ECE650 - METHODS AND TOOLS FOR SOFTWARE ENGINEERING

UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Final Course Project

Authors:
Akshay Rakheja (ID: 20471078)
Piyush Jha (ID: 20879185)

Date: April 19, 2021

1 Introduction

A vertex cover of a graph is the set of vertices in a graph in such a way that each edge is incident to at least one vertex of the set. In this project, we compare the performances of three different algorithms to find the minimum vertex cover:

1. CNF-SAT-VC: The approach uses a polynomial-time reduction to CNF-SAT, and then uses a SAT solver to get the final results.
2. APPROX-VC-1: In this approach, we pick a vertex of the highest degree (most incident edges), add it to the vertex cover and then throw away all edges incident on that vertex. This process is repeated till no edges remain.
3. APPROX-VC-2: In this approach, we pick an edge $< u, v >$, add both u and v to the vertex cover and throw away all edges attached to u and v. This process is repeated till no edges remain.

Minisat [2] has been used as the SAT solver for CNF-SAT-VC, which always gives us the optimal solution. However, the other two algorithms may not always give us the minimum vertex cover. In order to compare the three algorithms, we analyze their running time and approximation ratio, where the approximation ratio is the ratio of the size of the computed vertex cover to the size of an optimal (minimum-sized) vertex cover.

2 Methodology and improvements

We used the graph generator provided to us to create 10 graphs which were run 10 times each before calculating the mean and standard deviation. For V=25 in CNF-SAT-VC, we only ran 3 graphs only 2 times due to the high runtime (~5 hrs/run) of the CNF-SAT-VC algorithm. However, in the current program, we have reduced the timeout to be 30 minutes.

For APPROX-VC-2, we have not introduced a random component, instead, we have picked up the edges sequentially as they are stored in the adjacency matrix. We found that although CNF-SAT-VC produces the most optimal vertex cover for a graph, its running time increased exponentially with an increase in the number of vertices (details have been discussed in the further sections). However, in order the improve the runtime performance for certain scenarios, we apply two improvements as described below:

(1) If the graph is a binary tree: We first check if the input graph is connected and there are no cycles present in it. We did this by creating an adjacency list. Connectivity can be confirmed by traversing the graph and checking if all the vertices are reachable. For detecting cycles for our undirected graph, we look for a visited vertex v, adjacent to u such that u is already visited and u is not a parent of v. [3] In such a scenario there is a cycle in the graph and thus, it is not a tree. In the next step, we traverse the tree to store it in a pointer representation where we also check if the given tree is a binary tree or not. We assume the first vertex-id in the first edge pair of the input edge set to be our root node.

Once we have verified that the input graph is a binary tree, we use $O(n^2)$ memoization-based approach to calculate the minimum vertex cover. In this bottom-up approach, we store the solutions of our subproblems and calculate the minimum vertex cover (in the variable vc) for each node. This value represents the minimum vertex cover of the partial tree rooted at that node. In a bottom-up approach, we start from the leaf nodes, compute its minimum vertex cover and store it to that node. Now for the parent node, we recursively compute the vertex cover from its children nodes and store it to that node. In this approach, we calculate the vertex cover of any node only once and reuse the stored value if required. In order to calculate the minimum vertex cover, we take the minimum of the two conditions: (1) Root is a part of vertex cover: In this case, root covers all children edges. We recursively calculate the size of vertex covers for left and right subtrees and add 1 to the result (for root). (2) Root is not a part of vertex cover: In this case, both children of the root must be included in vertex cover to cover all root to children edges. We recursively calculate the size of vertex covers of all grandchildren and the number of children to the result (for two children of the root). Once, we have completed this iteration of finding the minimum vertex cover (vc) value for every node, we need to find the nodes/vertex id corresponding to the minimum vertex cover. In order to do this, we traverse the tree which has the vc values stored as an integer. We compare the value of vc of the parent node with their 2 child nodes (if they exist) to determine whether the parent node is the part of the minimum vertex cover. Therefore, if the input graph is a binary tree, the minimum vertex cover can be obtained in polynomial time. [1]

(2) If the graph is not a binary tree: In this case, we can't use the above technique and would have to stick to the solution given by Minisat. However, in order to determine the solution, we were initially linearly searching k in the range of $[1, V]$, in order to find a SAT solution, where V is the total number of vertices. However, we can reduce this search space of k. As described in the further sections, we find that APPROX-VC-1 was resulting in a vertex cover size close to that of the optimal solution. Therefore, we can use the value calculated by APPROX-VC-1 as the new upper limit for the search space of k, and then use a binary search to find the optimal value of k. We compared this heuristic for different combinations of graphs and found an interesting pattern. If the size of the minimum vertex set is higher and closer to the value of V, the speedup is around 1.3-1.5 times. However, if the value of k is smaller, there is a negligible difference in performance.

3 Analysis

For the three algorithms used to experiment and solve the vertex cover problem, we decided to run two kinds of analysis:

1. Running Time Analysis
2. Approximation Ratio Analysis

3.1 Running Time Analysis

We used the `pthread_getcpuclockid()` to capture the elapsed time for each thread used for each algorithm. Figures 1, 2 and 3 below, depict the performance results we observed for the three algorithms. As we can see in Figure 1, CNF-SAT-VC took the longest amount of time for any vertex of size V. It took over 2.5E+10μs for CNF-SAT-VC to compute the vertex cover for 25 vertices while both the approximate algorithms were able to compute it under 100μs. We used a logarithmic y-axis to show that the time taken increased exponentially as the number of vertices increased and we were timed out after 25 vertices. This can be attributed to the increase in the number of clauses added for each vertex and edge of the graph during our reduction to the CNF-SAT form. The algorithm needed to check if these increasing number of clauses were satisfied as the vertices and edges of the graphs increased. We know that the approximate time complexity of our optimal CNF-SAT algorithm is exponential and that the Vertex Cover problem is an NP-complete problem which further bolsters our observation of longer running time when increasing the number of vertices for our optimal CNF-SAT-VC solution.

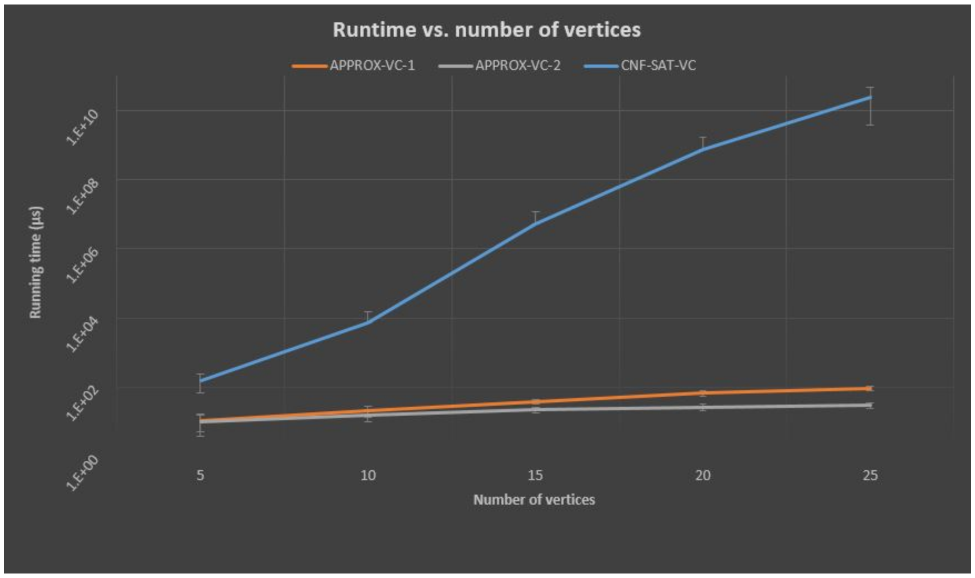

Figure 1: Runtime performance of different vertex-cover algorithms.

Another interesting observation from this figure is the error bars/standard deviation are almost negligible for the approximate algorithms while we see it increasing significantly for the CNF-SAT algorithm. We tried to dive deeper into this observation by plotting the running time of CNF-SAT-VC on a linear y-axis in Figure 2. Here we witness the variation in running time increase massively after 20 vertices. This means that the running time for the optimal solution varied more as the number of vertices increased. We believe this can be attributed to the difference in the complexity of graphs which means that a graph with fewer edges will have fewer clauses to satisfy than a graph with more edges for the same number of vertices in the graph. In the figure below we also notice the 100 fold jump in running time for 25 vertices due to the exponential nature of the time complexity.

In Figure 1, for upto 25 vertices, the approximate algorithms are consistently similar in terms of time taken and hence their plot looks linear and almost overlapping in Figure 1. The difference in the running time between the two approximate algorithms becomes more apparent in Figure 3. In order to further explore the difference between APPROX-VC-1 and APPROX-VC-2 we decided to benchmark the algorithms with graphs containing upto 100 vertices. You can

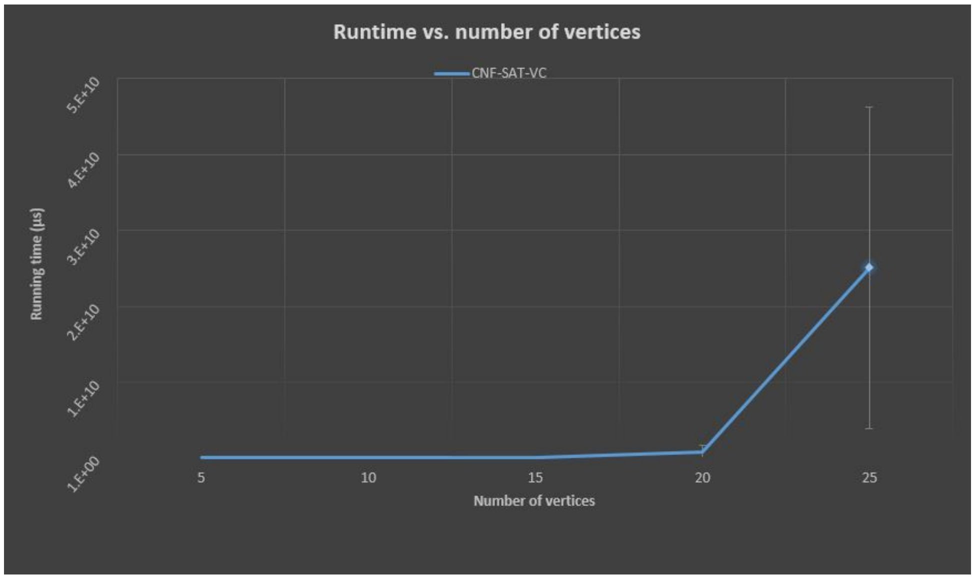

Figure 2: Runtime performance of CNF-SAT-VC algorithm.

see that for graphs with upto 25 vertices the running time for both approximate algorithms is almost the same. This is one of the 2 reasons why we see the two algorithms almost overlap in Figure 1. The other reason being that the graph is logarithmic which makes the minor difference in running time (upto 25 vertices) even more minute. For graphs with more than 25 vertices the algorithms diverge with APPROX-VC-1 taking longer to run. We observe APPROX-VC-1 computes the Vertex cover for 100 vertices in a graph in average time of 3816μs while APPROX-VC-2 takes only 559μs to compute it for the same graphs. This is due to different run time complexities of the two approximate algorithms. Another big difference between the two approximate algorithms in Figure 3 is the standard deviation in running times. There is far more variation in running times of APPROX-VC-1 than APPROX-VC-2. We believe this might be due to the 'greedy' nature of the algorithm.

3.2 Approximation Ratio Analysis

Another important benchmark to consider when comparing the algorithms is the approximation ratio. It is defined as the ratio of the size of the computed vertex cover to the size of an optimal (minimum-sized) vertex cover. It can be represented in the following equation:

$$\frac{Sizeof\,VertexCover(ApproximateAlgorithm)}{Sizeof\,VertexCover(CNF-SAT-VC)} \tag{1}$$

Since we know CNF-SAT-VC is the most optimal solution available, we use it to compare the vertex cover for other approximate algorithms. As seen in Figure 4, we have calculated the ratios for the two approximate algorithms. Here, ratio of 1 signifies that the algorithm is performing as good as the CNF-SAT-VC which is the most optimal solution available. These are the average ratios for the 10 graphs produced with 10 runs each. We can see that APPROX-VC-1 performs very close to the optimal solution (CNF-SAT-VC) with a ratio of 1.031 while APPROX-VC-2 has

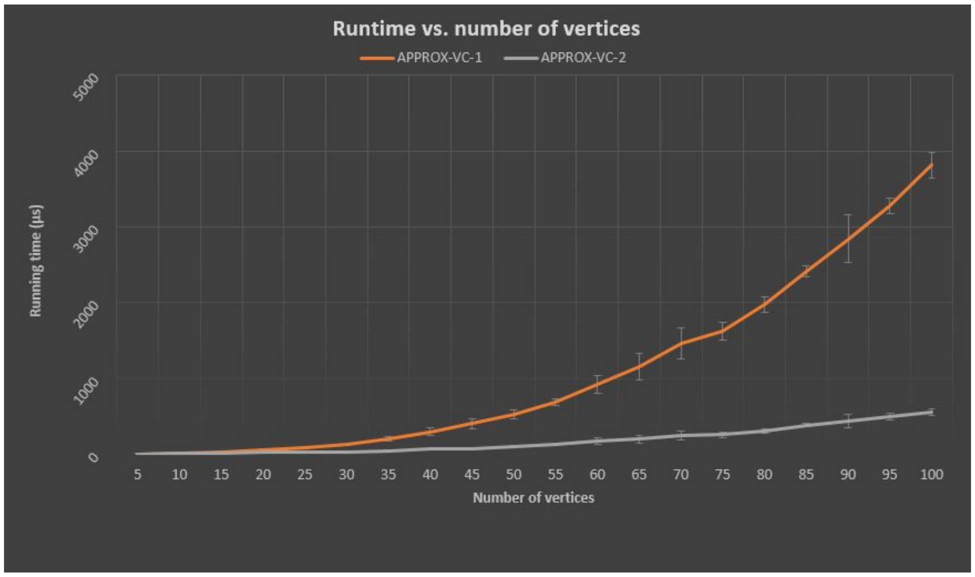

Figure 3: Runtime performance of approximate vertex-cover algorithms.

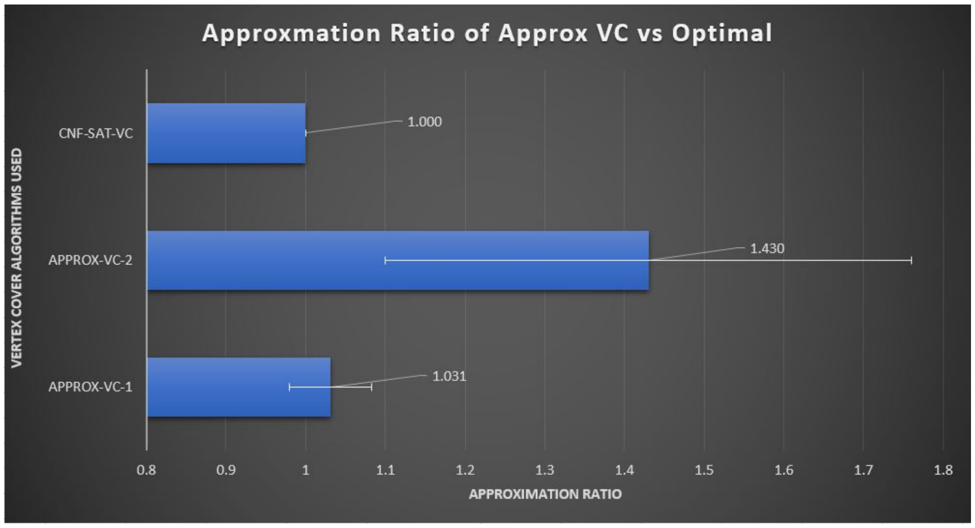

Figure 4: Approximation ratio of different vertex cover algorithms.

a nearly 40% higher ratio comparatively. We also notice the longer error bars in the case of APPROX-VC-2. This shows that the vertex cover's produced by the algorithm for different input graphs with the same number of vertices varied significantly.

In Figure 5, we try to deep dive into approximation ratios for vertices in the range [5,25]. We notice that APPROX-VC-1 performs as good as the CNF-SAT-VC for graphs with vertices 5 and 10 and slightly worse in the case of vertices in the range [15,25]. On the other hand, APPROX-VC-2 does very poorly for the entire range of vertices [5,25]. The standard deviation for the algorithm also increases significantly as the number of vertices in the graph increases.

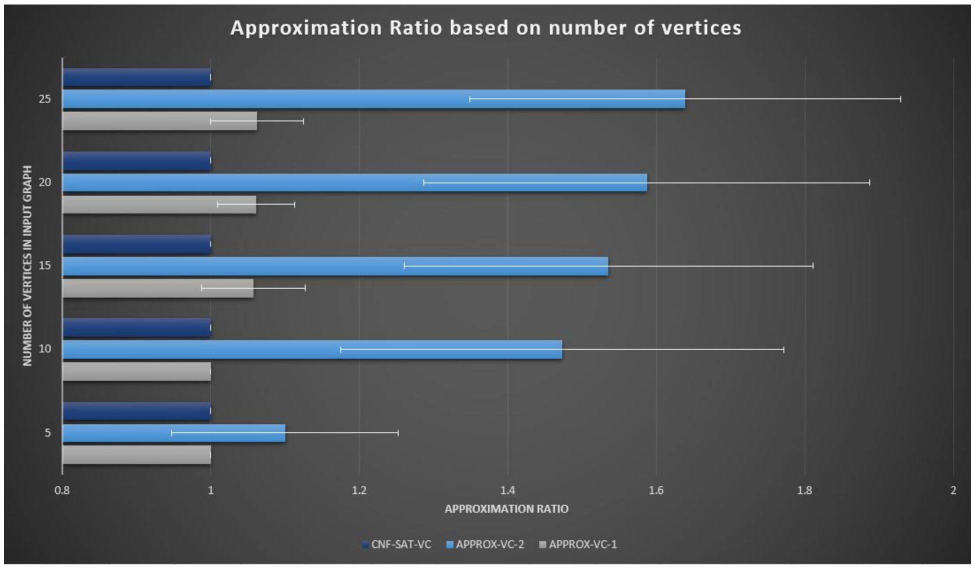

Figure 5: Approximation ratio for different input size.

4 Conclusion

In this project, we analysed 3 algorithms, CNF-SAT-VC, APPROX-VC-1, and APPROX-VC-2. We found that all three algorithms had their running time increase as the number of vertices in the graph increased. The increase was different for different algorithms and not all of them were equally efficient in producing the optimal vertex cover. We found that although CNF-SAT-VC produces the most optimal vertex cover for a graph, its running time increased exponentially with an increase in the number of vertices. Therefore, it is not an ideal solution for a graph with a large number of vertices.

On that note, we find both the approximate algorithms time-efficient as they produce their respective solutions in polynomial time. APPROX-VC-2 is the quickest in terms of time taken ($O(n)$), since it is simply iterating through all the edges to calculate the vertex cover. However, APPROX-VC-2 also has a high average approximation ratio of ~1.4, which means that it will not always produce the most optimal vertex cover. On the other hand, APPROX-VC-1 has an approximation ratio close to 1 (~1.03), which means that it produces an optimal solution or one very close to it. It also does this in polynomial time. Hence, this makes APPROX-VC-1 an ideal solution for a graph with a large number of vertices where a very small margin of error is acceptable.

As a future work, the improvement in the binary tree based approach can be extended to an n-ary tree. Furthermore, reduction in the number of clauses of CNF-SAT-VC based on certain scenarios can be implemented.

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009. pages 3

[2] Minisat. User guide. `https://www.dwheeler.com/essays/minisat-user-guide.html`. pages 2

[3] Vinit Verma. Check if a given graph is tree or not. `https://www.geeksforgeeks.org/check-given-graph-tree/`, 2020. pages 2