord. J. Comput.* 1, 4 (1994), 475–492.
[6] Carla Savage. 1982. Depth-first search and the vertex cover problem. *Information processing letters* 14, 5 (1982), 233–235.