Algorithms for Solving the Minimum Vertex Cover Problem

Pratyusha Karnati
Georgia Institute of Technology
gkarnati3@gatech.edu

Prajwal Kumar
Georgia Institute of Technology
pkumar98@gatech.edu

Sahil Arora
Georgia Institute of Technology
sarora@gatech.edu

1 INTRODUCTION

The Minimum Vertex Cover (MVC) problem is a well known NP-complete problem with numerous applications in various industries. MVC aims to find a subset of vertices for a given graph such that all vertices in the subset cover all the edges of the graph. In this paper, we will explore solving the MVC problem using four different approaches and evaluate their theoretical and experimental complexities. Our Branch and Bound algorithm was slow in its running time but was fairly accurate in finding a minimum vertex cover for the given graph. On the other hand, the approximation performed very quickly but had the highest relative error. Our first local search implementation of hill climbing did very well with fast runtimes and outputted accurate MVCs close to the optimal solution. Lastly, the second local search max independent set approach did slightly worse than the hill-climbing approach. Our best approach was hill climbing. Our worst approach was construction heuristic.

2 PROBLEM DEFINITION

Consider an undirected graph G = (V,E) denoting the set of edges and vertices. Formally, a vertex cover V' can be defined as a subset of V such that for each (u, v) ∈ E, either u ∈ V' or v ∈ V'. In this case, V' would cover all the edges of G. Our goal is to use different algorithmic approaches to find the minimum vertex cover V' that satisfies the properties of a vertex cover as mentioned above. The following graph in figure 1 shows an example of a minimum vertex cover as represented by vertices a, c, f, g.

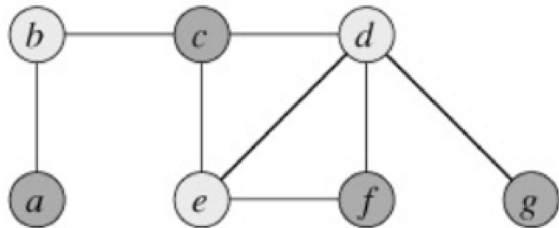


Figure 1: Minimum vertex cover of graph G = (V,E)

3 RELATED WORK

In 1972, Karp proved that MVC is NP-complete [1]. Exact methods to solve MVC mainly include branch-and-bound algorithms that guarantee the optimality of the solutions they find, but may fail to give a solution within reasonable time for large instances. An exhaustive search algorithm can solve the problem in time $2kn^{O(1)}$, where k is the size of the vertex cover. Vertex cover is therefore fixed-parameter tractable, and can be solved in polynomial time for small k. Downey and Fellows used kernelization on the fixed parameter problem to reduce the runtime to $O(kn + 2^k k^2)$ [2]. Parallelization has also been employed to make finding exact solutions more feasible. Researchers like Dehne et al. reported that they used fixed parameter tractable algorithm to solve the minimum vertex cover problem on coarse grained parallel machines successfully [3].

Due to the inherent computational intractability of the MVC problem, however, many researchers have instead focused their attention on the design of approximation algorithm for delivering quality solutions in a reasonable time. Garey and Johnson presented a simple approximation algorithm based on maximal matching that gave an approximation ratio of 2 for the general graphs [4]. Improvements have been made on different types of graphs that have better approximation ratios, but a hard limit was found. In 2005, it was proven that it is still NP-hard to approximate MVC within a factor smaller than 1.3606 [5]. Many unique approaches have been explored, such as the use of Hopfield neural networks to find efficient covers [6]. Simulated annealing [7] and genetic algorithms such as Ant Colony Optimization have also had promising results [8]. However, this report will focus on exact solutions via branch-and-bround, and approximate approaches with local search and a construction heuristic.

4 ALGORITHMS

The following describe our four algorithmic approaches: branch and bound, construction heuristic, hill climbing, and max independent set. Each section highlights the algorithm's description, pseudocode, time and space complexity analysis, and an analysis on its strengths and weaknesses.

4.1 Branch and Bound

4.1.1 Description. The branch and bound algorithm is derived from the backtracking strategy for decision problems. This algorithm searches through the complete space of solutions for the given problem for the best solution and eliminates the unpromising alternatives from consideration that don't meet the bounds of the optimal solution. However, as the size of the graph increases exponentially, it becomes nearly impossible for explicit enumeration. Therefore, the algorithm further refines the backtracking idea by optimizing. By keeping track of the best solution found, it computes a new lower bound to constrain the search space. When a solution for the sub-problem is found that is worse than the lower bound, we consider it an unpromising solution and continue to prune it and all its child nodes. If a solution is found that is better than the current vertex cover, it will replace the current best solution. The algorithm will terminate once there are no more branches to explore or once the running time exceeds the cutoff time threshold.

4.1.2 Pseudocode. See Algorithm 1.

In our implementation, the upper bound is the best solution minimum vertex cover and starts off as the |V|, the number of nodes

in G. On the other hand, the lower bound is defined as LowerBound(subproblem) = |VC'| + $\frac{G'.edges}{maxnodedegreeinG'}$, where VC' is the partial vertex cover of the explored section and G' is the unexplored graph. This algorithm considers vertices in decreasing order of vertex degree to find the optimum solution. This is because we can assume the vertices with the highest degree are guaranteed to allow a less number of vertices used to cover all the edges of the graph and solve the minimum vertex cover problem.

For clarification on notation used in our psuedocode, our frontier set, F, contains tuples representing subproblems of the form (vertex, state, (parentvertex, parentstate)). Additionally, the vertex cover solution, VC, contains tuples of the form (vertex, state) where the term state is defined by if the vertex is in the VC solution (denoted by a 1) or not (denoted by a 0).

4.1.3 Time and Space Complexity. The worst case for branch and bound would be that every possible solution is explored. In this case, the time complexity is O($2^{|V|}$), which represents the total number of possible scenarios and the number of iterations of the while loop since each iteration takes O(V+E) time to explore a subproblem of the search space.

The frontier set contains all candidate vertices that need to be explored. Therefore, the space complexity would be O(V+E) because of the 2V space needed for storing the vertices in the frontier set and E space for finding the minimum vertex cover solution.

4.1.4 Strengths and Weaknesses. Branch and Bound ensures that we get an exact solution since it can be exhaustive and try every possible solution. However, a caveat is its terrible time complexity.

4.2 Construction Heuristic

4.2.1 Description. Instead of choosing any remaining vertex when building the vertex cover, we assume that the best improvement comes from adding the vertex that has the most edges connected to it. Our heuristic therefore entails greedily adding vertices to the cover that have the highest degree. The graph is updated so that a vertex selected for the cover has its corresponding edges removed as well. The algorithm terminates when there are no uncovered edges left so it always produces a vertex cover, although it may be suboptimal.

To determine the approximation factor, we consider the worst case for the greedy heuristic, which would be a bipartite graph consisting of $L \cup R$, where L is a set of n vertices and R is a collection of sets $R_1, R_2, \ldots$ of vertices, where set R_i has n/i nodes in it. So, overall there are $\Theta(n \log n)$ nodes. We now connect each set R_i to L so that each vertex in R_i has i neighbors in L while no two vertices in R_i share any neighbors in common. Now, the optimal vertex cover is just the n vertices in L, but the greedy algorithm can first choose R_n then R_{n1}, finding a cover of total size $n \log n - n$. So the approximation factor is $\Theta(\frac{n \log n}{n})$ giving an approximation guarantee of $\Theta(\log V)$ where V is the number of vertices.

4.2.2 Pseudocode. Algorithm 2 outlines how the vertex of maximum degree is added to the cover. The graph is also updated as vertices in the cover are no longer considered, which entails decreasing the degree of all the neighbors that share an edge with the selected vertex.

Algorithm 1: Branch and Bound

```
input  : graph G = (V,E), cutoff
output : best MVC cost and solution, solution times
F = [(v, 0, (-1, -1)), (v, 1, (-1, -1))];
VC <- Ø ;
upperBound <– |V|;
gCopy <– copy of graph G;
v <– vertex with highest degree;
while F != Ø and time < cutoff do
    (vx, state, parent) = F.pop();
    backtrack = false;
    if state == 0 then
        for all neighbors of vs, set state = 1 ;
        VC <– neighbors of vx;
        remove neighbors from gCopy;
    else if state == 1 then
        remove vx from gCopy
    if G.edges == Ø // solution found then
        if |VC| < upperBound then
            optimal <– VC;
            upperBound <– |VC|;
        backtrack = true //explore next subproblem
    else
        if LowerBound(G) + |VC| < upperBound then
            vy <– current vertex with highest degree;
            F <– [(vy, 0, (vx, state)), (vy, 1, (vx, state))]
        else
            backtrack = true; //explore next subproblem
    if backtrack = true and F != Ø then
        nextParent = last element in F;
        if nextParent in VC then
            idx = idx of nextParent in VC;
            while idx < |VC| do
                node, state = VC.pop();
                add node and edges back to gCopy
        else
            reset VC <– empty;
            reset gCopy <– G
return optimal, solutionTime
```

Algorithm 2: Greedy Approximation

```
input  : graph G = (V,E), cutoff
output : best MVC cost and solution, solution times
C <– Ø;
while E ≠  do
    Pick highest degree vertex v in the current graph;
    C <– C + v;
    E <– E - {e ∈ E : v ∈ e}
return optimal, solutionTime
```

4.2.3 Time and Space Complexity. This algorithm takes at most $O(VE)$ time for each iteration to find the maximum degree vertex, remove it, and recalculate the degrees of the graph. There are at most V iterations, so the runtime complexity in the worst case is $O(V^2E)$. The space complexity is at worst $O(V)$ to keep track of the degrees of the vertices and to store the set of vertices that are the chosen cover.

4.2.4 Strengths and Weaknesses. This algorithm is guaranteed to output a complete vertex cover and is relatively fast. However, the greedy approach is naive and does not always find the minimum covering on even smaller instances. This is further reinforced by the fact that the heuristic does not have tight guarantees in optimality with an approximation ratio of $O(\log V)$.

4.3 Local Search - Hill Climbing

4.3.1 Description. The local search algorithm attempts to initialize a naive vertex cover solution and optimize it over time. With this framework, the algorithm solves the decision version of the problem which is "Given an integer k, is there a vertex cover of size at most k". This is done by iteratively looking at edges that have not been covered and exchanging vertices based on lower cost. While the vertices are evaluated based on overall edge cost, we incorporate random restarts with a random seed. This allows us to more thoroughly cover possibilities across the search space. The specific selection scheme, or way we choose the vertex to remove and add, can be defined by various heuristics. One recent paper developed a novel way to speed up the search by developing an algorithm called FastVC [9].

4.3.2 Pseudocode. The pseudocode for this algorithm can be seen in algorithm 3. The entire algorithm optimizes values using a loss and gain value as reference. Loss for a vertex v is defined as the number of covered edges that would become uncovered by removing vertex v. Gain for a vertex v is defined as the number of uncovered edges that become covered by adding vertex v. The overall algorithm is broken up into 2 parts, the initialization of the Vertex Cover and then the search optimization. We start by initializing a Vertex Cover using the ConstructVC algorithm (algorithm 4) which works by starting at a vertex s, exploring edges neighboring it and adding vertices with the higher degree to the cover. After going through all nodes, we then check our cover for vertices with loss=0 and remove those nodes as it would not affect the cover and reduce the quality. This is also our upper bound for the algorithm. The optimization works by using the Best from Multiple Selection Heuristic to select which vertex to remove. This heuristic works by "Choose k elements randomly with replacement from your cover and return the best one (with respect to some function)". The vertex to add is chosen by randomly selecting an uncovered edge and adding the vertex from it with higher gain.

Algorithm 3: FastVC

```
input  : graph G = (V,E), cutoff, random seed
output : best MVC cost and solution, solution times
VC <– ConstructVC(G) ;
gain(v) <– 0 for all v ∉ VC ;
random variable <– random seed;
while time < cutoff do
    if VC covers all edges then
        optimal <– VC;
        v <– vertex with minimum loss in VC;
        VC <– VC - v;
        continue;
    u <– random element from VC;
    for iteration 1 to k do
        u' <– random element from VC;
        if loss of u' < loss of u then
            u <– u';
    VC <– VC - u;
    if uncovered edges > 0 then
        e <– random edge;
        v <– endpoint of e with larger gain;
        VC <– VC + v;
return optimal, solutionTime
```

Algorithm 4: ConstructVC

```
input  : graph G = (V,E)
output : Vertex Cover of G
C <– Ø ;
foreach edge ∈ E do
    if edge is uncovered then
        C <– C + v where v is endpoint of edge with
          higher degree;
loss(v) <– 0 for all v ∈ C ;
foreach edge ∈ E do
    if only one vertex of edge is in C then
        loss(v) += 1;
foreach vertex ∈ C do
    if loss(v) = 0 then
        C <– C - v;
        update losses of C;
return C
```

4.3.3 Time and Space Complexity. The time complexity of this algorithm can be analyzed in 2 parts. The ConstructVC part runs within O(m) where m is the number of edges. The reason for this is that we iterate through all of m to generate a cover and then iterate through the cover (some subset of m) to remove unnecessary vertices. Thus at worst case this is O(2m)=O(m). The optimization part runs within O(k)=O(1) because choosing a vertex to remove requires selecting k vertices where k is constant. Similarly choosing a vertex to add is randomized and requires only selecting 1 edge. Thus the overall time complexity for this algorithm is O(m). The space complexity for this algorithm is about O(VE) as it requires information on the loss and gain values for every vertex, this can be optimized via various packages in python.

4.3.4 Strengths and Weaknesses. This strengths of this algorithm are that it optimizes very quickly. The weaknesses are that is heavily

dependent on the random seed and it utilizes a lot of space for maintaining graph data.

4.4 Local Search - Max Independent Set

4.4.1 Description. Another local search solution find the minimum vertex cover by solving its dual problem, the maximum independent set. An independent set of a graph G = (V;E) is a subset of V, M, whose elements are pairwise non-adjacent. Notice that finding the Minimum Vertex Cover C of a graph is equivalent to finding the Maximum Independent set M because VC = V - M. Using this theory, we can generate a maximum independent set of the vertices using a randomized search across the graph. Every time we find a larger independent set, we would have a more optimal vertex cover because it would be complementary to the number of vertices.

4.4.2 Pseudocode. The pseudocode for this algorithm can be found in algorithm 5. The maximum independent set problem can be optimized by local search using the 2-improvement method. This method works by taking a vertex of a maximal independent set and replacing it with two of its neighbors that are not adjacent to each other. This way, the total number of vertices in the solution is increased by one. Our pseudocode starts by maintaining 3 values for every vertex, whether it is covered, uncovered, and what the gain is. At the start of our loop, we check if we have a larger independent set and update our graph with that information. The using the largest independent set in our memory, we select a node at random to reconsider. This randomness works by selecting 5 random nodes and selecting the node with the smallest gain. We then update our set by removing the node and adding all of its neighbors that are valid and then recalculate the covered, and uncovered nodes as well as value. Finally, we make sure we have no open vertices left by iterating through the remaining open vertices and putting all neighbors into our independent set that make it valid. Our final vertex cover is the set of vertices in the graph that don't exist in the largest independent set we generated.

4.4.3 Time and Space Complexity. The time complexity of this algorithm is worst case O(V^2) because when generating the max independent set, we process the list of open vertices and check all their neighbors for whether they are covered or not. In the worst case, our list of open vertices is of size V and the neighbors are fully connected of size V as well. The update step runs within O(V) by comparison because we select a random vertex, O(1) operation, and then check all neighbors for whether we should swap it. The space complexity of this is the same as the other local search algorithm which is O(VE) because we must maintain a full version of the graph with covered, uncovered, and gain values for reference when generating max independent set.

4.4.4 Strengths and Weaknesses. The strengths of this algorithm are that it utilizes the random restart to its full capacity, allowing it to check most possible swaps. This results in it having good chance for finding an optimal solution given enough time. The downside is the runtime and how long it takes to generate a vertex cover.

Algorithm 5: MaxIndSetConversion

```
input  : graph G = (V,E), cutoff, random seed
output : best MVC cost and solution, solution times
VC <– ConstructVC(G) ;
random variable <– random seed;
uncov <– V - VC ;
close(v) <– 1 for all v ∈ uncovered else 0;
open(v) <– 1 for all v ∈ VC if neighbor is uncovered else 0;
val(v) += 1 for all v ∈ VC if neighbor is uncovered else 0;
currIndSet <– v for all v ∈ V if cov(v) is 1 ;
MaxIndSet <– Ø ;
while time < cutoff do
    if number of node in uncov > MaxIndSet then
        optimal <– V - uncov;
        MaxIndSet <– uncov;
        updated graph <– Graph;
        new open <– currIndSet;
    else
        uncov <– MaxIndSet;
        Graph <– updated graph;
        currIndSet <– new open;
    open vertices = V - uncov;
    u <– random element from uncov;
    for iteration 1 to k do
        u' <– random element from uncov;
        if val(u') < val(u) then
            u <– u';
    V <– neighbors of u;
    foreach v ∈ V do
        val(v) += 1;
        if close(v) then
            W <– neighbors of v;
            foreach w ∈ W do
                val(w) -= 1;
            remove v from uncov;
    foreach v ∈ V do
        W <– neighbors of v;
        foreach w ∈ W do
            if close(w) and val(v) = 0 and w ∉ currIndSet
              then
                add w to currIndSet;
    add u to uncov;
    while currIndSet has nodes do
        u <– randomly selected node from currIndSet;
        V <– neighbors of u;
        foreach v ∈ V do
            val(v) += 1;
            if open(v) and not close(v) then
                open(v) = 0;
                remove v from currIndSet;
        close(u) = 1;
        val(u) = 0;
        remove u from currIndSet;
        add u to uncov;
optimal <– V - MaxIndSet;
return optimal, solution time;
```

5 EMPIRICAL EVALUATION

5.1 Branch and Bound

5.1.1 Platform. Branch and bound was tested on all graphs on a device with the following configuration:

- Processor: 2.9 GHz Intel Core i5
- Memory: 8 GB 2133 MHz LPDDR3
- Language: Python
- System: Mac OS

5.1.2 Procedure and Criteria. We ran the branch and bound algorithm for each dataset with a cutoff time of 10 minutes or 600 seconds. The resulting performance was evaluated based on the time to find the optimal solution, the best minimum vertex solution, and the percentage relative error found within the cutoff.

5.1.3 Results. See Table 1 for a comprehensive evaluation of the branch and bound algorithm.

Table 1: Branch and Bound Results

Dataset	Time(s)	VC Value	RelErr(%)
jazz	0.256	158	0
karate	0.0039	14	0
football	303.87	94	0
as-22july06	345.089	3307	0.12
hep-th	100.78	3944	0.46
star	230.86	7374	6.8
star2	256.56	4697	3.4
netscience	5.029	899	0
email	1.93	605	1.9
delaunay_n10	262.11	736	4.7
power	36.24	2277	3.4

Since the branch and bound algorithm can explore up to the entire search space, the algorithm is known to take a long time to complete its search for the optimal solution. However, since our implementation of BnB considers vertices in decreasing order of vertex degree to find the optimum solution, the initial solution was found rather quickly. This lower bound was a good starting off point and was pretty close to optimal solution. As seen in Table 1, the algorithm finds an optimal solution for four of the eleven graphs. Additionally, the maximum percentage relative error found was for the star graph which had a relative error of 6.8%, but this is still reasonable considering we constrained the running time to 600 seconds.

5.2 Construction Heuristic

5.2.1 Platform. Construction heuristic was tested on all graphs on a device with the following configuration:

- Processor: 2.9 GHz Intel Core i5
- Memory: 8 GB 2133 MHz LPDDR3
- Language: Python
- System: Mac OS

5.2.2 Procedure and Criteria. We ran the constructive heuristic for each dataset with a cutoff time of 10 minutes or 600 seconds. The resulting performance was evaluated based on the time to find the optimal solution, the best minimum vertex solution, and the percentage relative error found within the cutoff

5.2.3 Results. See Table 2 for a comprehensive evaluation of the heuristic algorithm. Regardless of size, this approximation algorithm outputs a covering within the cutoff at a relatively quick time. However, the tradeoff with efficiency lies in the optimality of the solutions. In only a couple of cases does the greedy degree approach reach an optimal solution. Even with smaller graphs, suboptimal covers were found, although depending on the tolerance they may be acceptable coverings.

Table 2: Construction Heuristic Results

Dataset	Time(s)	VC Value	RelErr(%)
jazz	0.0029	160	1.26
karate	0.00011	14	0
football	0.00083	96	2.13
as-22july06	4.35	3312	0.27
hep-th	1.92	3947	0.53
star	4.50	7366	6.72
star2	3.63	4677	2.97
netscience	0.088	899	0
email	0.038	605	1.85
delaunay_n10	0.041	740	5.26
power	0.64	2272	3.13

5.3 Local Search

5.3.1 Platform. All local search algorithms ran and generated graphs on the following platform:

- Processor: 2.8 GHz Intel Core i7
- Memory: 8 GB 2801 MHz
- Language: Python
- System: Windows OS

5.3.2 Procedure and Criteria. We ran both local search algorithms for each dataset with a cutoff time of 10 minutes or 600 seconds. We ran both algorithms on each graph 10 times using random seeds 1-10. The resulting performance was evaluated based on the time to find the optimal solution, the best minimum vertex solution, and the percentage relative error found within the cutoff. We have also specifically provided QRTD, SQD, and box plots for the Power and Star2 graphs in the Appendix section below.

5.3.3 Results for Local Search - Hill Climbing. While local search algorithms are not guaranteed to search all possibilities, the hill climbing approach seems to work very well for these datasets. The relative errors are fairly low even for the larger graphs and the runtimes are manageable in comparison to Branch and Bound regardless of the random seeding. The only exception to this is the star2 graph which seems to have both high runtime and a relative error. The QRTD plots for this indicate that there is a exponential increase in optimal solutions the longer we run the algorithm for

Table 3: Hill Climbing Results (Average over 10 runs)

Dataset	Time(s)	VC Value	RelErr(%)
jazz	0.57	158	0
karate	0.00	14	0
football	0.08	94	0
as-22july06	43.83	3303	0
hep-th	69.20	3926	0
star	274.27	6942	0.0057
star2	597.50	4607	0.0143
netscience	0.00	899	0
email	13.67	594	0
delaunay_n10	120.43	703	0.0099
power	40.763	2203	0

and this may be a result of the optimization process. The use of random restarts for picking vertices in our cover is extremely useful as well as it prevents a majority of these graphs to avoid getting stuck in a local minimum. The QRTD plot for the power graph has the opposite trend where it seems to find a very close optimal solution early on, but has trouble optimizing after a threshold of about 0.8%. The SQD plots show a distinct increase in relative solution quality overtime for the Power graph. For the Star2 graph specifically, we see that the algorithm has about the same solution quality beyond the 6 min mark (360 seconds) and requires a lot more time to search for a better solution. Finally, we see a large gap in runtime as the Power graph optimizes within half a minute and the Star2 graph requires the full amount of time.

Table 4: Max Independent Set Results

Dataset	Time(s)	VC Value	RelErr(%)
jazz	2.8	158	0
karate	0	14	0
football	0.42	94	0
as-22july06	532.23	3310	0.0021
hep-th	506.21	3929	0.0008
star	573.14	7023	0.0175
star2	592.86	4822	0.0616
netscience	0	899	0
email	150.333	632	0.0639
delaunay_n10	432.18	703	0
power	551.85	2209	0.0027

5.3.4 Results for Local Search - Max Independent Set. The maximum independent set conversion looks to be a more convoluted way to solve the problem with overall greater relative error on the graphs and longer runtimes as can be seen by the table and the box plots. Computationally, this method is more time consuming and resource extensive. This can also be seen by the QRTD graphs where the number of valid solutions drastically increases in amount of time required to compute (ie. q^* = 1.1% to 0.5% for the power graph). There also tends to be less of a exponential growth in solution quality towards the beginning and is more linear. The SQD graphs also indicate a larger disparity in optimal solutions in comparison to runtimes. Overall this shows that the max independent set conversion relies more heavily on randomness and brute force than first local search method.

6 DISCUSSION

From our results, we see a clear distinction in output values and runtimes. The local search hill climbing method was the best out of the 4 methods, with the lowest average runtimes and lowest relative errors for all graphs. The local search maximum independent set conversion method has low relative errors as well, but had significantly larger runtimes. In this regard, the branch and bound algorithm did better than the maximum independent set method. Finally, the construction heuristic method did the worst with the highest relative error, but with the fastest runtimes out of the 4 methods. This output is expected as they match the expected time complexities of the algorithm. The only anomaly was the runtime comparison of maximum independent set vs. branch and bound as bnb manually checks the entire search space. This might be attributed to the difference in platform and system as well as estimation of the graph. The local search hill climbing method had the most optimal time complexity next to the construction heuristic, but incorporated Best from Multiple Selection heuristic to search the graph. Additionally, even with an exponential time complexity, the branch and bound algorithm still found promising solutions, some of which were were better than the construction heuristic search.

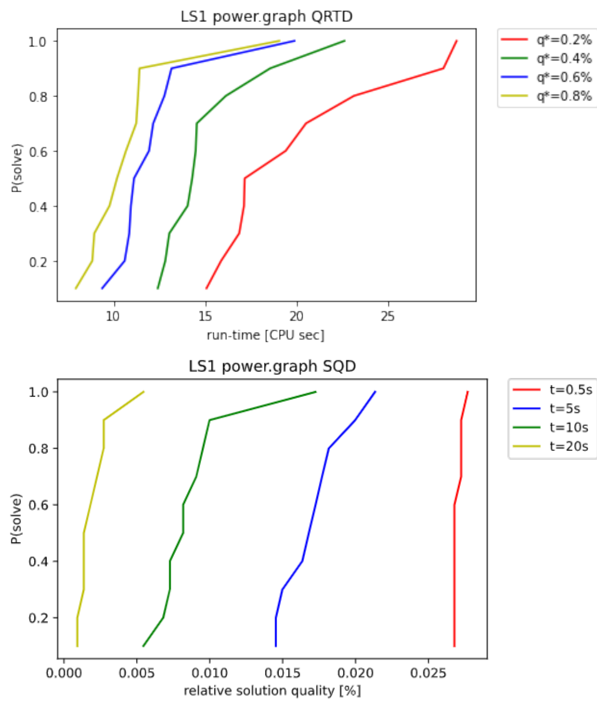
7 CONCLUSION

In conclusion, we see that the local search and branch and bound methods provide the best optimization for the minimum vertex cover problem. In contrast, the construction heuristic method provides a fast estimate in return for low accuracy. The usage of heuristics within local search proved to be more effective as it was able to avoid getting stuck in local minimum and utilize randomness to explore the search space more thoroughly. Meanwhile, the branch and bound method manually checked all possible solutions within the given time frame and so had the largest runtime of the 3 methods.

REFERENCES

[1] Karp, R.M. Reducibility Among Combinatorial Problems. Complexity of Computer Computation, Plenum Press, New York (1972) 85–103.
[2] Downey, R.G., Fellows, M.R.: Fixed Parameter Tractability and Completeness II: Completeness for W [1]. Theoretical Computer Science, Vol. 141 (1995) 109–131.
[3] F. Dehne et. al, Solving large FPT problems on coarse grained parallel machines, Available: http://www.scs.carleton.ca/fpt/papers/index.htm
[4] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the theory NP - completeness, San Francisco: Freeman (1979).
[5] Dinur, Irit; Safra, Samuel (2005). "On the hardness of approximating minimum vertex cover". Annals of Mathematics. 162 (1): 439–485.
[6] C. Z. Tang, X. Xu et al., An algorithm based on Hopfield network learning for minimum vertex cover problem, Lecture Notes in computer science, Vol. 3173, (2004), 430 - 435,.
[7] X. Xu and J. Ma, An efficient simulated annealing algorithm for the minimum vertex cover problem, Neurocomputing, Vol. 69, Issues 7-9, (2006), 613 - 616.
[8] S.J. Shyu, P.Y. Yin and B.M.T. Lin, An ant colony optimization algorithm for the minimum weight vertex cover problem, Annals of Operations Research, Vol. 131, (2004), 283 - 304.
[9] Cai, Shaowei. “Balance between Complexity and Quality: Local Search for Minimum Vertex Cover in Massive Graphs.” IJCAI (2015).

8 APPENDIX



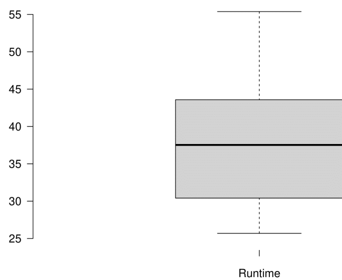


Figure 2: LS1: Power Graph Results 1) QRTD Plot for q* values = [0.2%, 0.4%, 0.6%, 0.8%] 2) SQD Plot for t values in seconds = [0.5, 5, 10, 20] 3) Box plot for Runtimes in seconds of Power graph

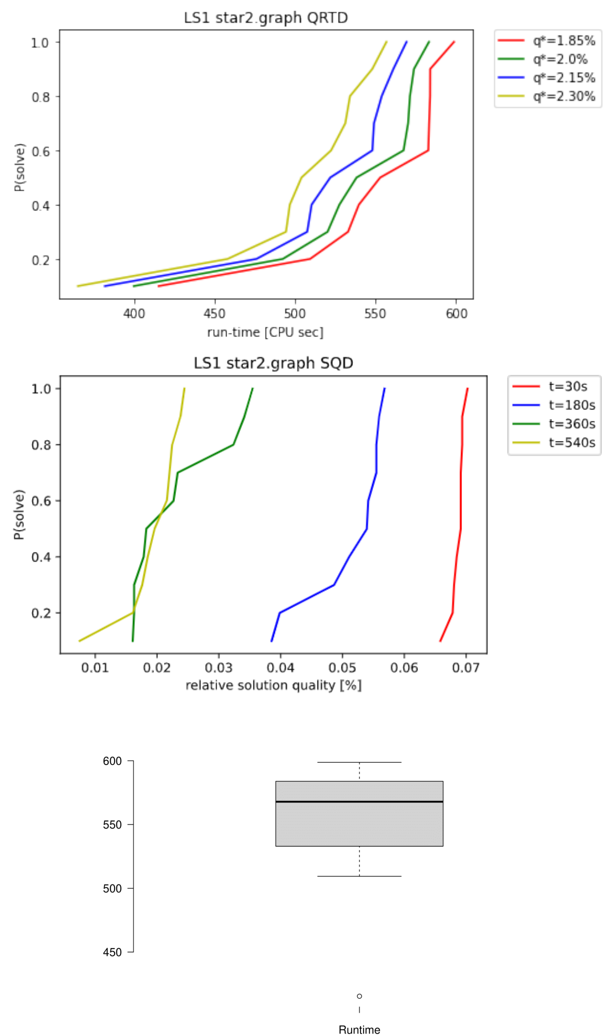
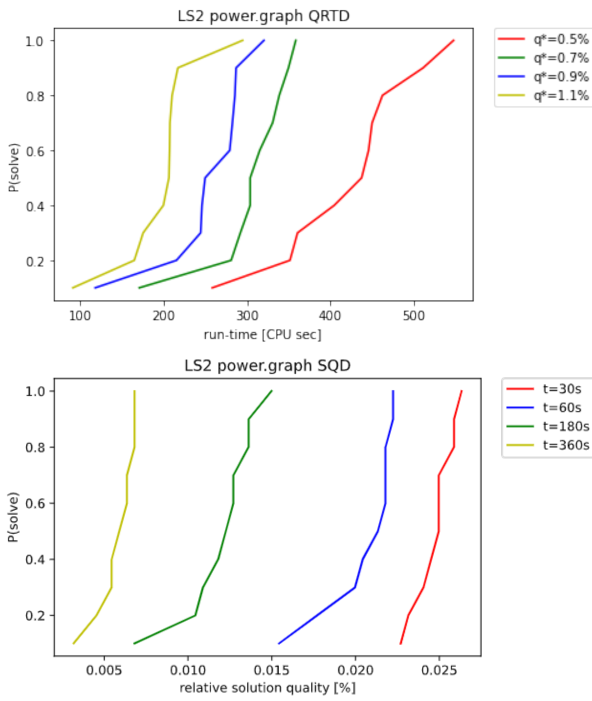


Figure 3: LS1: Star2 Graph Results 1) QRTD Plot for q* values = [1.85%, 2%, 2.15%, 2.3%] 2) SQD Plot for t values in seconds = [30, 180, 360, 540] 3) Box plot for Runtimes in seconds of Star2 graph



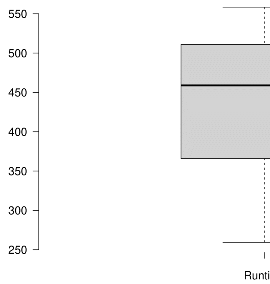
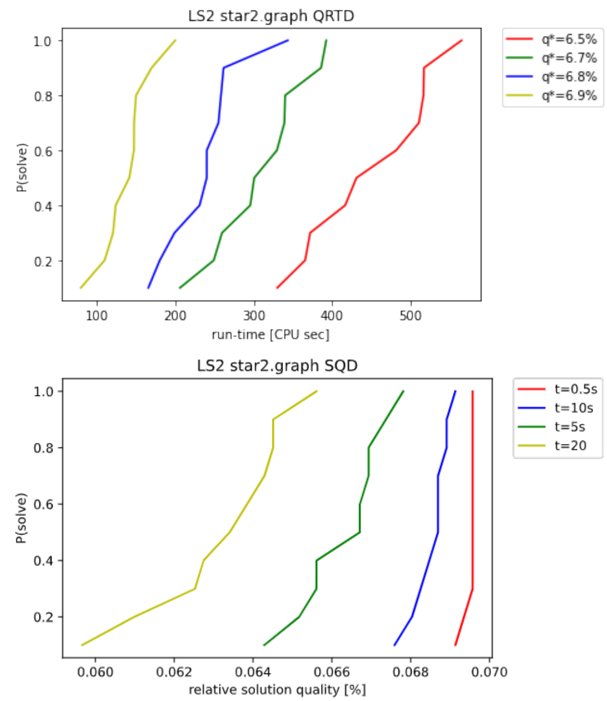


Figure 4: LS2: Power Graph Results 1) QRTD Plot for q* values = [0.5%, 0.7%, 0.9%, 1.1%] 2) SQD Plot for t values in seconds = [30, 60, 180, 360] 3) Box plot for Runtimes in seconds of Power graph



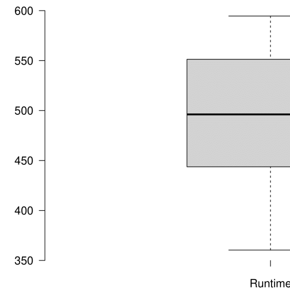


Figure 5: LS2: Star2 Graph Results 1) QRTD Plot for q* values = [6.5%, 6.7%, 6.8%, 6.9%] 2) SQD Plot for t values in seconds = [0.5, 5, 10, 20] 3) Box plot for Runtimes in seconds of Star2 graph