Minimum Vertex Cover

Sasha M. Bakker
Georgia Institute of Technology
Atlanta, Georgia, USA
sbakker3@gatech.edu

Gyujin Park
Georgia Institute of Technology
Atlanta, Georgia, USA
gpark98@gatech.edu

Sebastian Gutierrez Hernandez
Georgia Institute of Technology
Atlanta, Georgia, USA
shern3@gatech.edu

Nathan B. Williams
Georgia Institute of Technology
Atlanta, Georgia, USA
nathanwilliams@gatech.edu

Abstract

This paper is an analysis of various algorithms that solve the Minimum Vertex Cover problem of graphs, an NP-Complete minimization problem. The performance of four different algorithms for computing an exact or estimated minimum vertex cover is explored. The results show that if computational runtime is the main concern, the Approximation Algorithm is the fastest, followed by the Hill Climbing algorithm. On the other hand, if relative error is the main concern, the Hill Climbing algorithm performs the best in most cases, with the Genetic algorithm being a close second.

Keywords: Minimum Vertex Cover, Exact Algorithms, Approximation Algorithms, Local Search Algorithms, Branch and Bound, Maximum Degree Greedy, Hill Climbing Genetic Algorithms

1 Introduction

The Minimum Vertex Cover (MVC) problem is an NP-complete minimization problem common in graph theory. Graph theory is used to model a variety of situations and can provide an abstract and simple representation of often large and complex models. Furthermore, graphs are mathematical constructs that can be explored, manipulated, and operated on. This paper demonstrates four algorithms that are used to solve the MVC. Each of these algorithms have their advantages and disadvantages. The first algorithm used to solve the MVC was the Branch and Bound algorithm, which calculates the optimal solution. However, it comes at the cost of sacrificing running time, which can be exponential. The second algorithm used was an approximation algorithm – Maximum Degree Greedy. Approximation algorithms tend to sacrifice the quality of a solution but are able to quickly find a solution that is bounded and most likely not very bad. The third and fourth algorithms used are both local search algorithms. These types of algorithms are very good at quickly finding a solution but cannot give any quality guarantees. The third algorithm used was the Genetic Algorithm and the fourth algorithm was Hill Climbing.

2 Problem Definition

Consider an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, a vertex cover is a subset $C \subseteq V$ such that $\forall (u, v) \in E : u \in C \vee v \in C$. The Minimum Vertex Cover problem is therefore simply the problem of finding the minimizing $|C|$.

3 Related Work

There has been a variety of research done on solving the Minimum Vertex Cover problem, from exact solution techniques such as the Branch and Bound algorithm, to many different approximation techniques that sacrifice the quality of the solution, but are able to quickly solve the problem, to local search methods such as the Genetic Algorithm, Hill Climbing, Simulated Annealing, and many others. Many of these algorithms, and many more, have been used to solve the minimum vertex cover problem.

3.1 Exact Algorithms

The Branch and Bound (BnB) algorithm is a modified backtracking algorithm. Instead of performing a brute force search, the backtracking algorithm constructs alternatives component-by-component, and eliminates some unnecessary cases from consideration, which makes it faster than the brute force algorithm. It has better time for man instances, but it is still exponential in the worst case. The BnB algorithm refines this backtracking idea and is used for optimization problems. Essentially, a systematic search is performed where some options are removed when possible, and it often takes much less time than the exhaustive approach [5]. In the study by Luzhi Wang, Shuli Hu, Mingyang Li, and Junping Zhou, a new exact algorithm, EMVC, for solving the MVC problem, was introduced based on the BnB search scheme. Two approaches were used to compute the lower bound, one based on the degrees of the vertices, and the other based on MaxSAT reasoning, with the results showing that EMVC is a competitive exact solver. The largest optimum MVC computed using EMVC was 3319, which took 114.15s and the longest running time for any of their tested graphs was 177.73s [3]. It is not stated how many vertices and edges were in these graphs, and we cannot deduce these numbers

since there is no distinct relationship between the optimal MVC and the number of vertices and edges. For example, our dataset "as-22july06.graph" has $|V| = 22963, |E| = 48436$ with 3303 as the MVC whereas the dataset "hep-th.graph" has $|V| = 8361, |E| = 15751$ with 3926 as the MVC. Thus, the results of their study cannot be directly compared to our results.

3.2 Approximation Algorithms

To be considered an approximation algorithm for a minimization problem such as the Minimum Vertex Cover problem, the algorithm must have a time complexity of, at most, polynomial time and must be able to compute a solution whose quality is within a range between the optimal solution and the optimal solution times the approximation ratio r. Thus, the running time is considered "reasonable" and the quality of the solution is guaranteed to be within a range. Some approximation algorithms include the Maximum Degree Greedy algorithm, which is covered in this paper, the Greedy Independent Cover algorithm, the Depth First Search algorithm, the Edge Deletion algorithm, and the ListLeft and ListRight algorithms, among others. In a study done by Francois Delbot Ibisc and Christian Laforest Limos,the aforementioned algorithms are compared over a variety of graphs to solve the minimum vertex cover problem. In particular, to understand the results and the overall performance of an algorithm, they compared the extreme values of the percentage of error. This can be seen in the following figure depicted Table IV of their study[1].

Table IV. Summary of the Variances Obtained by Each Algorithm for Each Sample

	Erdös-Renyi		BHOSLIB		Trees		Reg. graphs		Av. worst case graphs	
	V_{min}	V_{max}	V_{min}	V_{max}	V_{min}	V_{max}	V_{min}	V_{max}	V_{min}	V_{max}
LL	0.8	5.39	2.28	4.15	7.01	14.95	0.0	71.97	1.36	1,319.45
LR	0.0	7.69	1.75	3.29	2.21	11.37	0.0	318.06	0.0	3,3242.0
SLL	0.0	2.21	0.0	0.5	0.12	3.05	0.0	72.55	0.0	31.16
SLR	0.0	6.16	0.0	3.38	0.0	1.55	0.0	318.77	0.0	17.61
ED	0.29	6.31	2.80	5.13	4.97	13.33	0.0	74.87	0.0	51.23
DFS	0.0	3.15	0.0	0.01	0.19	0.25	0.0	5.24	0.0	0.19
MDG	0.0	2.08	0.7	2.32	0.0	1.81	0.0	9246.3	0.0	12.76
GIC	0.0	3.59	0.07	1.97	0.0	0.0	0.0	11.91	0.0	0.0

The first number represents the smallest variance obtained for the sample and the second represents the largest.

This table depicts how the different algorithms compared to one another in terms of percentage error, which demonstrates how the actual structure of the graphs often determines the performance for some of the approximation methods. Unfortunately, as different graphs were used, the results of these algorithms cannot be directly compared to the results from our study.

3.3 Local Search Algorithms

This section explores another type of local search algorithm, the Efficient Simulated Annealing (ESA) algorithm. In a study done by Xinshun Xu and Jun Max, this algorithm has shown promise in solving the Minimum Vertex Cover problem. The ESA algorithm works in a similar way the original Simulated Annealing algorithm. Simulated Annealing is an algorithm founded on the idea of annealing, the process of heating and cooling metals in order to improve their strength. It begins with an initial solution. A neighbor is then selected at random and the change in cost is evaluated. If there is a reduction in cost, the current solution is then replaced by the neighborhood. However, there is a chance that this neighbor leads to a worse solution, thus, there is a probability, continually decreasing with time, that the algorithm selects a different neighbor. This probability is referred to as the acceptance function and is as follows:

$$p = e^{-\Delta F/T}$$

where ΔF is the potential increase in cost and T is the control parameter analogous to the temperature in the annealing process of heating and cooling metals. The Efficient Simulated Annealing algorithm differs from the Simulated Annealing algorithm in the acceptance function that determines the probability of a neighbor leading to a worse solution. The Efficient Simulated Annealing algorithm takes into consideration the idea that a vertex with a larger degree has a higher probability of being in a vertex cover. In comparison to the original Simulated Annealing Algorithm, the Efficient Simulated Annealing Algorithm ties or outperforms the Simulated Annealing algorithm across nine graphs[4]. However, as different graphs were used, the results of the Efficient Simulated Annealing algorithm, cannot be directly compared to the results of this paper.

4 Algorithms

This section details the implementation of each algorithm.

4.1 Branch and Bound

The BnB algorithm, given by the pseudocode in Algorithm 1, makes choices using the structure of a state-space tree. The root is an empty cover $C' = \emptyset$ such that we have the complete graph $G' = (V', E') = G = (V, E)$ and no partial solution. In each sub-graph of G', the most promising node v is the highest degree node, and BnB must choose whether its state will be 0, not in the vertex cover, or 1, in the vertex cover, such that these two choices are added to the frontier for exploration. This frontier is a Last-in-First-Out (LIFO) queue, which leads to a depth-first search in the tree. When the algorithm begins, it selects the last entry in the frontier to construct the leaf, which will contain the updated partial cover C' and updated sub-graph G'. If the state is zero, C' is updated by adding all the neighbors $N(v)$, because the neighbors of v must be in the solution for it to cover all the edges of v. Furthermore, G' is updated by removing the closed neighborhood $N^*(v)$, because all the edges of v are removed such that v is also removed by default. On the other hand, if the state is one, C' is updated by adding v to the set. Then, G' is updated by removing all instances of v. In summary, when C' is modified, the updated vertices V' and

edges E' of G' are given by the equation:

$$G' \equiv \begin{cases} V' = V - C' - \{v \in V | \forall (v,u) \in E, u \in C'\} \\ E' = \{(u,v) \in E | v,u \in V'\} \end{cases}$$

Following the creation of the leaf (C', G') from the choice of v having state zero or one, the next step is to evaluate whether C' is a candidate solution. If $|E'| = 0$, then G' has no edges such that C' covers all edges in G, and is therefore, by definition, a vertex cover of G. For this candidate solution, it must be determined whether it is a smaller vertex cover than the current upper bound (initialized to $|V|$). If it is a better solution, then the current upper bound becomes $|C'|$ and our best solution so far is now the current instance of C'. Once a candidate solution is found, this is a dead end and G' must become sub-graph in the parent leaf for the last configuration in the frontier. In the case that $|E'| \neq 0$, C' is only a partial solution, and it must be determined whether this partial solution is worth exploring, meaning, we want to know whether the partial solution may lead to a complete solution. If $|C'|$ plus some lower bound is less than the current upper bound, then the branch is indeed worth exploring. Thus, the highest degree node of G' is added to the frontier for both states zero and one. If the condition is not satisfied, then the partial solution is not worth exploring and we prune the branch. It is therefore a dead end and G' must become the sub-graph in the parent leaf for the last configuration in the frontier. Now, there are two details left to discuss: the lower bound on G', and the method of retrieving the graph of a parent leaf.

The lower bound is used as the degree-based lower bound for MVC, as described in the article "An exact algorithm for minimum vertex cover problem" [3]. Given an input graph $\bar{G} = (\bar{V}, \bar{E}) = G = (V, E)$, the vertex with the largest degree is selected and named as v'_1. It is removed from $\bar{G}$ such that the degree of the other vertices are updated. Then, the largest degree vertex of $\bar{G}$ is selected, named v'_2, and removed from the graph. This process continues iteratively until the first vertex v'_i that satisfies $\sum_{k=1}^{i} \text{degree}(v'_k) \geq |E|$ is found. Them the lower bound of G is defined by:

$$\text{degLB}(G) = \left\lfloor i + \frac{|\bar{E}|}{\text{degree}(v'_{i+1})} \right\rfloor$$

A potential strength of this lower bound is that it may be tighter than lower bounds produced from other methods. However, its weakness is run time since the method is iterative, which is an issue for large-scale graphs. Essentially, in BnB, degLB(G') is an upper-bound on the vertex cover of the partial graph G'. Thus, the number of iterations is at most the number of nodes in an upper-bound estimate for the MVC of G' and at least the number of iterations is the exact MVC of G'. Thus, the time complexity is a function of a minimum vertex cover of G'.

Algorithm 1 BnB

```
Inputs: G=(V,E), time limit
Frontier, C_OPT, C' ← ∅, ∅, ∅
Dead end = False
UB ← |V|
G' ← G
Set child node v to highest degree node of G
Set v's parent node as 0 with state 0 (root)
Append ((v,0), parent) and ((v,1), parent) to Frontier
while Frontier not empty do
    if current time > time limit then break
    if Dead end == True then
        Set G' to parent graph for last config in Frontier
        Dead end = False
    end if
    Choose child = (v, state) from last config in Frontier
    if state == 0 then
        G' ← G' \ N*(v)
        C' ← C' ∪ N(v)
    end if
    if state == 1 then
        G' ← G' \ v
        C' ← C' ∪ v
    end if
    if |E'| == 0 then
        # Candidate solution found
        Dead end = True
        if |C'| < UB then
            C_OPT ← C'
            UB ← |C'|
        end if
    else
        # Check if partial solution is worth exploring
        if |C'| + LB < UB then
            w ← highest degree node of G'
            Append w to Frontier for both states 0, 1
        else Dead end = True
        end if
    end if
return UB, C_OPT
end while
```

Now that the lower bound is demystified, the next matter is the method of retrieving the graph of a parent leaf. Say a node v with corresponding state s_v is chosen from the frontier. Its parent node is p and its parent node's state is s_p. There are two cases to consider. The first case is that $p = 0$ and $s_p = -1$, this means the parent node is the root, such that G' is re-initialized to G and C' to $\emptyset$. The next case is that p is a node in the current branch, meaning, the current branch contains a leaf constructed from the choice of p having the state s_p. This must be true because, for v to have been added to the frontier, there would have been a leaf previously constructed

due to the state of p. Thus, the branch that led to our current C' is back-tracked by iteratively re-adding all the nodes and edges previously removed due to the choices in the branch. Meaning, we are moving up leaf-by-leaf in the branch. Now that G' corresponds to the sub-graph of the parent leaf, its child leaf can be constructed from the last configuration in the frontier.

The last matter of discussion is the time and space complexity of BnB. The number of iterations in the algorithm is dependent on the number of configurations in the frontier, which covers all possible cases of each node having state zero or one. This number is $2^{|V|}$, such that BnB is $O\left(2^{|V|}\right)$. The space complexity of BnB is $O(|V|+|E|)$ because the algorithm only needs to keep track of the current set of vertices V' and edges E' need, where $|V'| \leq |V|$ and $|E'| \leq |E|$.

4.2 Maximum Degree Greedy

The Maximum Degree Greedy (MDG) algorithm, given by the pseudocode in Algorithm 2, is an approximation technique used to approximate a solution to NP-complete problems such as the minimum vertex cover. It adapts the Greedy algorithm used in the classical set cover problem. In the context of the minimum vertex cover problem, the algorithm is given a graph $G = (V, E)$ where V is the set of vertices in the graph and E is the set of the corresponding edges. The MDG algorithm approximates a solution by first initializing a solution list and repeatedly appending the vertex with the maximum degree of G to the solution list. Once a vertex is appended to the solution list, it, along with all of its corresponding edges, are removed from the graph. This method is repeated until there are no edges left in G. The algorithm's pseudocode is depicted in Algorithm 2.

Algorithm 2 Maximum Degree Greedy Algorithm [1]

Data a graph $G = (V, E)$
Result a vertex cover of G
C $\leftarrow \emptyset$
while $E \neq \emptyset$ **do**
    selected a vertex u of maximum degree;
    $V \leftarrow V - \{u\}$;
    $C \leftarrow C \cup \{u\}$;
end while
return C

Furthermore, the worst-case approximation ratio of MDG is $H(\Delta)$, with $H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ the harmonic series ($H(n) \approx ln(n) + 0.57$ when n is large) and Δ the maximum degree of the graph[1]. The goal of the MDG algorithm is to quickly find a solution that is probably not very bad. As such, its worst-case running time is polynomial, however, it does not guarantee providing the optimal minimum vertex cover of a given graph.

Additionally, the Maximum Degree Greedy algorithm is an iterative algorithm that, with each run of the loop, adds the vertex with the highest degree to the minimum vertex cover. Because of the way it is written in the code, the time complexity of each iteration is $O(n')$ where n' is equal to the number of vertices left in the graph. A generous upper-bound for the time complexity of the overall algorithm is $O(n)$ where $n = |V|$ because we know it will always have less iterations than the number of vertices in the original graph. Furthermore, given a graph, $G = (V, E)$ where V is a set of vertices and E is a set of edges, the space complexity of this algorithm is $O(|V| + |E|)$.

4.3 Hill Climbing

For hill-climbing local search, we used Maximum Degree Greedy solution as our initial solution. After that, we remove one vertex from the solution randomly followed by exiting and entering one vertex. For exiting vertex, we used a cost function as the number of edges not covered by the current vertex set. And we determined exiting vertex for which we have minimum cost after removing. For entering vertex, we choose a vertex not covered by the current vertex set randomly. The algorithm's pseudocode is depicted in Algorithm 3.

Algorithm 3 Local Search, Hill Climbing

Data a graph $G = (V, E)$
$C \leftarrow \emptyset$
while $E \neq \emptyset$ **do**
    select a vertex u of maximum degree
    remove u from V
    append u to C
end while
$C' = C$
while elapsed time < cutoff **do**
    if C is a vertex cover **then**
        $C' = C$
        Remove one vertex from C randomly
    end if
    u <- a vertex that produces the minimum increase in number of edges not covered by C
    remove u from C
    v <- a random vertex not covered by C
    append v to C
end while
return C'

4.4 Genetic Algorithms

The origins of genetic algorithm start in the 1940s, Alan Turning proposed a genetic or evolutionary search. The first implementation was in 1962 by Bremermann. After this, until the 1990s, there were mainly three perspectives on

what today is called genetic algorithms. These were evolutionary programming, genetic algorithms, and evolution strategies. Later, a fourth idea, called genetic programming gained strength as a research direction. What today is called a genetic algorithm encompasses the ideas from these four research directions. The inspiration for these algorithms are the following key points of Darwin's theory of evolution

- Given a constant-sized population, with finite resources, only the fittest will survive. The fitness of an individual is determined by its unique phenotypic traits.
- A correct combination of two individuals' phenotypes, will produce fitter offspring.
- Small random variations or mutations in phenotype traits might happen during the creation of an offspring.

Genetic algorithms can be used to solve free optimization problems like the traveling salesman problem, constraint satisfaction problems like the eight-queens problem, or constrained optimization problems such as the tsp with an order in which two or more cities need to be visited [2]. The part of the pseudocode for our genetic algorithm is in Algorithm 4, and the other algorithmic components are in Section 8. A genetic algorithm starts by constructing a population for the problem. For a given k we test if the genetic algorithm finds a vertex cover. In case it finds a vertex cover size k we test for $k-1$ and the procedure continues until the genetic algorithm cannot find a vertex cover for a given size. The stopping criterion is the number of iterations. If after 150 iterations, the algorithm has not improved, it stops. To initialize the solution to the decision problem using a genetic algorithm, we have to initialize a population, we consider a population of 150 elements. A population is a list of individuals. Each individual is an array of 0's and 1's. The arrays have length equal to $|V|$. Each individual is an indicator function for candidate subset $S \subset V$. If the node $i \in S$ then the i–th entry of the individual is 1, if $i \notin S$ then the corresponding entry is 0. To assess the quality of a solution, define the fitness function $h : [0,1]^{|V|} \to \mathbb{R}$ for a binary array c by

$$h(c) = \sum_{i=1}^{|V|} \left[(1 - c[i]) \sum_{j:(i,j)\in E} (1 - c[j]) \right]$$

This function returns the number of edges that are not covered by the subset S associated with the binary array c.

The fitness of each individual in the population is used to do the selection process. Sample 1% of the population and select the individual with the lowest score. We repeat this process 150 times, the size of the population. After the fittest individuals of the population have been selected, the crossover procedure is done. For this, we select two contiguous individuals and randomly an integer p in the interval $[0, |V|-1]$, then the children of the two parents p_1 and p_2 are defined by

$$c_1 = p_1[:p] + p_2[p:] \quad c_2 = p_2[:p] + p_1[p:].$$

Since we are solving the decision version of the problem, it is important that the children c_1 and c_2 have at most k entries with value 1. For ensuring this, if a child has more than k, let us say k', entries with value 1, form a set of nodes corresponding to the child and add 30 copies of the $k' - k$ nodes with the least amount of edges, let us call this set S'. We sample $k' - k$ elements, without replacement, from S' and we make the corresponding entries of the child 0 to remove them from the candidate vertex cover. The value 30 was selected experimentally. The final step for a genetic algorithm is the mutation of the individuals. The mutation process happens for 25% of the population, they are randomly sampled from the population. For a selected individual, let S be the corresponding cover and E' the set of edges not covered by S. Define m' to be a random sample in the interval $[0, min(|S|, |E'|)]$ then define

$$m = max(1, m').$$

Algorithm 4 Local Search, Genetic algorithm (decision algorithm)

```
Inputs: G=(V,E), k, num-iterations,
Create a population, whose elements have k 1s.
Include the previous vertex cover in the current population.
Evaluate the fitness of every individual.
Find the element with the lowest fitness.
if best fit == 0 then
    Return vertex cover
end if
for gen = 1, num-iterations do
    Evaluate the fitness of every individual.
    Find the element with the lowest fitness.
    if best fit == 0 then
        Return vertex cover
    end if
    if the best fit has not changed in 150 iterations then
        Return current best vertex cover
    end if
    Do selection process. Algorithm 5
    for i in the population range do
        Generate two children from i and i+1. Algorithm 6
        Mutate each child. Algorithm 8
    end for
end for
Return vertex cover
```

We then randomly select m elements for S and m edges not included E'. For each of the randomly sampled edges, randomly select an end from the edge, obtaining m nodes. The nodes sampled from S are removed from the individual and the m nodes samples from E' are included in the individual. Note that this mutation ensures that each individual

maintains a fixed amount of 1s. The procedure just described is one iteration of the algorithm. For each iteration, the current best solution is the element of the population with the lowest fitness. Note that if the fitness is 0 then we have found a vertex cover. As mentioned earlier if the best result is not improved after 150 iterations, the algorithm terminates.

It is worth mentioning that the minimum vertex cover can be solved directly, without doing the decision problem. In this case, the fitness function is :

$$h(c) = \sum_{i=1}^{|V|} \left[c[i] + |V|(1 - c[i]) \sum_{j:(i,j)\in E} (1 - c[j]) \right]$$

We experimented with this algorithm and there are some differences that should be pointed out. Because there is no restriction in the length of the vertex covers, the cross-over and mutation procedures can be done in a straightforward way. Which makes the algorithm faster. In our experiments, we observed that going through the decision problem in most cases reaches a smaller vertex cover.

Genetic algorithms have a significant amount of parameters, on which, the quality of the solution is highly dependent. For this project, finding an appropriate set of parameters for each problem was time-consuming. We choose a set of parameters that yielded reasonable solutions for all the problems. It it is likely that the solutions can be improved by choosing an appropriate set of parameters for each problem. In our experimentation, we observed that having a large mutation parameter had a positive impact on the graphs with the most amount of nodes, but a negative impact on smaller graphs.

We now discuss the computational costs of the algorithms that depend on the graph G. We do not include the Maximum Degree Vertex algorithm. The cost function, for a complete graph, has the cost $O(|G||E|)$. In the cross-over, if a child has more than k 1s, we find the vertices with a minimum degree, this, in the worst case, requires $O(|G|log(|G|))$. In the mutation step we look for the edges not covered by a proposed cover, since we have to go through every edge this has cost $O(|E|)$. The rest of the algorithms depend on the parameters chosen for the genetic algorithm and we omit their cost. We thus claim that the cost of the algorithm is $O(|G|(log(|G|) + |E|))$.

5 Empirical evaluation

This section describes the platform, experimental procedure, evaluation criteria, and obtained results.

5.1 BnB

BnB was executed for all graph files using Google Colab Pro with the Intel(R) Xeon(R) CPU at 2.20 GHz with 32GB RAM for a maximum run-time of 10 hours (36,000s). For the files 'star.graph', 'star2.graph', and 'as-22july06.graph', the NVIDIA Tesla T4 GPU accelerator was added. These three graph instances were unable to produce a single vertex cover within the ten-hour limit, such that their estimate for $|MVC|$ is the number of vertices in the input graph $|V|$. The instances of the other graph files produced vertex covers with a relative error below 4-percent (Table 1), with the input files 'jazz.graph', 'karate.graph', and 'netscience.graph' producing the exact MVC such that their relative error is zero.

5.2 Maximum Degree Greedy

The Maximum Degree Greedy algorithm was executed for all graph files using the Spyder IDE on a Macbook Pro with the Apple M1 Max chip. The computer consists of a 10-core CPU, a 32-core GPU, a 16-core Neural Engine, and 64 GB unified memory. The maximum relative error across all graph files was found in the "star" graph, a relative error of 0.068. Furthermore, the Maximum Degree Greedy algorithm produced the optimal minimum vertex cover in two of the graphs: "karate" and "netscience". The results for all of the graphs are depicted in Table 1. Furthermore, the Maximum Degree Greedy algorithm's worst-case approximation ratio is $H(\Delta)$, with $H(n) = 1 + \frac{1}{2} + \cdots + \frac{1}{n}$ the harmonic series ($H(n) \approx ln(n) + 0.57$ when n is large) and Δ the maximum degree of the graph[1]. Thus, analysis was performed to determine whether the algorithm's results stayed within the worst-case approximation ratio bound. Let us consider the ratio r to be ($H(n) \approx ln(n) + 0.57$ where n is the maximum degree of the initial graph. To do this, we consider A to the be the solution computed by the Maximum Degree Greedy algorithm. Thus, $A \leq r*OPT$. For each of the graphs, the solution computed by the Maximum Degree Greedy algorithm stayed within the worst-case approximation ratio bound. In fact, the Maximum Degree Greedy algorithm performed extremely well, staying within 2.5 times of the worst-case approximation ratio bound. Furthermore, this algorithm demonstrated very quick results in terms of time, especially in comparison to the other algorithms in this paper.

5.3 Hill Climbing

The hill-climbing algorithm was executed for all graphs using visual studio code on a HP 15z-ef2000. The computer has 8 GiB of RAM, it has AMD Ryzen™ 5 5500U, and AMD Radeon™ Graphics. For each graph, the algorithm ran until 100 seconds. The case where the hill climbing algorithms outperformed the rest of the algorithms are the hep-th, delaunay-n10 and power. From Table 1 we can see that the genetic algorithm is always within a 7% relative error even when it was ran until 100 s. In Figures 3, Fig 4, Fig 5, Fig 6, Fig 7, and Fig 8 we see the QRTD, SQD and box plots for the power and star2 graphs.

5.4 Genetic Algorithms

The genetic algorithm was executed for all graphs using visual studio code on a Lenovo Legion Y72. The computer has 16 GiB of RAM, an 8 Inter Core i7-7700HQ, and a GeForce GTX 1060. The genetic algorithm is slower than the rest of

the algorithms. The reason for this is that for a fixed k the operations to keep an individual in the sets of k vertices are expensive. We have to find the vertex with the minimum degree or several of them. Networkx does not keep the edges sorted, this implies that for finding the vertex with a minimum degree it has to sort them first. The computational cost of this operation depends on the algorithm that python uses to sort a list. The case where the genetic algorithms outperformed the rest of the algorithms are the star, and email graphs. The stopping criteria for the genetic algorithms are the number of iterations without progress. For the star graph, the time it took the algorithm to reach the same solutions as LS1 was around 53 s. In Figures 9, Fig 10, Fig 11, Fig 12, Fig 13, and Fig 14 we see the required plots for the power and star2 graphs. The reason why the variability of the q is so small is because of the initialization done with the Maximum Degree Algorithm. From Table 1 we can see that the genetic algorithm is always within a 0.07 relative error.

6 Discussion

We have two main criteria to compare the different algorithms, the time and the relative error. In terms of time, the fastest algorithm was consistently the Approximation algorithm, followed by the Hill Climbing algorithm. In some cases, when the graph has a small state space, the BnB algorithm is faster than the genetic algorithm but the genetic algorithm is able to provide answers for larger problems, that BnB is not able to in a reasonable amount of time (Fig.7).

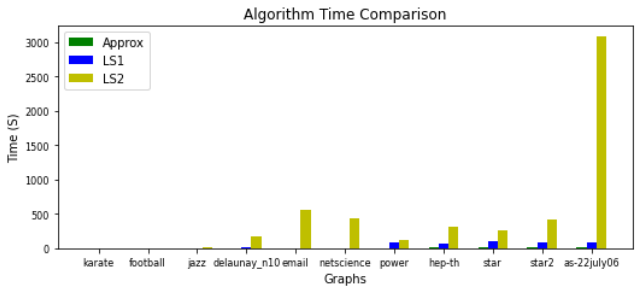

Figure 1. Time (s) comparison (Ref. Table 1).

The error comparison shows that the Hill Climbing algorithm is the best, in some cases, for example, star, star2, and email the genetic algorithm slightly outperforms the Hill Climbing algorithm. It is important to mention that the Hill Climbing algorithm was always faster for the cited cases. See Fig. 1 Interestingly, the BnB algorithm did not always provide an exact answer as we would expect, for example, "email" and "delaunay_10" finish running much well before the time limit of 10 hours, but they do not produce an exact solution for the minimum vertex cover. This means that there is something incorrect in how the algorithm is searching the state space. However, the errors on these two graphs are less than 1.5% for BnB, so the result is still a good estimate for the MVC, even though the algorithm is not precisely "exact".

jazz	BnB	Approx	LS1	LS2
Time (s)	0.67	0.0062	0.2	2.78
|MVC|	158	159	158.9	158.90
Error	0.00	0.0063	0.0057	0.0015
karate	BnB	Approx	LS1	LS2
Time (s)	0.0027	0.00015	0.00082	0.00064
|MVC|	14	14	14	14.00
Error	0.00	0.00	0.00	0.00
football	BnB	Approx	LS1	LS2
Time (s)	0.20	0.0020	0.026	0.61341
|MVC|	94	96	95.7	95.20
Error	0.00	0.021	0.018	0.013
as-22july06	BnB	Approx	LS1	LS2
Time (s)	0.00	19.49	75.61	1406.34
|MVC|	|V| = 22963	3307	3310.6	3310.00
Error	5.95	0.0012	0.0023	0.0021
hep-th	BnB	Approx	LS1	LS2
Time (s)	34105.80	6.61	66.04	227.88
|MVC|	3942	3944	3939.2	3941.80
Error	0.0041	0.0046	0.0034	0.0038
star	BnB	Approx	LS1	LS2
Time (s)	0.00	13.99	96.31	191.36
|MVC|	|V| = 11023	7374	7366	7358.40
Error	0.60	0.068	0.067	0.064
star2	BnB	Approx	LS1	LS2
Time (s)	0.00	15.53	92.70	285.55
|MVC|	|V| = 14109	4697	4673.9	4672.70
Error	2.11	0.034	0.029	0.028
netscience	BnB	Approx	LS1	LS2
Time (s)	130.77	0.26	0.55	1.039
|MVC|	899	899	899	899.00
Error	0.00	0.00	0.00	0.00
email	BnB	Approx	LS1	LS2
Time (s)	91.47	0.14	1.05	451.19
|MVC|	602	605	602.2	599.60
Error	0.013	0.019	0.014	0.011
delaunay_n10	BnB	Approx	LS1	LS2
Time (s)	80.54	0.14	6.42	122.12
|MVC|	730	737	723.8	724.50
Error	0.038	0.048	0.030	0.038
power	BnB	Approx	LS1	LS2
Time (s)	34517.60	2.28	92.45	97.12
|MVC|	2259	2277	2234.3	2261.10
Error	0.025	0.034	0.014	0.024

Table 1. Comprehensive results for all *.graph files.

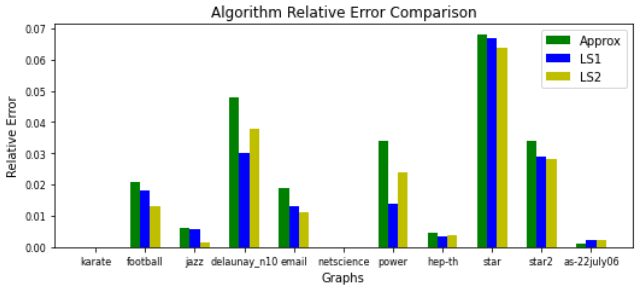

Figure 2. Error comparison (Ref. Table 1).

Since both local algorithms were initialized with an approximation method, we look at some examples, to see the time that it took to make the first improvement. The comparison in Table 2 shows that running either of the local search methods for less than two minutes will improve the results given by the maximum degree algorithm.

Graph	LS1	LS2
"hep-th"	22	22.69
"star"	90	28
"email	0.2	4.65

Table 2. First improvement

From the QRTD plots Fig. 3 (LS1) and Fig. 9 (LS2) for the power graph example we can conclude that the Hill climbing algorithm is faster. The box plots Fig 7, Fig 8 for Hill Climbing and Fig 14, and Fig 15 for the genetic algorithm show that the variance are all based on the time it takes for the algorithms to get a solution of quality $q = 0.03$. From these we can see that the Hill Climbing algorithm is faster, but there is more variability, compared to the genetic algorithm. An important path to explore is how the structure of the graph may determine algorithm choice; some graph structures impact the performance of approximation algorithms and local search algorithms in different ways.

7 Conclusion

This paper presented four algorithms used in solving the Minimum Vertex Cover problem. Each of the algorithms has their unique advantages and disadvantages. The clear trade-off in these algorithms is between accuracy and time. The use of any of the algorithms is dependent upon the stakeholders in the problem. If accuracy is required and time is of no object, the Branch and Bound algorithm should be used. If time is more important than accuracy, an approximation algorithm such as the Maximum Degree Greedy algorithm can be used to initialize a local search algorithm. If time is a concern, it is likely that within a few minutes the Hill Climbing algorithm will improve the current solution. Thus, depending on the requirement considerations of the given problem, each of these four algorithms may be used to solve the Minimum Vertex Cover problem.

References

[1] Francois Delbot and Christian Laforest. 2010. Analytical and experimental comparison of six algorithms for the vertex cover problem. *ACM Journal of Experimental Algorithms - JEA* 15 (03 2010). https://doi.org/10.1145/1865970.1865971

[2] A.E. Eiben and J.E. Smith. 2015. *Introduction to Evolutionary Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-44874-8

[3] Luzhi Wang, Shuli Hu, Mingyang Li, and Junping Zhou. 2019. An exact algorithm for minimum vertex cover problem. *Mathematics* 7, 7 (2019), 603. https://doi.org/10.3390/math7070603

[4] Xinshun Xu and Jun Ma. 2006. An efficient simulated annealing algorithm for the minimum vertex cover problem. *Neurocomputing* 69, 7-9 (2006), 913–916.

[5] Xiuwei Zhang. 2022. Coping with NP-completeness - 2.

8 Pseudocode

Algorithm 5 Local Search, Genetic algorithm (selection)

Input: population, fitness of the population
for i =1, size of the population **do**
    Randomly select 1% of the population.
    Select and save the fittest element
end for
Return: List of elements with the highest fit.

Algorithm 6 Local Search, Genetic algorithm (cross-over)

Input: population, cross-over parameter r, parent 1 p_1, parent $2p_2$
$c_1 := p_1$ and $c_2 := p_2$
Let p be the sample of a uniform r.v. in [0,1].
if $p < r$ **then**
    $c_1 = p_1[:p] + p_2[p:]$
    $c_2 = p_2[:p] + p_1[p:]$
    $c_1 =$ return-to-space(c_1, G, k)
    $c_2 =$ return-to-space(c_2, G, k)
end if
Return: c_1, c_2.

Algorithm 7 Local Search, Genetic algorithm (return-to-state-space)

Input: c, G, k.
if c has more than k 1s **then**
    $N = \sum c - k$
    Find the N vertices $v_1, \ldots, v_N$ with minimum degree in the cover associated with c
    From the set V' as the union of V cup 30 copies of each v_i.
    Randomly sample N elements of V'
    Remove the sampled elements from c.
end if
Return: c

Algorithm 8 Local Search, Genetic algorithm (mutation)

Input: c, mutation parameter m
Let p be the sample of a uniform r.v. in [0,1].
if [**then**$p < m$]
    E' := set of edges not covered by c.
    Let s be an integer uniform sample from $\{0, 1, \ldots, \min(|E`|, k)$
    Let $q = max(1, s)$.
    Remove q random nodes from the cover associated to c.
    Add q random nodes from the edges not covered E' to c.
end if
Return : c

Algorithm 9 Local Search, Genetic algorithm (optimization algorithm)

Inputs: G=(V,E), cutoff time
Initialize a solution using the maximum degree greedy algorithm, S.
Define as current best the cost of the current solution.
Define $k = |S| - 1$
Remove the node with the least degree from the current solution.
Run Algorithm 4 for current k.
Define current cost to be the cost for the decision problem of size k.
Define current vertex cover to be the vertex cover of the decision problem of size k.
while current cost < 1 and elapsed time < max time **do**
    Update best vertex cover and best cost with current vertex cover and current cost
    Remove the node with the least degree from the current solution.
    $k- = 1$.
    Run Algorithm 4 for current k.
    Define current cost to be the cost for the decision problem of size k.
    Define current vertex cover to be the vertex cover of the decision problem of size k.
end while
Return best vertex cover and its size k

Appendix

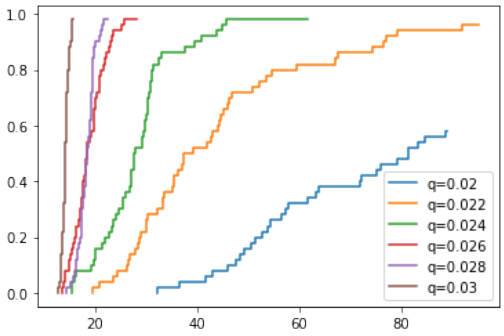

Figure 3. QRTD Power hill climbing

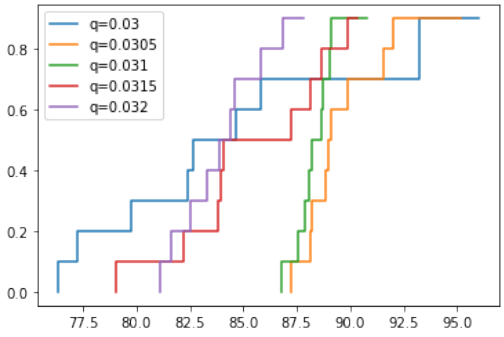

Figure 4. QRTD Star2 hill climbing

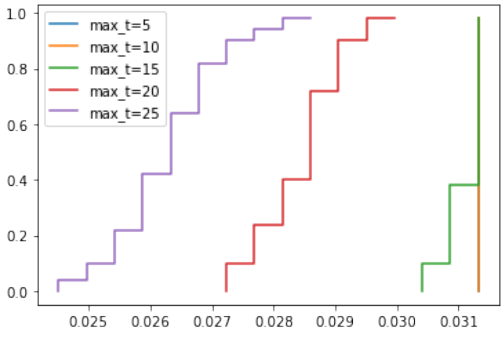

Figure 5. SQD Power hill climbing

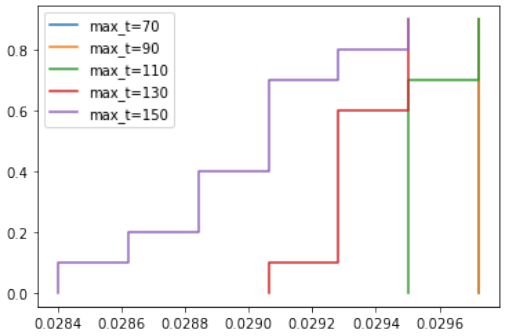

Figure 6. SQD Star2 hill climbing

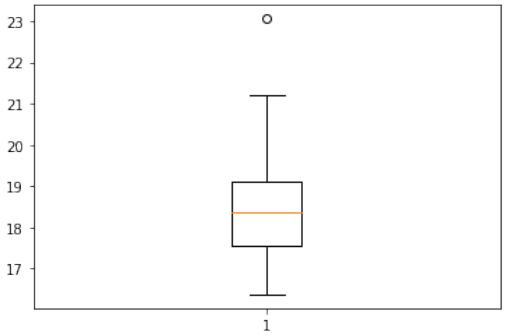

Figure 7. Box plot Power hill climbing

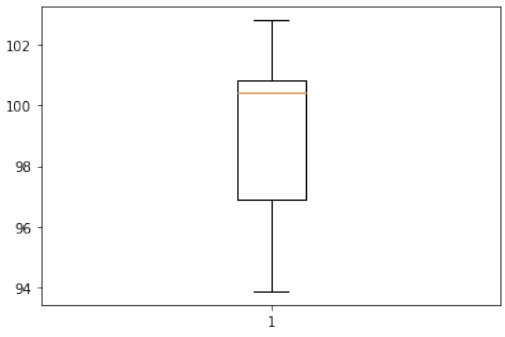

Figure 8. Box plot Star2 hill climbing

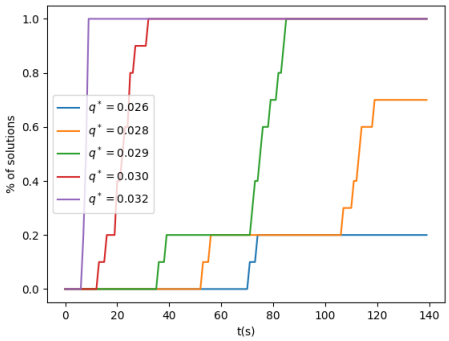

Figure 9. QRTD Power genetic algorithm

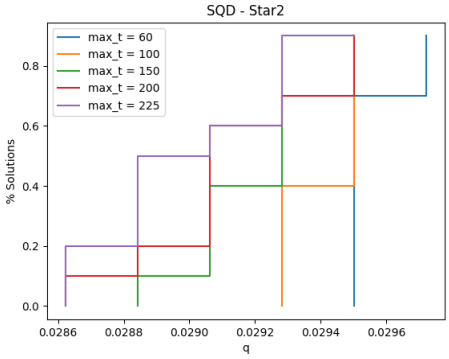

Figure 12. sqd-star2 genetic algorithm

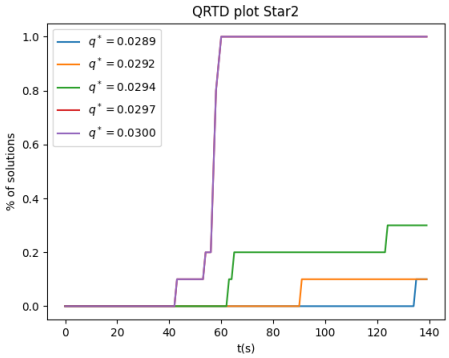

Figure 10. QRTD Star2 genetic algorithm

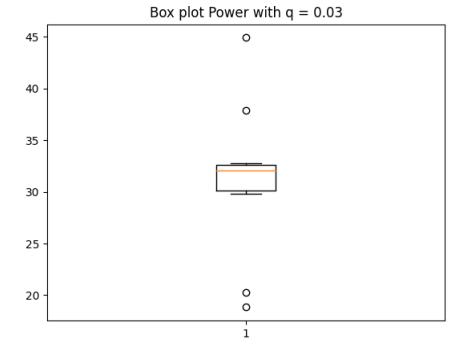

Figure 13. Box plot power genetic algorithm

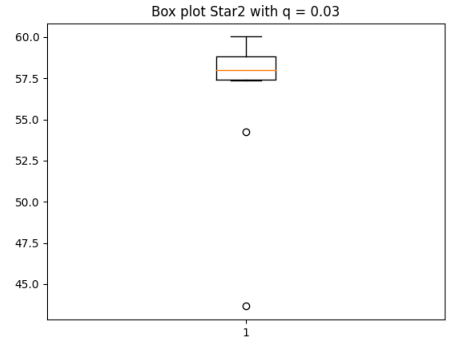

Figure 14. Box plot star2 genetic algorithm

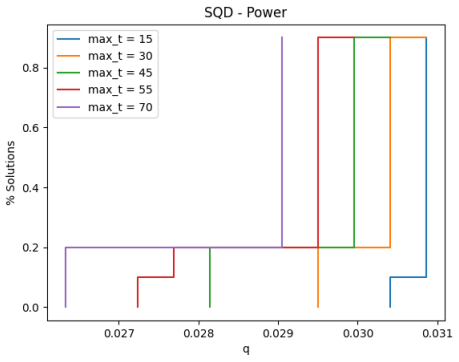

Figure 11. sqd-power genetic algorithm