CSE6140 Fall 2020 Project Minimum Vertex Cover

Alex Gurung
Georgia Institute of Technology
Atlanta, USA
agurung@gatech.edu

Shasha Liao
Georgia Institute of Technology
Atlanta, USA
sliao7@gatech.edu

Wai Man Si
Georgia Institute of Technology
Atlanta, USA
wsi33@gatech.edu

Haley Xue
Georgia Institute of Technology
Atlanta, USA
hxue42@gatech.edu

ACM Reference Format:
Alex Gurung, Shasha Liao, Wai Man Si, and Haley Xue. 2018. CSE6140 Fall 2020 Project Minimum Vertex Cover. In *Proceedings of Minimum Vertex Cover (CSE 6140 Final Project).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Minimum Vertex Cover (MVC) problem is a well-known NP-complete problem which has applications in numerous areas such as computational biology, operations research, and the routing and management of resources. In this paper, we study the MVC problem using four different algorithms: an exact algorithm using Branch-and-Bound, an approximation algorithm using maximum degree greedy approach [Delbot and Laforest 2010], and two local search algorithms using simulated annealing and genetic algorithms. The Branch-and-Bound algorithm tries to explore the whole search space while removing unnecessary solution candidates by keeping track of upper bounds and calculating lower bounds for partial vertex covers. It can be slow for large graphs but optimal solution is ensured given enough time. The approximation algorithm sacrifices the quality of the solution for speed. It quickly finds a solution with quality no bad than twice of the optimal solution. Local search algorithms searches for the best solution in a subset of the whole search space. We implemented two strategies, Simulated Annealing and Genetic Algorithms, to explore multiple mechanisms to explore the search space. For the graphs tested we found the most success with the Simulated Annealing approach and the least with the Genetic approach, highlighting the importance of problem-specific design decisions. Both Branch-and-Bound and the Approximation algorithm performed well, but would either take too long or settle for a non-optimal solution.

2 PROBLEM DEFINITION

A vertex cover for a graph $G = (V, E)$ is a subset of a vertices $V' \subseteq V$ such that for every edge $(u, v) \in E, u \in V' \vee v \in V'$. In other words, every edge has a least one endpoint that is in V'. The minimum vertex cover is the subset of V that satisfies the aforementioned condition with the least number of vertices (i.e. a vertex cover of the smallest size). The MVC problem is a well-known NP-complete problem, and we will discuss several algorithms that we have implemented to solve this problem in this paper.

CSE 6140 Final Project, Fall 2020, Atlanta, GA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

3 RELATED WORK

In graph theory, MVC is a classical NP-hard optimization problem in computer science. The problem becomes intractable as the number of nodes grows. Researchers proposed various approaches to solve the problem such as branch and bound, approximation, and local search.

Branch and bound (BnB) is used to search the enumeration of candidate solutions and return the exact solution ([Bansal and Rana 2014]). ; however, it could take really long time to find the solution. Later, Pajariene demonstrates the performance of BnB under different setup and points out that BnB can only finish within a short amount of time when the graph is small ([Pajarinen 2007]).

Alternatively, scientists attempt to construct heuristics with approximation guarantees to alleviate the issue. The algorithm can be finished in polynomial time and return a solution that quality is with a specific approximation ratio [Delbot and Laforest 2010]. Also, if an approximation algorithm exists, we guarantee that the running time is expected and the quality of the solution is bounded via the analysis of the algorithm (using an approximation ratio).

Furthermore, local search is another possible approach and it is based on the oldest optimization method: trial and error. Local search algorithms move from solution to its neighbor solutions in the space of candidate solutions, then we exchange part of the solution until the quality close to the optimal or a cut-off time has been reached. Cai proposed NuMVC two-stage exchange and edge weighting with forgetting to reduce the running time of the existing method and optimize current implementation and outperform state-of-the-art heuristic algorithms. [Cai et al. 2012]

4 ALGORITHMS

4.1 Branch-and-Bound

4.1.1 Description. The Branch-and-Bound strategy finds the exact solution by exploring the whole search space. It eliminates some unnecessary cases by keeping track of the best solution found so far and computing a lower bound on the size of the vertex cover for all extensions of the partial vertex cover. Using these bounds, the

algorithm rules out certain "nonpromising" partial vertex covers, or decision nodes, to prune the decision tree and guides the search direction as a measure of "promise". In our implementation, we also utilize the number of uncovered edges as a measure and the experiments showed that it navigates to solutions much faster than just use the lower bound.

Algorithm 1 Branch-and-Bound

Input: $G = (V, E), cutoff$
Output: Set of vertices in vertex cover VC
Initialize opt_cover and opt_num with results from a 2-approx algorithm
Initialize $LB \leftarrow opt\_num/2$
$v \leftarrow$ the vertex in G with the highest degree
$pqueue \leftarrow [(n_0, node(v, None, LB, 0)), (n_1, node(v, None, LB, 1))]$ { n_0 and n_1 are the number of uncovered edges, i.e., $|E'|$ for $C' = \{\}$ and $C' = \{v\}$ respectively; $node$ is an object with properties $(vertex, parentnode, lowerbound, state)$. }
while $pqueue \neq \phi$ and running time < $cutoff$ **do**
  $n, Dnode \leftarrow$ smallest $|E'|$, node in $pqueue$ with smallest LB
  $C' \leftarrow$ partial VC, the explored vertices with $state$ 1
  $Explored \leftarrow$ the explored vertices with $state$ 0
  $G' \leftarrow$ uncovered graph
  $v\_next \leftarrow$ vertex in $G' \backslash Explored$ with the highest degree
  if no v_next found **then**
    continue
  end if
  $C' \leftarrow C' \bigcup \{v\_next\}$
  $G' \leftarrow G'$ with vertex v_next and its edges removed
  if $G'.vertices = \phi$ **then**
    if $|C'| < opt\_num$ **then**
      $opt\_num, opt\_cover \leftarrow |C'|, C'$
    end if
    continue
  end if
  $LB \leftarrow |C'| + A(G')/2$ {defined above in section 4.1.2}
  if LB < opt_num **then**
    add $node(v\_next, Dnode, LB, 1)$ to $pqueue$
  end if
  if $node(v\_next, Dnode, Dnode.LB, 0)$ is not a dead end and $Dnode.LB < opt\_num$ **then**
    add $node(v\_next, Dnode, Dnode.LB, 0)$ to $pqueue$
  end if
end while
return opt_cover

4.1.2 Algorithm. Given a graph $G = (V, E)$ and a partial vertex cover C', the lower bound is computed as $LB = |C'| + A(G')/2$ with $G' = (V', E')$ being the subgraph of G not covered by C', $V' = V - C' - \{v \in V | \forall (v, u) \in E, u \in C'\}$ and $E = \{(u, v) \in E | u, v \in V'\}$. Here $A(G')$ is the quality of a 2-approximation algorithm on G'. The algorithm selects vertices in decreasing order of their degrees in the uncovered graph G'. A binary decision tree was constructed to explore the whole search space by assigning each vertex a state of 1 or 0 to indicate if it is included in the candidate vertex cover solution or not respectively. A node is determined to be a dead end if the partial vertex cover excludes two vertices u, v with $(u, v) \in E$.

4.1.3 Time and Space Complexity. Since each vertex can be selected to be included in a vertex cover or not, there are a total of $2^{|V|}$ decision nodes in the whole binary decision tree. To choose, expand and check each node, it takes a time of $O(|V| \log(|V|) + |E|)$. Overall, our algorithm has the worst case time complexity $O(2^{|V|}(|V| \log(|V|) + |E|))$. The worst case space complexity will be $O(2^{|V|} + |E|)$ as we have $2^{|V|}$ decision nodes to store and $O(|V| + |E|)$ space is needed to store the graph and candidate vertex cover solution.

4.2 Approximation Algorithm

4.2.1 Description. For the approximation algorithm, we adapted the maximum degree greedy approach outlined in [Delbot and Laforest 2010].

We also tried various other approaches including maximal matching [Sanjoy Dasgupta 2006] and greedy independent cover [Delbot and Laforest 2010]; however, we found that simplicity of the maximum degree greedy approach allowed for a good compromise between runtime and accuracy. The idea behind the approximation algorithm, which is based on the traditional greedy approach for solving the set cover problem, is quite simple: while there are still edges that are uncovered by the selected vertices in the current vertex cover, V', select the next vertex $v \in V \backslash V'$ that has the maximum degree (V is the set of all vertices in graph G) and add it to V'.

The worst-case Approximation ratio guaranteed by the maximum degree greedy algorithm is $H(d)$ where d is the maximum degree of the graph and $H(n)$ is the harmonic function. We omit the full proof here as a detailed proof can be found in [Kleinberg and Tardos 2005].

4.2.2 Algorithm. Graphs are parsed and stored using the networkx Python library as an adjacency list. In our first iteration, we used the *remove_nodes* function from the networkx library, but found that it ran slower when compared to our final version. To optimize the runtime of the algorithm, instead of removing nodes in each iteration, we simply updated the degree of the current maximum degree vertex to 0 (indicating it has been removed from the graph) and updated the degrees of the neighboring vertices accordingly. The difference in runtime may be attributed Python's methods of managing memory (i.e. deleting nodes takes longer than just updating entries).

Algorithm 2 Maximum Degree Greedy

Input: $G = (V, E)$
Output: Set of vertices in vertex cover VC
$VC \leftarrow \emptyset$
while edges $\neq \emptyset$ **do**
  $u \leftarrow$ vertex with $\arg\max_{v \in V} deg(v)$
  $V \leftarrow V - \{u\}$
  $VC \leftarrow VC \cup \{u\}$
end while
return VC

4.2.3 Time Complexity and Space Complexity. The while loop has worst case $O(|V|)$ iterations. Each iteration has a for loop that has in the worst case $O(|V|)$ iterations and updating the degrees of the neighbors takes $O(|V|)$; therefore the for loop takes $O(|V|^2)$. Updating u requires $O(|V|)$ time. The overall time complexity of our approximation algorithm is therefore $O(|V|^3)$.

Storing the degrees of each vertex takes $O(|V|)$ space and the storing of the vertex cover takes $O(|V|)$. The graph itself is stored as an adjacency list with takes $O(|V| + |E|)$ space. Thus the overall space needed is $O(|V| + |E|)$.

Algorithm 3 Maximum Degree Greedy (Adapted)

```
Input: G = (V, E)
Output: Set of vertices in vertex cover VC
VC ← ∅
degrees ← dictionary of vertices and corresponding degrees
u ← vertex with max degree
while deg(u) > 0 do
    degrees(u) ← 0
    VC ← VC ∪ {u}
    for each neighbor n of u do
        degrees(n) ← degrees(n) − 1
    end for
    u ← vertex with max degree
end while
return VC
```

Algorithm 4 Simulated Annealing

```
Input: G = (V, E), initial solution S_init, cutoff
Output: Set of vertices in vertex cover VC (S_return)
T = 0.8
S_best ← S_init
S_return ← ∅
while runningtime < cutoff do
    T = T * 0.95
    if S is a VC then
        S_return ← S
    end if
    randomly delete a vertex {u} ∈ S
    randomly add a vertex {u} ∉ S
    if |S| < |S_best| then
        p = exp(|S|−|S_best|) / T
        α = uniform sampling from [0, 1]
        if α > p then
            S_best ← S
        end if
    end if
end while
return S_return
```

4.3 Local Search: Simulated Annealing

4.3.1 Description. Simulated Annealing (SA) is used to approximate the global optimal of the target function, it is proposed based on the idea of the physical process of annealing [Kirkpatrick et al. 1983] In the beginning, there is a wider search space to be explored when the temperature is high, and is then narrowed down to several likely spaces as the temperature decreasing. This prevents the algorithm from being trapped in local solutions by introducing randomness with the idea of Monte Carlo sampling. The algorithm is implemented as follows: First, the initial solution is generated by removing all the unnecessary vertices. Second, the initial solution is modified by randomly removing a vertex that is in the current solution and randomly adding a new vertex that is not covered by the current vertex cover. Third, the new solution is either accepted or denied based on the probability which is proportional to the difference between the quality of the best solution and the current solution, and the temperature. Initial temperature is 0.8 and the temperature decreases at a rate of 0.95 [Ben-Ameur 2004]. Simulated annealing can be used to find a relatively good solution within a short amount of time, and it can be done via parallel computing which can increase the performance significantly. On the other hand, the optimal solution cannot be guaranteed, but the sub-optimal solution usually is good enough in most cases.

4.3.2 Time Complexity and Space Complexity. Given a graph G(V,E) where V is nodes and E is edges, time complexity is $O(|V|)$, since we search for all the neighbor of a node when the temperature is high and that's the worse case. The space complexity is $O(|V|+|E|)$ since we store the information of the graph.

4.4 Local Search: Genetic Algorithm

The second local search algorithm we attempt takes a genetic approach. The main potential benefit of this approach is it avoids stalling in local maxima and builds upon successful solutions. However, this approach lacks a guarantee of optimality and has the possibility of ignoring nearby improvements in the search space in favour of exploration.

4.4.1 Description. Our representation of an individual follows standard genetic algorithm procedure and is a binary string of length $|V|$ where a 1 in the i-th index indicates we include the i-th vertex in our candidate.

As the given problem values increasing good *valid* solutions, we select our initial population with an emphasis on validity. We always include an all 1-s individual in our initial population (corresponding to including all vertices in our vertex cover), and create new individuals by randomly removing vertices from that complete individual or specifically removing one from each individual. The number of individuals in the population is fundamentally a hyperparameter we denote P, but to allow the algorithm to incorporate a larger search space at a time as the size of the graph increases, our selection of P scales linearly with V with a cap of 1000.

While possibly resulting in a more complex problem to solve, we chose to iteratively produce better solutions by way of a Las Vegas Algorithms approach as it fit our requirements better. As such, we return an 'invalid' score marker if the individual is improperly formed or if it is not a valid vertex cover (that is, if it does not touch all edges in G) and then score the individual $|V| - I_V$ where I_V is the number of vertices this individual contains. We aim to maximize

this value, and as $|V|$ is constant this means reducing I_V as much as possible.

The second set of design choices for this algorithm concern the selection of new individuals. We select the 'parents' from the previous generation through tournament selection and mate individuals through two-point crossover on their binary string representations. We then perform mutations on these individuals by flipping random bits and by shuffling indices around. Both of these operations, mating and mutation, occur with a random chance (by default, 70% chance of mating and 20% chance of mutation).

As is standard procedure for genetic algorithms, we continue this selection and pruning process until time expires or a set number of generations is reached. For our purposes our default stagnation criteria is either 600 seconds or 1000 generations (whichever occurs first).

Algorithm 5 Genetic Algorithm

```
Input: G = (V, E), cutoff, genCutoff,
       probCrossover, probMutation
Output: Candidate Minimum Vertex Cover VC
P ← generate initial population
gen = 0
while runningtime < cutoff, gen < genCutoff do
  Select parents P for next population
  for each parent p of P do
    if random() < probCrossover then
      performCrossover()
    else
      if random() < probMutation then
        performMutation()
      end if
    end if
  end for
  P = selectNextGeneration()
  gen+ = 1
end while
return VC
```

4.4.2 Time Complexity and Space Complexity. The time and space complexities are primarily dependent on the hyper-parameters selected. We denote the maximum number of generations N and the size of the population P. The outside loop has a time cutoff, but on small values is limited by the number of generations. The internal loop (as well as the selection process) are both limited by the number of individuals in the population. The crossover and mutation operations are limited in time complexity by the length of the individual and are therefore $O(|V|)$ (the mutation loops over the indices and flips random bits, for example).

Putting the pieces together, the time complexity of the algorithm on small values of N (relative to the cutoff time) is $O(N \cdot P \cdot |V|)$ and the time complexity for large numbers of generations is $O(N \cdot |V|)$.

The space complexity of this algorithm is $O(P \cdot |V| + |E|)$, as we store P number of individuals of length $|V|$ as well as the graph itself.

5 EMPIRICAL EVALUATION

5.1 Setup

All the algorithms were implemented in Python 3.7, and were tested on the same PC, which has a 3.5 Ghz Intel 12 processor and 64 GB RAM to compare the result fairly.

The experiment is conducted on datasets extracted from the 10th DIMACS challenge. Also, randomness is involved in our experiment, so both local search algorithms are run with 10 different random seeds in order to reduce the bias and subsequent results were averaged.

The experiment is measured based on the time of getting the optimal solution or the best candidate solution given a cut-off time which is 600 seconds. The running times and associated sizes of minimum vertex covers found for all the graph instances are presented in Table 1.

5.2 Branch-and-Bound Results

Our Branch-and-Bound algorithm was able to explore the whole search space and find the optimal solution for small graphs. For large graphs, it also did a very good job in finding solutions within a maximum of 6.8% relative error in 10 minutes. It performed well because the partial vertex cover which covers the most edges was considered first and the vertex with the greatest degree in the uncovered graph was chosen with preference. It computes the lower bound of each uncovered subgraph in linear time and uses it to eliminate "nonpromising" decision nodes. Moreover, our algorithm is able to quickly detect dead ends and prune it immediately to avoid investigating unnecessary cases. The performance could be further greatly improved by simplifying the procedure of finding graph G' for partial vertex cover C'. However, we ran out of time to test the algorithm again since all algorithms should be tested on the same computer.

We reuse the graph G' from the previous step if the current decision node is a child of the previous one. Otherwise, we trace back to the root to find the partial vertex cover C' and construct a new graph G'. We believe that the performance can be further improved if we design a dynamic way to modify the graph G' by finding the common ancestor of the current decision node and the previous decision node.

5.3 Approximation Results

Our maximum degree greedy algorithm performed well in comparison to the other algorithms and was able to find a VC with a small relative error quickly. In the worst case, the approximation algorithm found a solution with a 7% relative error in 3.68 seconds on the *star* graph. Within our time cutoff of 10 minutes, the MDG algorithm achieved accuracies very close to our Branch-and-Bound algorithm in a fraction of the time. As expected, the Simulated Annealing local search algorithm was able to consistently find solutions closer to the optimal than MDG. The Approximation algorithm performed well because the graphs given were relatively sparse and ran faster than the theoretically implied runtime.

We compared the vertex cover outputs by the approximation algorithm with the solutions guaranteed by the theoretical worst-case approximation ratio and found that our runs were well within the

bounds, and were, in fact, even well within a 2-times approximation bound.

Graph	Max Degree	Optimal	Approx	H(n)*OPT
football	12	94	96	292
delaunay_n10	12	703	740	2182
karate	17	14	14	48
power	19	2203	2272	7816
netscience	34	899	899	3702
hep-th	50	3926	3947	17664
email	71	594	605	2879
jazz	100	158	160	820
star2	1531	4542	4677	35933
star	2109	6902	7366	56813
as-22july06	2390	3303	3312	27601

Figure 1: Maximum Degree Greedy Algorithm Results

5.4 Local Search (SA) Results

SA consistently performed the best in terms of relative error compared to the other implemented algorithms. It was able to achieve optimal results for almost all graphs with non-optimal solutions only beginning to appear on larger graphs.

It is interesting that the relation between the running time and vertex cover isn't linear or exponential; a possible reason could be that the performance also depends on the structure of the graph. If there are too many edges, it could take a really long time even the graph is small; vice versa, if there are only a few edges, it could be finished within a few seconds even there is a lot of nodes. For QRTD, 0.05%, 0.1%, 0.2%, 0.5%, 1% are selected for power, and 0.1%, 0.2%, 0.5%, 1%, 2% are selected for star2. We can see that SA is able to achieve a high-quality solution within a short amount of time even when the graph is large. The result also suggests that we can stop the algorithm earlier instead of spending too many resources for a minor performance gain. It is also proved that when we look at SQDs, 10s, 25s, 50s, 75s, 100s is selected for power, and 100s, 200s, 300s, 400s, 500s is selected for stars. The results show that it takes more time to find a better solution when we have a high quality in the early stage, and it could indicate that we have a high-quality initial solution. Since SA is an algorithm involving randomness, we can inspect the effect of randomness via boxplot. Indeed, the performance is fluctuated when we used different random seed, but they are relatively close in the most of time.

5.5 Local Search (GA) Results

As explored further in 5.5.1, the Genetic Algorithm Local Search approach performed the worst out of our selections, producing increasing relative errors (in addition to producing large average times) as the size of the graphs increased. Many of the failings of this approach on this problem are especially evident in comparing its results on the *power* and *star2* to that of the other local search algorithm: Simulated Annealing. In addition to significantly better solution quality across the two graphs as shown in Table 1, the Solution Quality Distributions (SQDs) and Qualified Run-Time Distributions (QRTDs) also show Simulated Annealing approaches its best solution in a better way.

Comparing Figure 3 and Figure 2, we see that over time Simulated Annealing is able to iteratively improve on solutions in the search space leading to an increasing but more gradual improvement across solution quality percentages. In contrast, the Genetic Algorithm approach is characterised by large spikes in quality, indicating its reliance on random phenomena to discover better solutions. This pattern is also present in Figures 7 and 6, where the Genetic Algorithm's QRTD exhibits essentially zero separation between solution qualities over time indicating that it failed to improve on its initial solution. In contrast the Simulated Annealing QRTD for the *star2* dataset sees the likelihood of a given solution quality increase iteratively over time.

One possible benefit to the Genetic Algorithm approach present in the SQDs is the surprising consistency with which solutions of a given quality where found with respect to time as indicated by the vertical lines (this pattern was replicated with smaller time increases as well). However, the worse solution quality far outweighs this benefit in practice.

We also plotted the running times of the two algorithms in Figures 10 and 11, where we see the Genetic Algorithm took significantly longer and had a much smaller deviation in time. This is most likely due to two things: the inability to determine when each algorithm has reached an optimal solution, and the Genetic Algorithm's poor solution quality. The Genetic Algorithm appears to continually reach its time cutoff before quitting due to other stagnation parameters, and could not achieve optimal results before then due to its having trouble efficiently exploring the search space.

5.5.1 Analysis of Genetic Algorithm Performance: The Genetic Algorithm approach under-performed on a variety of datasets, and consistently had the highest relative error compared to other approaches. We believe that this is largely due to the construction of the algorithm itself, and that while improvements could be made to the algorithm's specifics there are over-arching problems with the choice for this problem.

The primary advantage of using a genetic algorithm is the ability to, in a parallel fashion, search many different parts of a search space at a given time. Combining and mutating individuals allows the algorithm to build on good candidate solutions and explore promising areas. However, for this problem the graphs tended to be sparse and thus random generation of individuals was unlikely to produce a small vertex cover. Similarly, combining individuals would be unlikely to produce a better *valid* vertex cover than before. Whereas other algorithms have the ability to intelligently select paths and improve on candidate solutions, the genetic approach relies heavily on randomization and as a result has difficulty in sparse search spaces.

A potential improvement to these problems would be to reverse the cost function and minimize edges not-included in a candidate solution in addition to minimizing the number of vertices. This reduces our reliance on valid individuals (as previously invalid vertex covers were discarded for their low fitness) and allows better searching of the search space. This change doesn't completely fix all of the problems described above, but may better utilize the strengths of the genetic approach. The reason this design decision wasn't made, as discussed before, was the over-arching requirement of a

guaranteed valid solution. By changing the cost function we lose this guarantee as the best individual may not be a valid vertex cover.

6 DISCUSSION

The Simulated Annealing local search algorithm performed the best out of the four algorithms in terms of accuracy and runtime. It was able to find a vertex cover very close to the optimal solution within our time cutoff of 10 minutes.

The Branch-and-Bound algorithm was able to achieve accuracies generally better than those of the Approximation algorithm for small graphs because it systematically traverses through the entire search space. However, the Brand-and-Bound had higher relative errors for larger graphs because each decision node took a longer time to search through. We see this in our empirical results —the Branch-and-Bound takes much longer to run as the graphs become larger because it has an exponential runtime complexity. The Approximation algorithm performed well because the graphs given are relatively sparse, and the inner for loop over the neighbors of the vertex with the maximum degree had much fewer iterations in our experiments than the theoretical worst-case runtime of $O(|V|)$. We would expect the performance of the approximation algorithm to deteriorate as graphs become more dense.

Interestingly, our two local search algorithms, Simulated Annealing and a Genetic Algorithm, performed the best and the worst respectively in terms of relative error. While both rely heavily on random processes, it's important to note the role that informed iterative improvement played on SA's performance. Whereas the genetic approach emphasized exploring many points on the search space in a parallel fashion, SA improved upon a single candidate solution. This reduced both the theoretical space and time complexity of each update, which was validated in our experiments by GA's significantly longer average times and displayed in Figures 10 and 11. While neither algorithm has a theoretical guarantee of optimality, empirically we found that SA produced significantly better results than our GA implementation.

7 CONCLUSION

In this report, we designed and implemented four Las Vegas Algorithms to solve the MVC problem. These four algorithms represented 3 different strategies: Branch-and-Bound (BnB), Approximation, and Local Search. We implemented two Local Search algorithms, Simulated Annealing and a Genetic Algorithm, and compared these implementations more closely.

For further exploration in the implementations of local search algorithms, we also make some recommendations:

(1) Further explore existing algorithms such as random hill climbing or Tabu search.
(2) We found initialization to be an important factor in performance, so exploring different initialization strategies may produce better results.
(3) Our neighbor exchange in SA only swaps one vertex, but this could be extended.
(4) The cost function in our GA implementation could be improved depending on user requirements (as explained further in our discussion).

The algorithm that performed the best in our tests was Simulated Annealing, although it is important to note that depending on user requirements different algorithms may be better choices. For example, while local search algorithms may be a better option to explore if you are trying to balance between performance and resources, if you don't require optimality, an approximation algorithm may provide a 'good enough' solution in significantly less time.

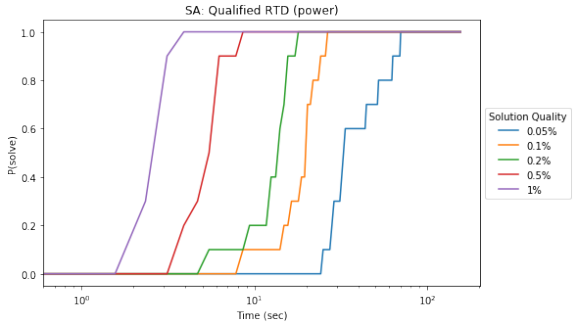

Figure 2: Simulated Annealing Qualified Runtime Distribution on Power Graph

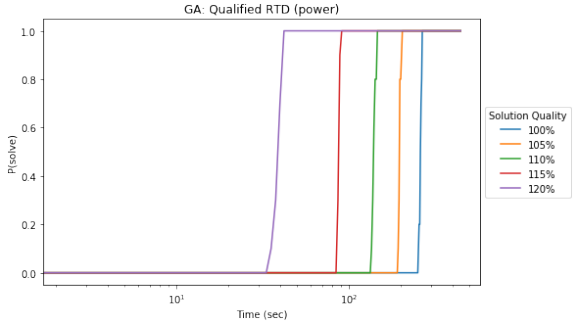

Figure 3: Genetic Algorithm Qualified Runtime Distribution on Power Graph

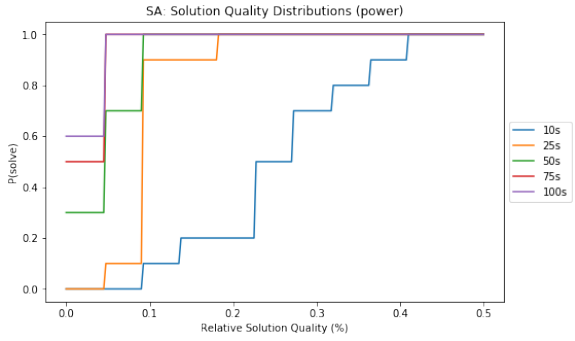

Figure 4: Simulated Annealing Solution Quality Distribution Graph on Power Graph

	BnB			Approx			SA			GA		
Dataset	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr
as-22july06	154.77	3309	0.0018	3.33	3312	0.0027	38.63	3303	0.0000	598.98	22652	5.8579
delaunay	159.36	734.3	0.0445	0.04	740	0.0526	75.60	703	0.0006	97.44	774	0.1016
email	2.58	605	0.0185	0.04	605	0.0185	4.14	594	0.0000	111.72	751	0.2641
football	0.05	95	0.0106	0.00	96	0.0213	0.02	94	0.0000	13.63	97	0.0330
hep-th	112.45	3944	0.0046	1.31	3947	0.0053	90.26	3926	0.0000	598.23	6853	0.7456
jazz	77.18	158	0.0000	0.01	160	0.0127	0.06	158	0.0000	25.02	164	0.0386
karate	0.00	14	0.0000	0.00	14	0.0000	0.00	14	0.0000	5.60	14	0.0000
netscience	3.53	899	0.0000	0.06	899	0.0000	0.05	899	0.0000	134.43	1013	0.1269
power	50.64	2276	0.0331	0.49	2272	0.0313	85.36	2203	0.0000	438.90	4129	0.8744
star	430.44	7374	0.0684	3.68	7366	0.0672	574.33	6963	0.0089	598.68	10517	0.5238
star2	266.43	4690	0.0326	3.10	4677	0.0297	483.21	4548	0.0013	598.84	13691	2.0144

Table 1: Comprehensive result

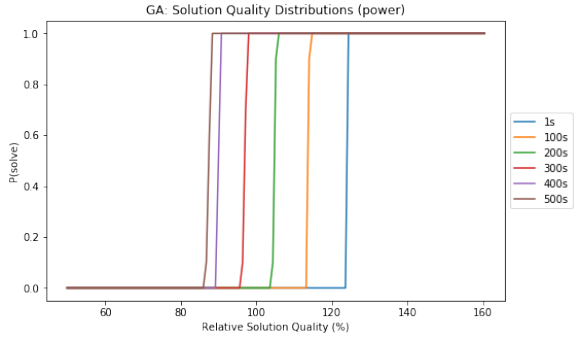

Figure 5: Genetic Algorithm Solution Quality Distribution Graph on Power Graph

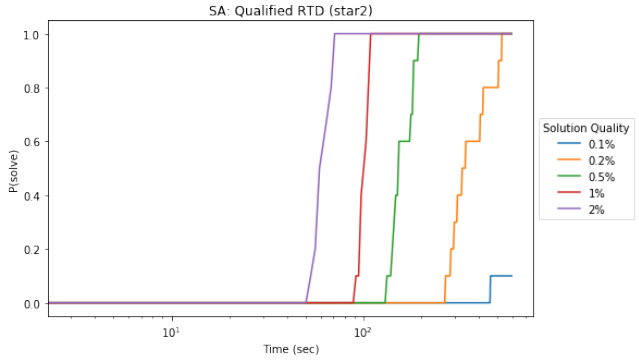

Figure 6: Simulated Annealing Qualified Runtime Distribution on Star2 Graph

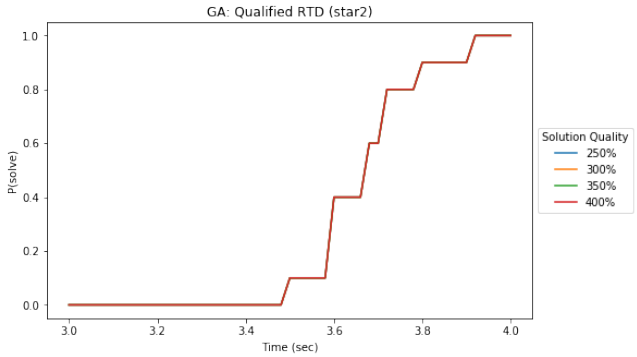

Figure 7: Genetic Algorithm Qualified Runtime Distribution on Star2 Graph

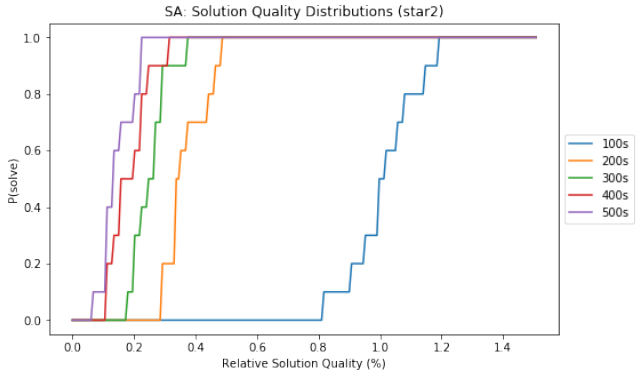

Figure 8: Simulated Annealing Solution Quality Distribution on Star2 Graph

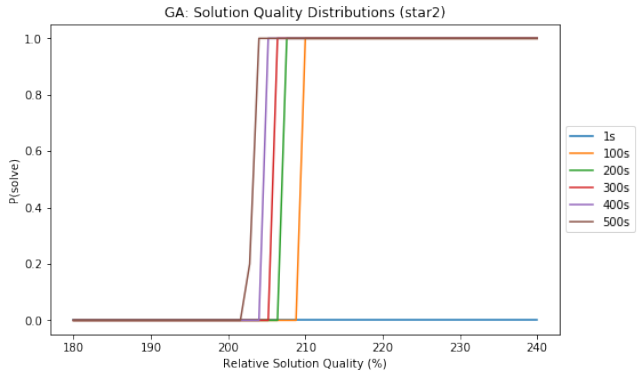

Figure 9: Genetic Algorithm Solution Distribution on Star2 Graph

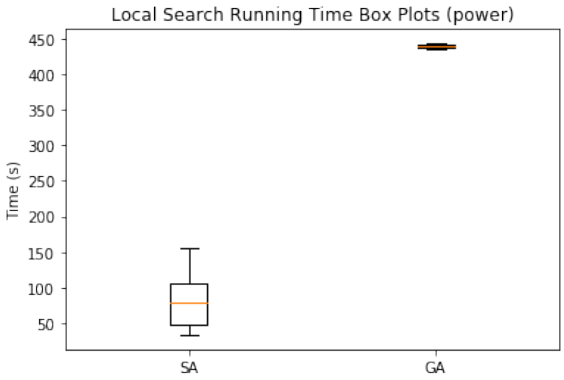

Figure 10: Boxplot of Local Search Running Times on Power Graph

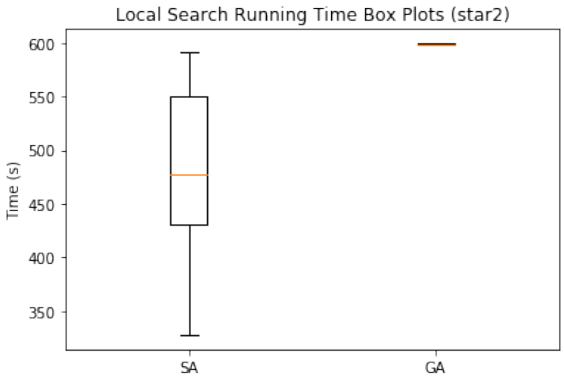

Figure 11: Boxplot of Local Search Running Times on Star2 Graph

REFERENCES

Sangeeta Bansal and Ajay Rana. 2014. Analysis of Various Algorithms to Solve Vertex Cover Problem. (2014).

Walid Ben-Ameur. 2004. Computing the initial temperature of simulated annealing. *Computational optimization and applications* 29, 3 (2004), 369–385.

Shaowei Cai, Kaile Su, and Abdul Sattar. 2012. Two New Local Search Strategies for Minimum Vertex Cover. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence* (Toronto, Ontario, Canada) *(AAAI'12)*. AAAI Press, 441–447.

François Delbot and Christian Laforest. 2010. Analytical and Experimental Comparison of Six Algorithms for the Vertex Cover Problem. 15, Article 1.4 (Nov. 2010), 27 pages. https://doi.org/10.1145/1671970.1865971

Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *science* 220, 4598 (1983), 671–680.

Jon Kleinberg and Eva Tardos. 2005. *Algorithm Design.* Pearson.

Joni Pajarinen. 2007. Calculation of typical running time of a branch-and-bound algorithm for the vertex-cover problem. (2007).

Christos Papadimitriou Sanjoy Dasgupta, Umesh Vazirani. 2006. *Algorithms.* McGraw-Hill Education.