Minimum Vertex Cover: CSE6140 Final Report

Group 38

Kevin Berry
Georgia Institute of Technology
kpberry11@gatech.edu

Mostafa Reisi Gahrooei
Georgia Institute of Technology
mrg9@gatech.edu

Yifei Wang
Georgia Institute of Technology
wangyf1229@gatech.edu

ABSTRACT

We implement and evaluate three different class of algorithms to solve the minimum vertex cover problem. Two local search algorithms, an approximation algorithm, and the branch and bound algorithm are tested on eleven different networks for their performance. We analyze the theoretical runtime and performance of each algorithm and compare it to their empirical performance based on relative error, runtimes, and SQDs and QRTDs where relevant.

1. INTRODUCTION

The Minimum Vertex cover (MVC) problem is a well known NP-complete problem with numerous applications in computational biology, operations research, and the routing and management of resources. In this paper, we treat the problem using different algorithms, evaluating their theoretical and experimental complexities on real world datasets. The goal of this paper is to solve the vertex cover problem using several different class of algorithms. In particular, we would like to compare and contrast each of the algorithms, branch and bound, local search, and approximation algorithms, in terms of time complexity and quality of the solution. For this purpose, we consider eleven different real networks and find the vertex cover of each network using the following algorithms:

- Branch-and-Bound: We used branch and bound algorithm to attempt to find the exact solution to VC problem. To find the lower bound at each configuration node, we used an approximation algorithm rather than a linear programming (LP) algorithm.
- Local search algorithms: We used two local search algorithms, hill climbing and FastVC to find approximate solutions to the VC problem.
- Approximation algorithm: We examined several greedy approximation algorithms with upper bound guarantees, and one without, to solve the problem.

We compare the solutions obtained by an algorithm to known optimal solutions (shown in table 1) to obtain solution quality. For the local search algorithms QRTD and SQD graphs are generated to illustrate the quality of solutions at different run-times in a probabilistic manner.

The rest of this report is organized as follows: In the next section, a formal definition of the problem is given. After this, we provide a brief discussion of related work in practically solving vertex cover problems. Next, in Section 4 we provide a description of each algorithm used in this project along with pseudocode and time and space complexities. We then provide an empirical evaluation of the algorithms along with a comparative analysis of how different algorithms perform on each of the eleven real-world networks in Section 5. In Section 6, we discuss the pros and cons of each algorithm. Finally, Section 7 summarizes the report.

Table 1: Real-world networks used for our analysis. The optimal VC is given in this table

Problem name	Description	$\|V\|$	$\|E\|$	Opt
jazz.graph	Jazz musicians network. List of edges of the network of Jazz musicians.	198	2742	158
karate.graph	Zachary's karate club: social network of friendships between 34 members of a karate club at a US university in the 1970s.	34	78	14
football.graph	American College football: network of American football games between Division IA colleges during regular season	115	613	94
as-22july06.graph	Graph of the whole Internet: a symmetrized snapshot of the structure of the Internet at the level of autonomous systems, reconstructed from BGP tables	22963	48436	3303
hep-th.graph	Weighted network of coauthorships between scientists	8361	15751	3926
star.graph	Star-like structures of different graphs with different types	11023	62184	6902
star2.graph	Star-like structures of different graphs with different types	14109	98224	4542
netscience.graph	Coauthorship network of scientists working on network theory	1589	2742	899
email.graph	List of edges of the network of e-mail interchanges between members of a university	1133	5451	594
delaunay n10.graph	Delaunay triangulations of random points in the unit square	1024	3056	703
power.graph	Network representing the topology of the Western States Power Grid	4941	6594	2203

2. PROBLEM DEFINITION

The formal definition of the vertex cover problem is as follows: Given an undirected graph $G = (V, E)$ with a set of vertices V and a set of edges E, a vertex cover is a subset $C \subset V$ such that $\forall\,(u, v) \in E$, we have $u \in C$ or $v \in C$. The Minimum Vertex Cover (MVC) problem is therefore simply the problem of minimizing $|C|$. Figure 1 shows an example of a minimum vertex cover on a graph. In the example, $\{b, c, d\}$ form a vertex cover that includes all the edges.

3. RELATED WORK

The minimum vertex cover (MVC) problem is an NP-complete problem with several real-world applications, such as network security, scheduling, VLSI design and industrial machine assignment [1]. The problem is known to be equivalent to Maximum Independent Set (MIS) problem and the

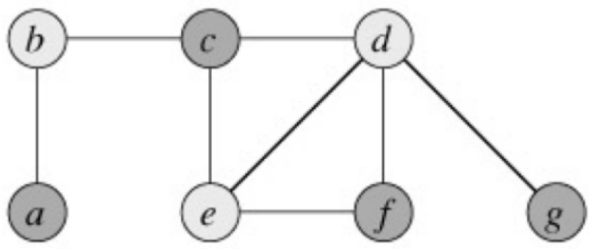

Figure 1: An example of vertex cover

Maximum Clique (MC) problem, which both have important real world applications. An algorithm that approximately solves the MVC problem can also approximately solve MIS and MC problems. In general there are three class of algorithms that can solve the MVC: exact algorithms which mainly use the branch and bound procedure [2], approximation algorithms that guarantee a bound on the solution, and local search algorithms with no guarantee. Some good approximation algorithms, some of which appear in this project, have been proposed by [3]. Several practical heuristics have been proposed for solving the MVC problem. [4] is a good review of such heuristics. Also, [5] has proposed two local search algorithms for solving the MVC. Among the local search algorithms, we found the fast vertex cover algorithm, proposed by [6] promising and used it as one of our local search approaches for finding MVCs of the given networks.

4. ALGORITHMS

In this section, we will discuss each of the algorithms that we used to solve the vertex cover problem. In particular, we provide the details of the branch and bound, local search, and approximation algorithms which we implemented.

4.1 *Branch and Bound*

A branch and bound algorithm finds the exact solution to a problem by searching and pruning possible solutions. We have implemented our BnB algorithm such that each branch either includes a particular vertex or excludes a particular vertex. When branching, we sort candidate vertices by increasing order of their degree, in the assumption that the less connected vertices are more likely to disqualify potential solutions when they are considered than the more connected vertices. In order to find the lower bounds used for pruning, we first attempted to use a linear programming solver to determine a lower bound for each graph; however, in practice this ran prohibitively slowly for large graphs. As a result, we used the edge deletion approximation algorithm described later in section **4.3** to find the lower bounds. We define dead ends to be candidates for which values for all vertices have been assigned or candidates for which the assignment of the vertices leaves isolated edges which have both endpoints assigned to not be in the vertex cover. The following is the pseudocode for the branch and bound algorithm:

```
Branch and Bound Algorithm
Input: graph $G = (V, E)$
Output: vertex cover of $G$
$C \leftarrow edgeDeletion(V)$ //initialize candidate from
  approximation
$B \leftarrow$ |VC| // initialize the best seen to be the candidate
$U \leftarrow V$ // set of unassigned vertices in VC
$lb \leftarrow findLB(G, \text{VC}, U)$
$X \leftarrow \{lb, \text{VC}, U\}$ as root
$F \leftarrow \{X\}$
while $F \neq \emptyset$ do
    sort $F$ by lower bound value
    $X = \{lb, \text{VC}, U\} \leftarrow$ pop(F)
    $v \leftarrow$ the vertex in $U$ with minimum degree
    forall $C \in \{V, V \setminus v\}$ do
        if $isSolution(C)$ and $|C| < |B|$ then
            $B \leftarrow C$
        end
        $lb \leftarrow findLB(G, C, U)$
        if $C$ is not a dead end then
            if $lb < |B|$ then
                $F \leftarrow F \cup \{lb, C, U\}$
            end
        end
    end
end
return $B$
```

We chose to start the candidate solutions with the aforementioned edge deletion algorithm since this provides a good starting candidate. When initialized with a more promising vertex cover such as one produced by a limited local search or a better approximation, the branch and bound typically failed to find many if any improvements on medium and large instances within the time in which we ran it.

We chose to represent vertex covers as Python arrays, where a 1 indicates the presence of a vertex in a cover and a 0 indicates the absence of a vertex. Thus, all copy operations for vertex covers take $O(|V|)$ time and space. We also chose to represent graphs as sparse adjacency matrices via dictionaries of sets, where the outer dictionary is indexed by vertex numbers and the presence of a vertex in a set indicates an edge between the dictionary key and that vertex. Thus, each graph takes up roughly $O(|V|+|E|)$ space, query operations take amortized constant time, and iteration operations take $O(|V|+|E|)$ time. We use these data structures to represent vertex covers and graphs in both local search approaches, as well as one of our approximation approaches.

Checking if a solution is a dead end takes $O(|E|+|V|)$ time, since each edge needs to be checked to see if it is isolated. Computing a lower bound takes $O(|V|+|E|)$ time since we compute it by finding the subgraph for which no assignments have yet been made, taking $O(|V|+|E|)$ time, and adding the current number of assigned vertices to half the number of vertices assigned by the 2-approximation algorithm on the subgraph, also taking $O(|V|+|E|)$ time. Sorting the heap-backed frontier takes $O(\log n)$ time as it is extended, where n is the current number of items in the frontier. Verifying solutions takes $O(|V|+|E|)$ time since each edge must be checked for coveredness. All other operations, such as copies or computing vertex cover sizes, run in $O(|V|)$ time or less. Thus, the overall runtime of each step is $O(|V|+|E|+\log n)$.

The branch and bound algorithm has exponential time complexity and only performs well with small networks. For very large networks the branch and bound algorithm takes a long time without approaching the optimal solution. This motivates solving the problem using the local search and approximation approaches which we discuss later.

4.2 *Local Search*

In order to test a local search approach to solving vertex cover, we implement our own variation on randomized hill climbing, as well as FastVC, as proposed in [6].

4.2.1 Hill Climbing

Our hill climbing algorithm parallels the traditional hill climbing approach, where a model, in this case a vertex cover, is maintained and some subset of its neighbors are randomly examined and moved to if they are better or only slightly worse. Our variant actually uses two neighbor selection techniques for two different scenarios. The first scenario is when the vertex cover candidate being examined is already a solution; in this scenario, many random vertices are examined until one is found that when removed leaves the candidate as a solution. We call the configuration with just one vertex removed a close neighbor. We perform the check to see if the vertex cover is still a solution immediately because it can be done quickly by examining the removed vertex's neighbors, rather than re-verifying the entire solution in $O(|V| + |E|)$ time.

If no close neighbors which are solutions are found, then we use the second neighbor selection algorithm: if the current candidate is already a solution, then we randomly remove between one and three of its vertices, and otherwise add one to three random vertices to the candidate. Thus, we have implemented something similar to a simultaneous 1, 2 and 3 exchange. This enables the algorithm to pass through configurations which are unreachable by only passing through solutions and also helps to escape local maxima where further improvements cannot be made. Neighbors are ranked by their fitness, which we have defined to be $f = E_c - V_c + V$, where E_c is the number of covered edges, V_c is the number of vertices in the vertex cover and V is the total number of vertices. This can be maximized by increasing the number of covered edges and decreasing the number of vertices in the cover. Note that the number of vertices has simply been added to keep the value positive at all times.

Our hill climbing incorporates randomness into its choices of successor states in the second neighbor selection scenario in order to be able to escape from local maxima when they are encountered. Even if a configuration is worse than a previous configuration, it still may be chosen with a relatively high probability. The reason we chose for the probability to be high is that the second scenario is only reached when no close neighbors can be found from a given position, meaning that both finding a neighbor with a one to three vertex difference and backtracking are both likely to be good steps. We again chose to initialize candidate solutions by using the edge deletion algorithm, since this algorithm is a 2-approximation and generates solutions which are relatively easy to improve upon for local search.

Again, we chose to represent a vertex cover as a Python array where the 0 or 1 value at an index indicates the presence or absence of that vertex from the vertex cover. We have also chosen to represent the graphs as sparse adjacency matrices, due to their compromise between space and access efficiency. For finding close neighbors, we examine 5000 random possible close neighbor candidates one at a time using a Python generator to produce new candidates. We only update the fitness value for close neighbors at random intervals with a probability of 0.01 since the sum of the ones in the vertex cover is also a reasonable comparison function when comparing a solution to another solution and takes only $O(|V|)$ to compute. This is as opposed to the $O(|V| + |E|)$ time to compute our proper fitness function, or the $O(|V| + |E|)$ time required to verify that a candidate is a vertex cover by checking that each edge is covered. We always evaluate the fitness for regular neighbors since the quality of the neighbor chosen is more important than the time taken to find the neighbor.

Hill Climbing Algorithm

```
Input: graph $G = (V, E)$
Output: vertex cover of $G$
$C \leftarrow edgeDeletion(G)$ //initialize candidate from
  approximation
$B \leftarrow C$ //initialize the best seen to be the candidate
while $elapsed\ time < cutoff$ do
    if $\exists N \in CloseNeighbors(C)$ s.t. $N$ is a VC then
        if $|N| < |B|$ then
            $C \leftarrow N$
            $B \leftarrow N$
        end
    else
        for $N \in Neighbors(C)$ do
            if $Fitness(N) > Fitness(B) \cdot Random$ then
                $C \leftarrow N$
                if $N$ is a VC and
                  $Fitness(N) > Fitness(B)$ then
                    $B \leftarrow N$
                end
            end
        end
    end
end
return $B$
```

4.2.2 FastVC

We have chosen to implement the FastVC algorithm as our second local search technique. We select this algorithm because it quickly achieves state of the art performance on even very large graphs [6]. It is also a good, straightforward benchmark against which to compare our randomized hill climbing approach.

To explain this algorithm, we first define the loss of a vertex in a vertex cover to be the number of edges which would become uncovered if that vertex were removed from the cover, and the gain of a vertex not in a vertex cover to be the number of edges which would be newly covered if that vertex were added to the cover. Essentially, the algorithm constructs an initial vertex cover candidate by adding the more connected vertex of each uncovered edge in the graph, covering edges as vertices adjacent to them are added, then removing any vertices whose loss is zero. Then, iterations are performed where the current candidate set of vertices has the vertex with minimum loss removed if it is a vertex cover, and has the vertex of greater gain from a random uncovered edge added to the vertex cover while the vertex with the minimum loss (with a high probability) is removed.

Losses and gains are recomputed as vertices are added and removed from the candidate. The algorithm returns the best candidate candidate after the allotted time for iterations has expired. The full pseudocode for the algorithm has been provided below:

```
FastVC Algorithm
Input: graph $G = (V, E)$
Output: vertex cover of $G$
$C := ConstructVC()$
$gain(v) := 0$ for each vertex $v \in C$
while elapsed time < cutoff do
    if $C$ covers all edges then
        $C^* := C$
        remove a vertex with minimum loss from $C$
        continue
    end
    $u := ChooseRmVertex(C)$
    $C := C \setminus \{u\}$
    $e :=$ a random uncovered edge
    $v :=$ the endpoint of $e$ with greater gain, breaking
      ties in favor of the older one
    $C := C \cup \{v\}$
end
return $C^*$
```

The ConstructVC and ChooseRmVertex operations are defined as follows:

```
ConstructVC Algorithm
Input: graph $G = (V, E)$
Output: vertex cover of G
$C := \emptyset$
forall $e \in E$ do
    if e is uncovered then
        add the endpoint of e with higher degree into C
    end
end
forall $e \in E$ do
    if only one endpoint of e belongs to C then
        for the endpoint $v \in C$, $loss(v) + +$
    end
end
forall $v \in C$ do
    if $loss(v) = 0$ then
        $C := C \setminus \{v\}$, update loss of vertices in $N(V)$
    end
end
return $C$
```

```
ChooseRmVertex Algorithm
Input: $V$ the vertices of $G = (V, E)$
Output: a vertex in $V$
$best :=$ a random vertex from $V$
for iteration := 1 to 50 do
    $v :=$ a random vertex form $V$
    if $loss(v) < loss(best)$ then
        $best := v$
    end
end
return best
```

Again, we represented the vertex covers as binary arrays and the graphs as sparse adjacency matrices. Updating solutions takes $O(|V|)$ time since the solutions need to be copied. Updating losses and gains takes $O(\Delta)$ time, where Δ is the maximum degree of any vertex in the graph, since only the losses and gains of neighbor vertices need to be updated. Choosing a vertex with approximately minimum loss takes $O(|V|)$ time since losses are precomputed, a constant 50 candidates are considered, and getting a list of candidates which are in the vertex cover takes $O(|V|)$ time. Getting an uncovered edge takes $O(|E|)$ time, since all of the edges may need to be examined. Verifying whether or not a cover is a solution again takes $O(|V| + |E|)$ time since the same procedure as in the other algorithms is used. Thus, each iteration takes approximately $O(|V| + |E|)$ time.

4.3 *Approximation*

We have implemented the Maximum Degree Greedy (MDG), Greedy Independent Cover (GIC), and Edge Deletion (ED) algorithms, as well as one modified greedy (MG) algorithm in order to find approximate solutions. The algorithms for MDG, GIC and ED are implemented as defined in [3].

The edge deletion algorithm is the one provided in class and it selects a random edge and adds the vertices of the edge to the vertex cover at each time step. For this algorithm, the worst-case approximation ratio is 2. It finds an arbitrary edge at each time step so its overall time complexity is $O(|V| + |E|)$.

```
Edge Deletion (ED) Algorithm
$C \leftarrow \emptyset$
while $E \neq \emptyset$ do
    $C \leftarrow C \cup \{u, v\}$
    remove from E every edge incident to either u or v
end
return $C$
```

The maximum degree greedy algorithm selects a vertex of maximum degree in an uncovered graph and adds it to the vertex cover at each step. Its worst-case approximation ratio is $H(\Delta)$, with $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ where Δ is again the maximum degree of the graph. The time complexity of finding the vertex of maximum degree is $O(|V| + |E|)$. Since the algorithm adds one vertex at each step, the total time complexity is $O(|V|^2 + |VE|)$.

```
Maximum Degree Greedy (MDG) Algorithm
$C \leftarrow \emptyset$
while $E' \neq \emptyset$ do
    select a vertex u of maximum degree
    $V \leftarrow V - \{u\}$
    $C \leftarrow C \cup \{u\}$
end
return $C$
```

The greedy independent cover algorithm selects a vertex of minimum degree and adds all of that vertex's neighbors to the vertex cover in each timestep. Its worst-case approximation ratio is at least $\frac{\sqrt{\Delta}}{2}$. As before, the time complexity of finding a vertex with maximum degree is $O(|V| + |E|)$ and in the worst case, the algorithm adds one vertex at each time, so the total time complexity is $O(|V|^2 + |VE|)$

```
Greedy Independent Cover (GIC) Algorithm
N(u) is the set of neighbors of the vertex u
$C \leftarrow \emptyset$
while $E' \neq \emptyset$ do
    select a vertex u of minimum degree
    $C \leftarrow C \cup N(u)$
    $V \leftarrow V - (N(u) \cup \{u\})$
end
return $C$
```

For the ED, MDG and GIC algorithms, we used the NetworkX package as it contains a data structure designed for graphs, digraphs, and multigraphs and provieds many standard graph algorithms. We first construct graphs using the input data and selecting an arbitrary edge or certain vertices according to the algorithm we use. Then, we just use the remove_node or remove_nodes_from functions to reconstruct the graph. When a vertex is removed from the graph, the edges connecting to that vertex are automatically removed from the graph. While it is very convenient for us to use NetworkX, it does appear to increase the running times of some of these algorithms.

We have also implemented our own greedy algorithm with a heuristic based on the connectivity of a vertex in comparison with the lack of connectivity of its neighbors. We define the score of a vertex to be its degree divided by the minimum degree of any of its neighbors. Intuitively, this corresponds to choosing vertices which have high degree but also have neighbors which are comparatively unlikely to have other neighbors, which should do well to select vertices which eliminate many edges, in particular edges which would only otherwise be covered by relatively bad vertex choices. At each step, we simply remove the vertex with the highest score and its corresponding edges. The full pseudocode for the algorithm is presented below.

```
Modified Greedy (MG) Algorithm
Let $score(u) = deg(u)/\min_{v \in N(u)} deg(v)$
$C \leftarrow \emptyset$
while $E' \neq \emptyset$ do
    Select a vertex such that its score is a maximum
    $C \leftarrow C \cup u$
    $V \leftarrow V - u$
    Remove from E every edge incident to u
end
return $C$
```

Unlike the other approximation algorithms, we chose to implement this one with the previously discussed binary arrays as vertex covers and sparse adjacency matrices as graphs. With this representation, finding the degree of a vertex takes constant time, and finding the minimum degree of any of its neighbors takes $O(\Delta)$ time. Removing all of the neighbors of a vertex takes $O(\Delta)$ time. Thus, selecting the vertex of maximum score takes $O(|V|\Delta)$ time. This algorithm selects $O(|V|)$ vertices, and thus runs in $O(|V|^2\Delta)$ time in the worst case, though it tends to run much faster than this in practice due to the high quality of the vertex choices it makes.

5. EMPIRICAL EVALUATION

5.1 Experimental Procedure

In this section we evaluate the performance of each algorithm when applied to each of the eleven networks.

All tests were conducted on a machine running Ubuntu 16.04 with 11.6 GiB of RAM and an Intel Core i7-5500U CPU with a 2.40GHz clock speed and 4 hardware threads. We used the default cpython interpreter for Python version 3.5.2.

For the approximation algorithms, we ran each approximation algorithm on each graph one at a time, since the total time to run the tests is short. However, for the local search and branch and bound algorithms, we ran the algorithms on 4 instances at a time using a bash script, meaning that each algorithm would occupy one thread on the testing machine with relatively minimal slowdown. The hill climbing algorithm was run for 600 seconds, since few improvements were made for any algorithm after the first minute or so. The FastVC algorithm was run for 1000 seconds, because some improvements were made later than several minutes into testing. For both local search algorithm, 12 random seeds were used, and the results are averaged to produce the final numbers. The branch and bound algorithm was run for 3600 seconds, since multiple vertex covers continued to improve past 1000 seconds.

5.2 Results

The performance of each algorithm is calculated based on the deviation of the algorithm solution from the optimal solution, i.e., $(Alg - Opt)/Opt$. The results of the algorithms are reported in table 4. It is evident from these results that the approximation algorithm is by far the best approach, with the local search approaches also performing very well. In general, the branch and bound algorithm performs very poorly when compared to the other algorithms.

Branch and bound performs particularly badly on large graphs such as as-22july06 where it achieves nowhere near the vertex cover quality of the other algorithms. It only performs satisfactorily on very small networks such as karate and football, where it finds an exact and a nearly exact solution, respectively. While the branch and bound is the only algorithm of the group which is guaranteed to find an optimal solution given enough time, it is clear from the length of time after which answers were still improving that a very, very large amount of time would be required to find comparable solutions to the other algorithms, much less optimal solutions.

Our modified greedy approximation algorithm performs extremely well and provides almost exact solutions for many small and medium network instances such as jazz, karate, and netscience, as well as the very large as-22july06. In all but two instances where it is slightly outperformed by hill climbing or FastVC, it outperforms each of the other approaches, sometimes by significant margins. In fact, it never has more than a 5.5% error on any graph, and achieves less than 1% error on most networks.

Table 2: Comparison of different approximation algorithm vertex cover sizes

Graph	ED	MDG	GIC	MG
as-22july06	6040	3307	3303	3303
delaunay_n10	928	737	715	714
email	838	605	597	596
football	108	96	95	95
hep-th	5770	3944	3930	3928
jazz	182	159	159	158
karate	22	14	14	14
netscience	1202	899	899	899
power	3620	2277	2204	2207
star	10202	7374	6946	7282
star2	6834	4697	4572	4557

Table 3: Comparison of different approximation algorithm running times in seconds

Graph	ED	MDG	GIC	MG
as-22july06	104.19	108.24	372.35	16.73
delaunay_n10	0.55	0.53	0.17	0.56
email	0.66	0.55	0.37	0.51
football	0.01	0.01	0.00	0.01
hep-th	23.40	32.66	20.23	15.08
jazz	0.04	0.02	0.01	0.04
karate	0.00	0.00	0.00	0.00
netscience	0.56	0.97	0.52	0.65
power	6.71	12.22	7.58	4.76
star	62.86	85.00	36.66	59.05
star2	79.26	93.40	116.30	33.77

As shown in Table 2, for approximation algorithms, MDG, GIC, and MG work better with more accurate results than ED. As the approximation ratio of ED is 2, the true optimum should be at least half of the result of ED. For example, on the email graph, ED gives the result 838 vertices as the vertex cover size. The optimal solution is 594 vertices which is more than half of 838. For all graphs given in the experiment, the optimal solution follows such rules. For the MDG, GIC and MG algorithms, however, the results approach the true optimum. Our modified greedy algorithm is on average faster than or comparable in speed to all of the other approximation algorithms, except for GIC, where it terminates later on the star network. It is worth noting that some of the performance differences may have arisen from the use of our custom graph data structures for MG as opposed to the other approximation algorithms where we used NetworkX, though more benchmarks would be needed to determine more precisely the cause of the performance differences. While we were not able to come up with a proof of an approximation ratio for MG, it consistently outperforms Edge Deletion algorithm and achieves comparable accuracy to the similar $H(\Delta)$-approximation MDG algorithm. Thus, we we feel safe in saying that it is at worst a 2-approximation algorithm, and, again, this bound appears loose when applied to the provided practical instances.

Both of our local search algorithms performed quite well. The hill climbing algorithm performed the best on the small football, jazz, and karate graphs (it being the only algorithm to find an exact solution to football), though it performed somewhat worse comparatively on some of the harder graphs, e.g., delaunay_n10 and star, where it had between a 4% and 9% relative error on average. The FastVC algorithm performs at least slightly better than hill climbing on all instances except for football, and has its most significant improvement for the star graph, where it has only a 4% error in contrast to the 8.9% error that hill climbing has. The time of last improvement for both algorithms varies significantly for both algorithms, though it is lower on average for FastVC.

To further analyze the performance of the two local search algorithms we generated QRTD and SQD plots. A QRTD plot shows the probability of obtaining certain solution quality within some time. Similarly, a SQD plot shows the probability of different qualities achieved for a fix amount of running time. The plots are illustrated in Figure 2 and Figure 3 for hill climbing and in Figure 4 and Figure 5 for FastVC.

As is illustrated in Figure 2, the hill climbing algorithm has a wide range of running time for different networks. For example, the running time range for the as-22july06 network is about 200 seconds, but for the karate network it is about 0.3 seconds. However, it is worth noting that for most of the graphs the algorithm can very quickly produce a vertex cover of size $1.1 \cdot opt$ solution with high probability for all graphs. For the star and star2 networks, the hill climbing algorithm reaches to $1.1 \cdot opt$ with probability of one. Figure 4 shows the QRTD plots for FastVC algorithm. Similar to hill climbing, for most of the graphs the algorithm can reach to $1.1 \cdot opt$ solution with some positive probability and for all networks the algorithm produces vertex covers of size at most $1.1 \cdot opt$.

Additionally, we generated boxplots for the running times of two local search algorithms, as shown in Figure 6 and Figure 7. While we ran hill climbing for 600 seconds and FastVC for 1000 seconds, the boxplots are based on the time that the algorithm last finds a new solution. From these plots, it is clear that both local search algorithms find high quality solutions for the small and medium networks much earlier than the cut off time but fail to improve afterwards. For larger networks however, they consume most of the given time when finding better solutions.

6. DISCUSSION

We have considered several diverse algorithms for solving the vertex cover problem. In theory, the branch and bound algorithm solves the problem exactly, but has an exponential time complexity. As a result the algorithm is only suitable for small graphs. Finding a good lowerbound is essential for the performance of this algorithm, but difficult to do accurately for the provided instances. While we used the 2-approximation Edge Deletion algorithm to determine lower bounds, it became clear that the fact that the approximation is often much better than $2 \cdot opt$ causes this to be an impractically low lower bound on larger graphs.

The local search algorithms both perform well for all of the given networks and produce results within 9% relative error for large graphs and exact results for small graphs. Both algorithms have very low amortized time complexities for each iteration, making them both fast in addition to being typically accurate. In addition to the good performance of FastVC and hill climbing, the algorithms are easy to implement. Furthermore, hill climbing is easily modifiable to be suitable for other problems, since only the neighbor selection and quality assignment functions need to be changed in order to adapt them to new tasks. While at a glance, it

Table 4: Comprehensive table comparing different algorithms used for solving VC problem. The time column denotes the last time that the algorithm finds a new solution. If the BnB cannot improve the initial solution, the time is zero. For approximation algorithm we used our modified greedy results.

	Hill Climbing			FastVC			Approximation			Branch and Bound		
Dataset	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr	Time (s)	VC Value	RelErr
as-22july06	70.42	3332.08	0.009	57.29	3328.75	0.008	16.73	3303	0.0	3553.81	5947	0.800
delaunay_n10	365.89	743.92	0.058	1.79	736.92	0.048	0.56	714	0.016	0.0	966	0.374
email	312.68	616.5	0.038	1.00	609.58	0.026	0.51	596	0.003	45.72	781	0.315
football	26.96	94.	0.0	28.14	95.83	0.020	0.01	95	0.011	709.23	101	0.0745
hep-th	2.05	4039.5	0.029	34.29	3939.42	0.003	15.08	3928	0.001	1021.27	5097	0.298
jazz	329.72	158.	0.0	0.16	159.25	0.008	0.04	158	0.0	2405.27	178	0.127
karate	0.035	14.	0.0	0.00	14.	0.0	0.0	14	0.0	0.12	14	0.0
netscience	62.2	906.92	0.009	0.00	899.	0.0	0.65	899	0.0	17.65	1073	0.194
power	143.73	2336.08	0.060	6.14	2278.17	0.034	4.76	2207	0.002	278.28	3215	0.459
star	329.94	7515.25	0.089	279.33	7181.08	0.04	59.05	7282	0.055	914.54	8971	0.300
star2	108.10	4803.58	0.058	221.76	4815.42	0.06	33.77	4557	0.003	153.65	6774	0.491

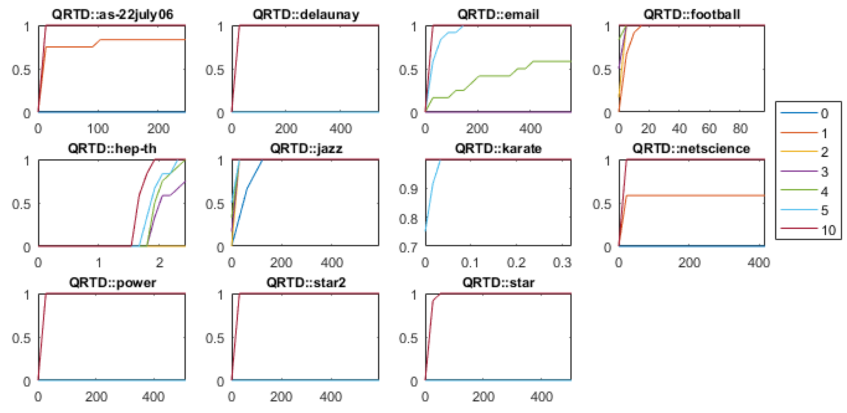

Figure 2: QRTD obtained for the hill climbing algorithm

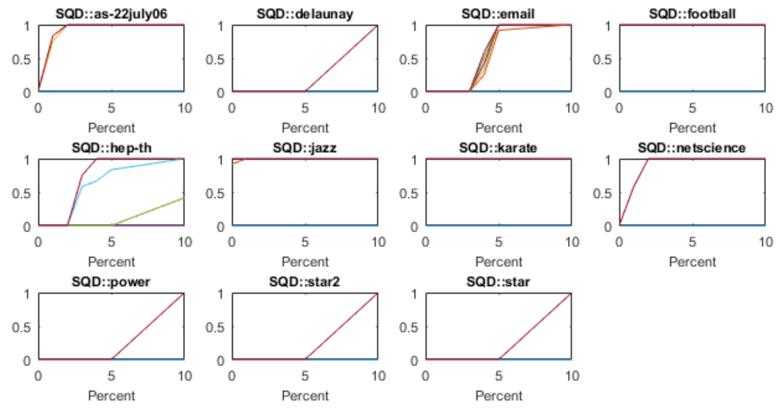

Figure 3: SQD obtained for the hill climbing algorithm

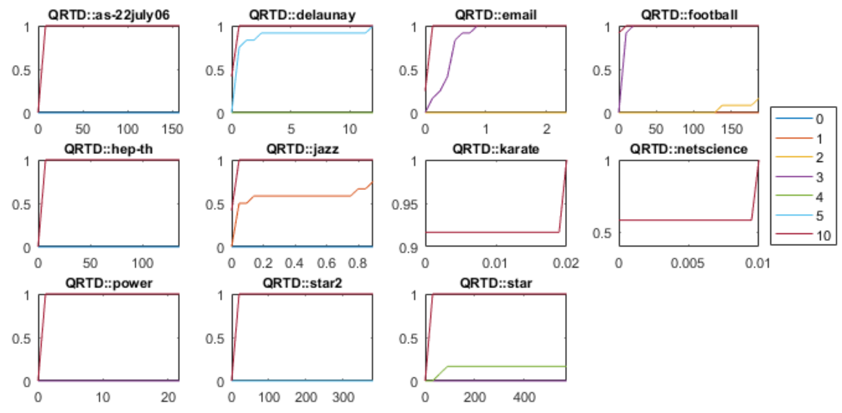

Figure 4: QRTD obtained for the FastVC algorithm

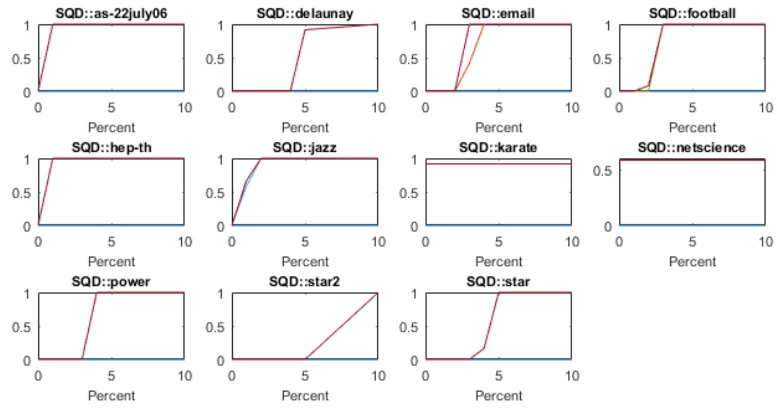

Figure 5: SQD obtained for the FastVC algorithm

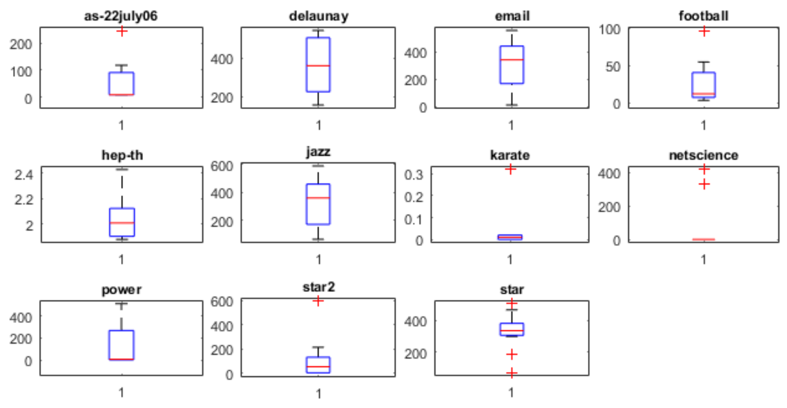

Figure 6: boxplot of running time obtained for the hill climbing algorithm

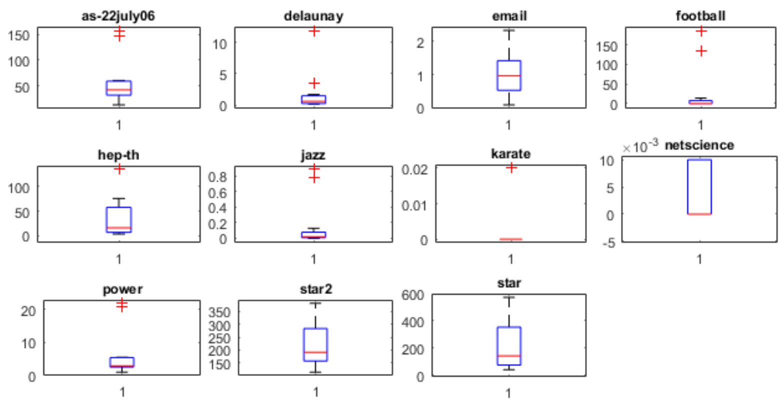

Figure 7: boxplot of running time obtained for the FastVC algorithm

may seem that FastVC finds solutions much more quickly than hill climbing in some instances, this is actually because hill climbing is sometimes able to find improvements after significant time due to its ability to escape local minima via 1, 2, and 3-exchanges. This is a particular strength of the algorithm which indicates that perhaps solutions could continue to improve over time.

We also demonstrated the result of four approximation algorithms. The time complexities of ED, MDG, GIC and MG algorithm are $O(|V|+|E|)$, $O(|V|^2+|VE|)$, $O(|V|^2+|VE|)$ and $O(|V|^2\Delta^2)$, respectively. In reality, MDG, GIC and MG run at least as fast as ED algorithm and produce more accurate results than ED due to better vertex choices. It is likely that the improvement in speed is due to termination in fewer iterations, as a result of the good vertex choices covering more edges per iteration. However, ED is easier to implement as it simply removes one random edge every time without calculating the degree of the vertex.

7. CONCLUSION

We have examined the minimum vertex cover problem has been examined in detail. In order to find the MVC for a network, three class of algorithms, branch and bound, approximation, and local search, were employed. The performance of each of the algorithms was evaluated by employing each algorithm on eleven real-world graphs with different sizes. The following are the main observations for each of the algorithms:

- Branch and Bound: The branch and bound algorithm only performed well on small networks such as Karate network. It failed to produce a near-optimal solution even within an hour of running time for large networks.

- Hill climbing algorithm with randomization: With randomizations and relatively large neighborhoods, the hill climbing algorithm was sometimes able to effectively escape local minima or essentially restart, yielding solution improvements beyond those possible with a naïve approach.

- The fastVC algorithm performs extremely well comparing to other algorithms and provide results within few percentage points (5%) of the optimal solution.

- All the four approximation algorithms perform well compared to the the branch and bound. In particular, the MG and GIC algorithms performed extremely well for all the network instances and produced results within (5%) of the optimal.

In practice, a local search or approximation approach would certainly be preferable to a branch and bound approach, given these results. Furthermore, approximation algorithms would be our algorithms of choice, given their comparatively short runtimes, approximation bounds, and practically low relative error.

8. REFERENCES

[1] Shaowei Cai, Kaile Su, Chuan Luo, and Abdul Sattar. Numvc: An efficient local search algorithm for minimum vertex cover. *Journal of Artificial Intelligence Research*, 46:687–716, 2013.

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

[3] François Delbot and Christian Laforest. Analytical and experimental comparison of six algorithms for the vertex cover problem. *Journal of Experimental Algorithmics (JEA)*, 15:1–4, 2010.

[4] Shaowei Cai, Kaile Su, and Abdul Sattar. Local search with edge weighting and configuration checking heuristics for minimum vertex cover. *Artificial Intelligence*, 175(9-10):1672–1696, 2011.

[5] Shaowei Cai, Kaile Su, and Abdul Sattar. Two new local search strategies for minimum vertex cover. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[6] Shaowei Cai. Balance between complexity and quality: Local search for minimum vertex cover in massive graphs. In *IJCAI*, pages 747–753, 2015.