CSE 6140 Project: Minimum Vertex Cover

ANDREA COVRE, acovre3
ZHAONAN LIU, zliu629
SHIKAI JIN, sjin42
YIFAN MA, yma379

1 INTRODUCTION

In the minimum vertex cover (MVC) problem, it is asked to find the smallest subset of vertices that for every edge in the graph, at least one of its endpoint is in the subset. The minimum vertex cover problem has been extensively studied due to its wide applications in both real-world and theoretical problems. The minimum vertex cover problem is a classical optimization problem. As one of the fundamental NP-hard problems, it cannot be solved by a polynomial-time algorithm if P ≠ NP. In this work, we applied four different approaches to solve the MVC problem, including a Branch and Bound (BnB) algorithm to get an optimal solution, an approximation algorithm using Maximum Degree Greedy, and two local search algorithms(Simulated Annealing and Genetic Algorithm). The performance of these four algorithms are tested with real and random datasets from theh 10th DIMACS challenge, and their accuracy and computational efficiency are analyzed by comparing their run time, relative error, etc.

2 PROBLEM DEFINITION

Given an undirected graph G = (V,E) with a set of vertices V and a set of edge E, a vertex cover is subset of vertices C ⊆ V such that for $\forall(u,v) \in E : u \in C \vee v \in C$. Literally, at least one end of every edge in graph G has to be included in the vertex cover C. The problem of interesting for this report is to search for minimum vertex cover, which is to find the vertex cover for the graph with least amount of vertices.

3 RELATED WORK

The Minimum Vertex Cover (MVC) problem is a prominent NP-hard optimization problem of great importance in theory and real-life application such as network security, scheduling and assignment problems. It is also closely associated with the Maximum Independent Set (MIS) problem and the Maximum Clique (MC) problem, for which the algorithm designed for one can also serve the two problems as well. MVC is also difficult to approximate as well, and is NP-hard to approximate within a factor of 1.3606 [3]. Typically, algorithms designed to solve MVC are either exact or heuristic. Exact algorithms mostly consist of branch-and-bound (BnB) algorithms, which can yield a optimal solution but is very expensive in terms of computational cost particularly for large instances [4]. To optimize the BnB algorithm, lower bound conditions based on vertex degree and MaxSAT had been proposed to help prune the search space [5].
On the other hand, heuristic algorithms such as local search do not guarantee optimal solution, but is efficient in finding near-optimal solutions within reasonable time, making it much more desirable for solving large instances of MVC problem.

4 ALGORITHMS

4.1 Branch and Bound

4.1.1 Description. The branch-and-bound algorithm performs a systematic search on possible solutions by exploring branches of smaller subproblems (in this case partial graphs) based on previous decisions. Starting out with a partial solution, lower bound for all the subproblems within the branch of possible solutions is computed and compared with the upper bound from the best existing solution, the branches that have a less optimal bound are considered not promising and can be eliminated from consideration, while the rest of the branches that could yield the optimal solution will be investigated.

4.1.2 Algorithm. In this branch-and-bound algorithm, the upper bound of the solution is updated using the best vertex cover solution found so far. The lower bound of each subproblem is evaluated using a degree-based method defined as: $LowerBound(G) = \frac{E}{\Delta}$, where E is the total number of edges in graph G, and Δ is the maximum degree of the graph. The obtained lower bound is at least as good as the optimal solution because for any graph G, at least LowerBound(G) number of vertices is needed to cover $|G|$. Other lower bounds such as maximal matching of the graph were also considered, and the method is chosen based on the time cost observed in empirical evaluation.

The choice of exploration path is also degree-based. The partial vertex cover grows each time by acquiring the vertex with the most degrees (uncovered edges connected to it) in the remaining graph (G'), which is considered the most promising move. Backtracking from the "mistakes" is achieved by storing states of each vertex (in/out of vertex cover) in the frontier set and recover the partial graph back to the condition corresponding to each partial vertex cover. Then the exploration can advance to search for another vertex cover as it did in the initial exploration.

The pseudo-code of this branch and bound implementation is presented in Algorithm 1.

4.1.3 Time and space complexities . Due to the nature of the branch-and-bound algorithm, this implementation is very expersive to run. For each vertex in the graph, two scenarios (in/out of vertex cover) are considered, so each additional node would introduce x2 new test cases. In the worst case scenario, no branch is pruned and every possible solution needs to be evaluated just like the brute force method. The total number of test cases is 2^V where V is the total number of vertices in graph G. Therefore, the time complexity of the BnB algorithm is exponential $O(2^V)$.

The space complexity of the algorithm also depends on the number of vertices and edges in the graph. For each vertex, 2 space is needed to store the possible solutions in the vertex set. For each edge in the graph, 1 space needed to explore the edge in search for the vertex cover. Therefore, the total space complexity of the BnB algorithm is $O(V + E)$

Algorithm 1: Branch and bound

```
Input: graph G, cutoff time
Output: vertex cover $VC$
$G' \leftarrow G$// Copy graph to explore ;
Initialize vertex cover set(VC) and upper bound;
$v$ ←initial vertex with maximum degree;
Frontier = [(v,state,(parent node and state)] //State=1 if in VC, 0 if not;
while $Frontier$ is not empty and cutoff time not reached do
    Select vertex $v$ from $Frontier$;
    $backtrack \leftarrow$ false //exploration;
    if $v$ not in VC then
        Add all neighbor of $v$ to VC (state = 1);
        Remove all neighbor of $v$ from $G'$;
    else if $v$ not in VC then
        Add $v$ to VC;
        Remove $v$ from $G'$;
    end;
    if $G'.edge == \varnothing$ (empty graph) then
        if len(VC) < Upperbound then
            $solution \leftarrow VC$ ;
            $Upperbound \leftarrow$ len(VC) //update UB with best VC ;
            Record solution and trace;
            backtrack ← true;
            //Advance to next subproblem;
        end
    else
        Lowerbound ← len(VC) + Lowerbound(G') //max. degree-based LB;
        if Lowerbound < Upperbound then
            $v_j \leftarrow$ max. degree vertex in G' ;
            Frontier $\cup[v_j, state, (v_i, state_{vi})]$ ;
        else
            backtrack = True //reject path;
        end;
    end;
    //backtracing if backtrack is true then
        if Frontier is not empty then
            Revert to parent node in Frontier;
            for vertices in VC after the parent node do
                remove from VC ;
                Recover in G' with connected edges;
            end
        else
            Reset to initial graph G;
    end;
end
return $VC$;
```

4.2 Approximation

4.2.1 Description. We adopted an approximation algorithm Maximum Degree Greedy (MDG) described in [2]. This algorithm greedily adds the vertex with the maximum degree in the remaining graph at each step, and removes this vertex from the remaining graph. When the remaining graph is empty, the solution must be a vertex cover, and this solution is returned. The worst-case approximation ratio is $H(\Delta)$, where $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} \approx \ln n + 0.57$. Δ is the maximum degree of the graph.

This algorithm is used to produce initial solutions for both Local Search algorithms.

4.2.2 Pseudo-code.

```
Algorithm 2: Approximation: Maximum Degree Greedy (MDG)
Data: graph G, cutoff time
Result: vertex cover $C$
$G' \leftarrow G$;
while $G'$ is not empty do
    find node $u$ in $G'$ with maximum degree;
    $C \leftarrow C \bigcup \{u\}$;
    $G' \leftarrow G' - \{u\}$;
    remove any isolated nodes;
    if $time \geq cutoff\ time$ then
        break;
    end
end
return $C$;
```

4.2.3 Time and space complexities. The time complexity of MDG is given by $O(V^2)$. At each iteration, the algorithm finds the node u with maximum degree, and this step will take $O(V)$ time. At each iteration, the algorithm removes u, and this step take $O(\Delta)$ time. The space complexity is given by $O(V\Delta)$, which is the space complexity of the graph adjacency list. Our code implements this variant.

Alternatively, we could save the degree of each node and update at each step, and store them in a fashion of a priority queue. In this way, finding the maximum will take $O(1)$ time, and updating the degree and priority queue takes $O(\Delta \log V)$ time. The time complexity of this MDG would be $O(\Delta V \log V)$. The space complexity still is $O(V\Delta)$. This variant is not implemented in our code.

4.3 Local Search: Simulated Annealing

4.3.1 Description. Simulated Annealing (SA) is a class of algorithms inspired by the annealing process in metal processing. Annealing is a process where heat is applied to increase mobility of the atoms in a metal, and gradually decrease temperature for the atoms to settle down. In an annealing process, atoms tend to move to a more stable position with lower energy. According to statistical physics, the expected number of classical particles N_i in a microstate can be described by the Maxwell-Boltzmann distribution:

$$\frac{N_i}{N} = \frac{\exp(-\frac{E_i}{kT})}{\sum_j \exp(-\frac{E_j}{kT})}, \tag{1}$$

where N is the total number of particles in the system, E_i is the energy of microstate i, T is the temperature, and k is the Boltzmann constant. The equilibrium of a classical system can be found by the M-B distribution, and the system will spontaneously approach this state. However, there may exist semi-stable states with energy barriers that the particles do not have enough energy to overcome. This is where annealing shows its power. The system is heated so that the particles can go to states with higher energy, and then gradually cooled down so that the particles can find their ways to lower energy states.

The same mechanism can be used to solve hard computational problems where local optima exist. Inspired by statistical physics, the probability P of accepting a change can be described by the following equation:

$$P = \exp(-\frac{\Delta E}{T}), \tag{2}$$

where ΔE is the change in solution quality, and T is a "temperature" used to regulate the distribution. When this "temperature" is high, the distribution is more "flat" so that the possibility of going to a higher energy is higher; when the "temperature" is low, the distribution is more "steep" so that the distribution is largely gathered at the low energy end.

We adopted the same algorithm structure as described in [1], where the vertex addition and removal are done separately instead of switching them. Since $\exp(-\frac{E_1-E_2}{T}) = \exp(-\frac{E_1}{T})\exp(\frac{E_2}{T})$, the probability of removing a vertex is $\exp(-\frac{loss}{T})$, while the probability of adding a vertex is $\exp(\frac{gain}{T})$, where *loss* and *gain* are defined by the loss and gain of number of covered edges. We used 2 as the base instead of e for easy implementation:

$$\begin{aligned} P_{remove} &= 2^{-loss/T} \\ P_{add} &= 2^{gain/T} \end{aligned} \tag{3}$$

An exponential decay of temperature is an easy and effective way to define temperatures in SA algorithms. The temperature at each iteration is a certain fraction of the temperature at the previous iteration. Here we call this fraction the "cooling rate". We have tried different values for initial temperature and the cooling rate, and found that a good initial temperature is between 1 and 0.1. A good cooling rate is above 0.98.

When trying out different initial temperatures and cooling rates, we found out that the quality of the best solution found drops (lower is better) quickly between 0.1 and 0.01, especially between 0.05 and 0.01. Therefore, staying at those temperatures for a longer time could be a better way than an exponential decay. A group member has experience in 3D metal printing, and recalled that the annealing process is in several stages where the temperature is kept constant at each temperature. Therefore, we designed a temperature profile with four stages: Heating, Annealing 1, Annealing 2, and Cooling. After some experiments, we have found a temperature profile that outperforms exponential decay on most graphs:

(1) Heating: 10% of total time, T=0.1
(2) Annealing 1: 30% of total time, T=0.05
(3) Annealing 2: 30% of total time, T=0.01

(4) Cooling: 30% of total time, T=0.001

Another merit of the annealing temperature profile is controlling the runtime. The algorithm will always run until it exceeds the cutoff time, which can be useful in certain scenarios where the time is limited, and we want to make the best use of the time. Otherwise, we could also set the profile as $\{\#\_of\_iterations : temperatures\}$.

4.3.2 Pseudo-code.

Algorithm 3: Local Search: simulated annealing

```
Data: graph data (adjacency list), cutoff time
Result: best vertex cover found $C^*$
Set temperature profile $Pf\{phase\_end\_times : temperatures\}$;
Run Algorithm 2 (greedy algorithm) to get an initial solution $C$;
while time < cutoff time do
    for phase in $Pf$.keys do
        if time < phase then
            $T = Pf[phase]$;
            break;
        end
    end
    if $C$ is a vertex cover then
        $C^* \leftarrow C$;
        randomly remove a vertex with probabilities proportional to P in Equation 3;
        continue ;
    end
    randomly add a vertex with probabilities proportional to P in Equation 3;
end
return $C^*$;
```

4.3.3 Time and space complexities. The time complexity of the vertex removal step is given by $O(V\Delta)$, where V is the total vertex count, and Δ is the maximum degree of the vertices in the graph. In the vertex removal step, the algorithm checks each vertex in the current solution, and calculates the loss in covered edges if each of them is to be removed.

The time complexity of the vertex addition step is given by $O(\Delta^2)$. In the vertex adding step, the algorithm first look for the candidates, and the number of candidates is bound by Δ, because adding the vertex just removed in the previous step will make the solution a vertex cover. Evaluation the gain in covered edges if a candidate is to be added will take $O(\Delta)$ time.

Generally considered, The time complexity of this SA algorithm is $O(V\Delta)$ for each step.

The space complexity of the algorithm is given by $O(V\Delta)$, which is the space complexity of the adjacency list of the graph.

4.4 Local Search: Genetic Algorithm

4.4.1 Description. Genetic Algorithms are a class of algorithms that imitates the Darwinian evolutionary process to produce incrementally better solutions to a problem. These algorithms have a population, where each individual is a candidate solution with a fitness score that represents how good that solution is. Individuals with higher fitness are more likely to reproduce and create a new individual of next generation's population. When two individuals reproduce, their "genetic information" is mixed and used to create a new individual that has characteristics from both parents (crossover). This new individual also undergoes a mutation step in which its genetics is randomly changed to both introduce new genetic information, as well as increase variety in the population. This new generation will then become the current population which will in turn generate new individuals.

We chose to develop a genetic algorithm for our second local search algorithm because it is not a class of algorithms commonly used, therefore we wanted to explore and learn a novel class of algorithms, and because genetic algorithms could be classified as a type of AI with which would be interesting to draw a comparisons with.

To solve the Vertex Cover problem, we developed and tested various functions and methods to initiate the population, perform the crossover, introduce mutations, and calculate the fitness score. For example, we tried to initiate the population with a "uniform" method, where each individual represents a random vertices. Another method we tried is initiating the population through a log-norm probability distribution with which individuals are more likely to be initialized with a number of vertices closer to the total number of vertices in the graph. At the end, we decided to initialize the population with a noisy version of the solution given by the Approx. algorithm.

In regards of the fitness function, we tried a variety of formulas that keep into account the number of verteces used as well as the number of covered edges. We initially opted for a very basic function $\frac{|\text{\# of edges covered}|}{|\text{\# of vertices used}|}$ which would make individuals with a higher number of edges covered and a lower number of vertices included much more likely to reproduce. However we noticed that the algorithm would end up focalizing it's search in solutions with a greater number of vertices than the best solution found so far, therefore progress was very slow. We tried other combinations as well where we would award an incrementally higher score for each additional edge covered. However, we actually found out that giving a major penalty to any individual that is not a vertex cover, and giving a linear reward for the number of vertices not used produced the best results across most scenarios. This function basically pushes the population to minimize it's average solution size while maintaining mostly valid solutions.

For the reproduction steps we decided to select parents probabilistically based on the ratio between the parent fitness and the total fitness of the population. For the crossover function we decided to create the new individual based on the shared vertices between the two parents. While for the mutation we tried different function. One of them included just adding/removing a random vertex, however, this method excessively pushed the population toward an equal number of vertices included or excluded (e.g. a vertex removal would happen more frequently on individuals of higher size and vice-versa). Another approach included removing random vertices which would help quickly minimize the average size of the population (especially if starting from a trivial solution), however this approach would only create variability through solutions of smaller size, therefore we decided to move forward by independently adding and removing a random amount of vertices, which maintains a large standard deviation in the population size, a average size centered on the current best solution rather then too much below it, and most importantly it allows to add vertices that might allow for a better minima.

Parameters like population size and mutation rate have been adjusted manually. The tuning was based on the performance of the algorithm (grid search tuning), on the chosen function for population initialization, crossover, mutation and fitness, and on the resulting average and standard deviation of the population size (the goal was indeed to keep the average size around the size of the best solution found so far, and to prevent the standard deviation to collapse to 0, but keep it at least above 4 to maintain a fair size variability across the population).

4.4.2 *Pseudo-code.*

```
Algorithm 4: Local Search: genetic algorithm
Data: graph data (adjacency list), cutoff time
Result: best vertex cover found $C^*$
Initialize population;
calculate population fitness;
while time < cutoff time do
    for 0 → population size do
        parent A ← probabilistically select an individual from population based on fitness
        parent B ← probabilistically select an individual from population based on fitness
        new individual ← mutate(crossover(parent A, parent B)
        append new individual to new population
    end
    population ← new population
    calculate population fitness;
    C ← get individual with highest fitness
    if C is a vertex cover then
        $C^*$ ← C;
        continue ;
    end
end
return $C^*$;
```

4.4.3 *Time and space complexities.* Generating each new generation has time complexity equal to $O(PVE)$ where P = population size, V = number of vertices in G, and E = number of edges in G. This is because at each iteration the most expensive operation is calculating the fitness of each of the P individuals, and such calculation entails counting all the edges covered by all the vertices that are part of the individual. The space complexity is instead just $O(PV)$ since we basically always maintain a list of vertexes for each one of the P individuals.

This algorithm is very costly to run as it heavily relies on many random operations that are particularly expensive (e.g. randomness is involved in parent selection, crossover and mutations). Moreover, the search is very scattered and disorganized. For example, any population might have individuals that are very similar and redundant, and new solutions are randomly probed in the search space. Whenever a good solution is found the population tends to converge to it decreasing variability. We also realized with hindsight, that a genetic algorithm might just not perform well on problems like the vertex cover due to the uni-dimensionality of the individuals (a vertex is either included or not), therefore it's very likely that the population will not branch out into evolving new and different solutions, but it's more likely to just converge, and get stuck in a local minima

5 EMPIRICAL EVALUATION

5.1 Experiment Setup

- **Implementation:** All algorithms are implemented and compiled in python 3.8.12.
- **Platform:**
  - **OS:** Windows
  - **OS Release:** 10
  - **OS Version:** 10.0.22000
  - **Processor:** Intel64 Family 6 Model 158 Stepping 10, GenuineIntel
  - **Processor Architecture:** AMD64
  - **RAM:** 32 GB

- **Experimental Procedure:** All graphs are solved with all four algorithms for 10 times each combination with a cutoff time of 100 seconds. star2.graph and power.graph are solved using the two Local Search algorithms for 20 times each combination with a cutoff time of 1000 seconds.
- **Outputs:** Each implemented algorithm outputs a trace file where the time and solution quality is stored whenever it finds a better solution. Each implemented algorithm also outputs a solution file with the best solution found.

Comprehensive Results Table from All Algorithms and Graphs with 100s cutoff time

Algorithm:	Branch and bound	Approximation	LS: simulated annealing	LS: genetic algorithm
jazz.graph				
Time (s)	0.09	0.01	0.11	0.23
VC Value	158	159	158	159
RelErr	0	0.0063	0	0.0063
karate.graph				
Time (s)	0.0	0.0	0	0.1
VC Value	14	14	14	14
RelErr	0	0	0	0
football.graph				
Time (s)	0.02	0	0.031	0.20
VC Value	95	96	94	95
RelErr	0.011	0.0	0	0.011
as-22july06.graph				
Time (s)	10.46	2.67	40.8	49.43
VC Value	3307	3307	3303.2	3305.9
RelErr	0.0012	0.0012	0.000061	0.00088
hep-th.graph				
Time (s)	11.21	2.52	72.59	82.14
VC Value	3944	3944	3929.9	3939.2
RelErr	0.0046	0.0046	0.00099	0.0034
star.graph				
Time (s)	35.35	7.13	92.17	85.7
VC Value	7374	7374	7319.6	7368.6
RelErr	0.068	0.068	0.061	0.068
star2.graph				
Time (s)	18.46	3.48	93.04	93.16
VC Value	4697	4697	4595.0	4689.4
RelErr	0.034	0.034	0.012	0.032
netscience.graph				
Time (s)	0.52	0.12	0.12	0.57
VC Value	899	899	899	899
RelErr	0	0	0	0
email.graph				
Time (s)	0.34	0.07	16.21	6.53
VC Value	605	605	594.3	602
RelErr	0.019	0.019	0.00051	0.013
delaunay n10.graph				
Time (s)	0.34	0.07	52.52	4.45
VC Value	737	737	703.6	735
RelErr	0.048	0.048	0.00085	0.0046
power.graph				
Time (s)	3.74	0.82	86.72	83.20
VC Value	2277	2277	2209.5	2258.5
RelErr	0.034	0.034	0.003	0.025

Figure 1 shows the QRTD and SQD of the two Local Search algorithms on power.graph and star2.graph, respectively. These two graphs are quite large, so we are using 1000 seconds as the cutoff time. Each combination (e.g., LS1 on power.graph) is run for 20 times to address the randomness. In the QRTD plots, the x axis, cutoff time, is in log scale, while P_{solve} is in linear scale. The qualities for the QRTD plots are distributed evenly in the range of the maximum quality and minimum quality achieved.

In the SQD plots, both axes, quality and P_{solve}, are in linear scale. The cutoff times for the SQD plots are evenly distributed in log scale.

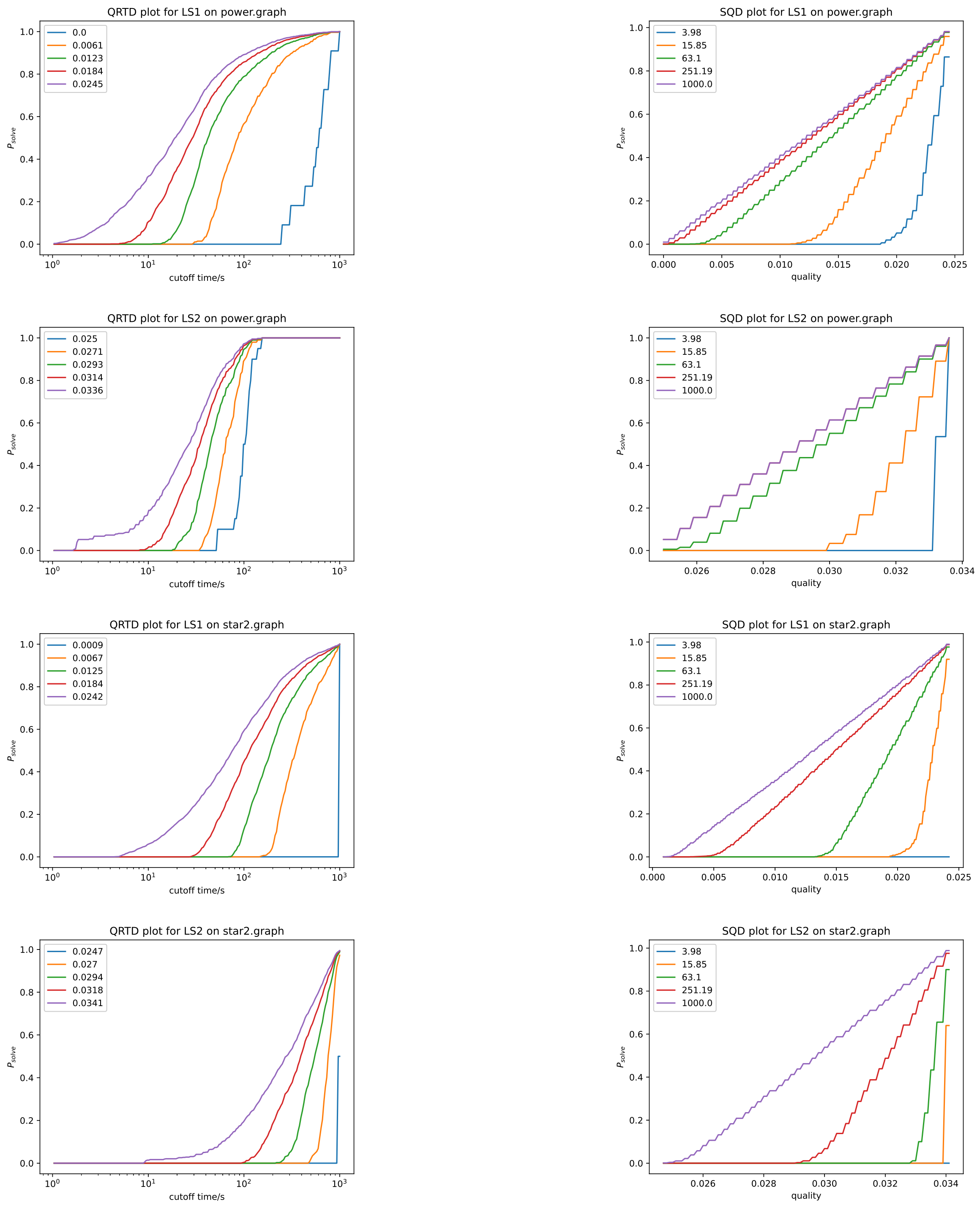

Fig. 1. QRTD and SQD plots of Local Search algorithms on power.graph and star2.graph.

Figure 2 shows the distribution of critical runtimes and best solution qualities for both Local Search algorithms.

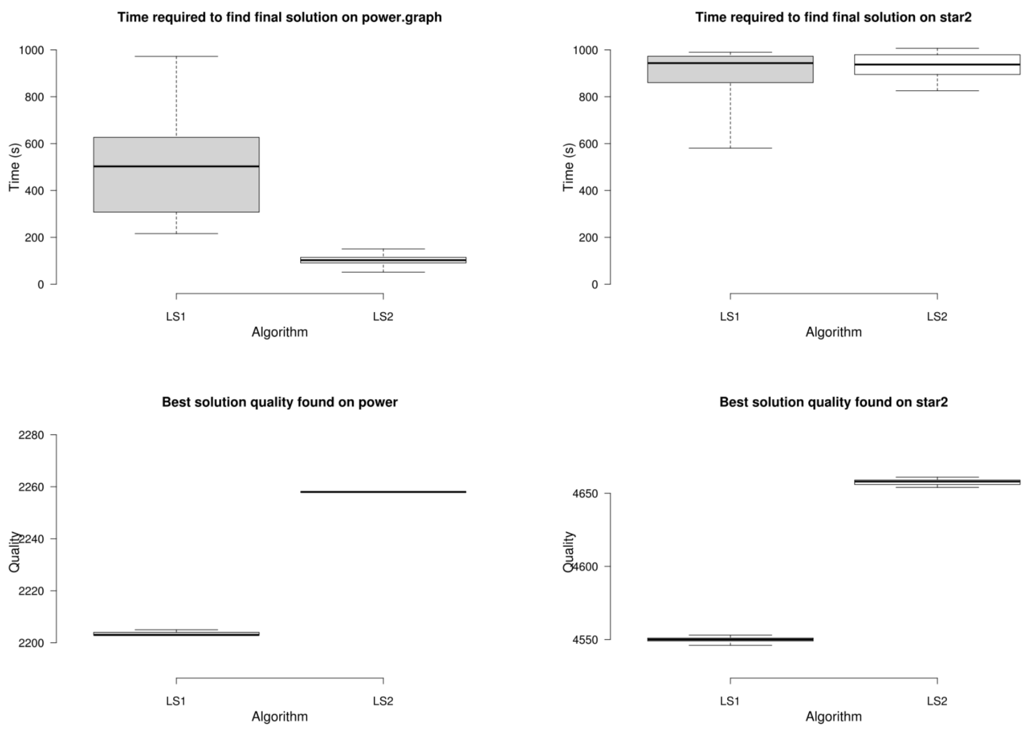

Fig. 2. Results of LS1 and LS2 on power.graph and star2.graph (*cutoff time* = 1000*s*).

6 DISCUSSION

6.1 Branch and Bound and Approximation Algorithm Results

Based on the reported results, it can be observed that the BnB algorithm can obtain the initial result fairly quickly, and the quality of the results mostly resembles that of the approximation algorithm in this study. This is to be expected because both BnB and approximation algorithm in this study relies on searching for the maximum degree node in the graph to construct their solutions. Therefore, in practice the first complete vertex cover obtained by BnB should be exactly the same as the solution returned by the approximation algorithm. After the initial solution is found, the BnB algorithm then begins to investigated other potential branches that could possibly yield a more optimal solution than the current solution. Because this operation is so computationally expensive, the subsequent solutions are very slow to return. For relatively large instances, only the first solution can be obtained in the test case with 100s cutoff time. Limited trials of extended cutoff time indicate that the more optimal solution from backtracking would typically be obtained in hours, and is deemed too expensive in practice,

6.2 Local Search algorithms

From the QRTD and SQD plots shown in Figure 1, we can have a clear view of how the algorithms are doing on power.graph and star2.graph.

In QRTD plot for LS1 on power.graph, we can see that the blue curve is corresponding to quality of 0, which means that LS1 has found an optimal solution several times for this graph within 1000 seconds. Other curves also show that this algorithm finds a good solution with relative error better than 1%. LS2 in comparison, does not perform as well, as we can see that it struggles to find the optimal solution as quickly as LS2.

In QRTD plot for LS1 on star2.graph show that LS1 does not perform so well as on power.graph. LS1 only finds solutions with relative error better than 1% around 100 seconds. This is only natural since star2.graph is significantly larger than power.graph. Similarly, LS2 also performs not as well as it did on power.graph due to the larger graph size.

By comparing QRTD plots for LS1 and LS2, we can see that LS1 performs significantly better than LS2 on power.graph. The same can be found for star2.graph. The reason why LS2 performance is lower on these graphs is probably its time complexity. LS2 needs to update a population of vertex covers for each generation, which can be very time-consuming. However, as a genetic algorithm, LS2 may find better solutions than LS1 when the optimal solution is very far from the results generated by the MDG algorithm.

7 CONCLUSION

We have designed, implemented, and tested four algorithms to solve the Minimum Vertex Cover problem: one Branch and Bound algorithm, one approximation algorithm, and two Local Search algorithms.

We have found out that although Branch and Bound algorithm can secure an optimal solution, but the runtime is too long. Also, the performance of the BnB algorithm in finding sub-optimal solutions is significantly worse than Local Search algorithms. The approximation algorithm runs very fast, and finds a solution with decent quality.

The first Local Search algorithm, a Simulated Annealing algorithm can find solutions close to optimal within a reasonable runtime. We tried different modes and parameter values for LS1, and found an annealing temperature profile with multiple stages (temperature is kept constant in each stage) that works particularly well on the given graphs.

The second Local Search algorithm, a Genetic Algorithm, although not performing as good as the Simulated Annealing algorithm, also finds sub-optimal solutions much better than from the approximation algorithm with a reasonable runtime. However, we think that a Genetic Algorithm is an excessively complex approach to solve a straightforward problem like the Vertex Cover, hence other simpler local search algorithms, such as Hill Climbing, might be better suited as they could offer better solutions at much lower computational costs.

REFERENCES

[1] Shaowei Cai. 2015. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *Twenty-Fourth International Joint Conference on Artificial Intelligence.*

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

[3] Irit Dinur and Samuel Safra. 2005. On the Hardness of Approximating Minimum Vertex Cover. *Annals of Mathematics* 162, 1 (2005), 439–485. http://www.jstor.org/stable/3597377

[4] Etsuji Tomita and Toshikatsu Kameda. 2007. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global optimization* 37, 1 (2007), 95–111.

[5] Luzhi Wang, Chu-Min Li, Junping Zhou, Bo Jin, and Minghao Yin. 2019. An Exact Algorithm for Minimum Weight Vertex Cover Problem in Large Graphs. https://doi.org/10.48550/ARXIV.1903.05948