ECE650 - METHODS AND TOOLS FOR SOFTWARE ENGINEERING

UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Project: Analysis of Vertex Cover Problem with 2 Approx Algorithms

Authors:

Nick De Marchi
ID: 20848920

Gaurav Prasad
ID: 21082013

Date: December 2, 2023

Contents

Abstract

This project focuses on addressing the Vertex Cover Problem (VCP) which is a classical optimization challenge in computer science and engineering. Our methodology integrates the CNF reduction and SAT solvers to explore solutions for the VCP. Additionally, we employ two approximation techniques (APPROX-VC-1 & APPROX-VC-2) and contrast their performance against the CNF SAT solver. Our findings reveal that as the vertex value (V) that dictates the graph size increases, the CNF SAT solver experiences exponential time growth, indicative of scalability challenges. In contrast, the two approximation techniques demonstrate efficient computational scaling, following a linear trajectory. However, it is crucial to note that these approximations do not guarantee exact solutions to the minimum VCP. In terms of vertex cover size ratios, our analysis uncovers that the first approximation closely aligns with the CNF SAT solver, providing a noteworthy alternative in scenarios where computational efficiency is a priority, and an exact solution is not imperative. This study contributes valuable insights into the trade-off between computational efficiency and solution optimality for the VCP, offering practical implications for real-world problem-solving scenarios.

1 Introduction

The minimum VCP, which is NP-Complete, is an optimization problem that can be solved in polynomial time if the P = NP conjecture is substantiated. A vertex cover C is a subset of vertices V from an undirected graph $G = (V, E)$ (where V is the number of vertices and E is the set of edges) such

that for every edge $E = \{u, v\}$ of the graph, either u or v is in the vertex cover. Let us consider the following example in Figures 1 $\rightarrow$ 5 where we visualize and example of calculating a vertex cover:

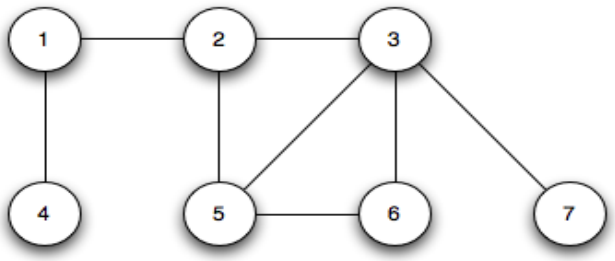

Figure 1: Iteration 1 - Choose edge [1,2] resulting in initial Vertex Cover Set $C = \{1, 2\}$ (edges [1,4], [2,3], and [2,5] are removed since these edges contain a vertex in our cover C [1].

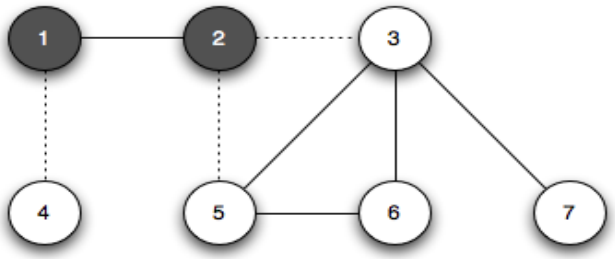

Figure 2: Iteration 2 - Choose edge [5,6] resulting in Vertex Cover set $C = \{1, 2, 5, 6\}$ (edges [2,5] and [3,5] removed as above) [1].

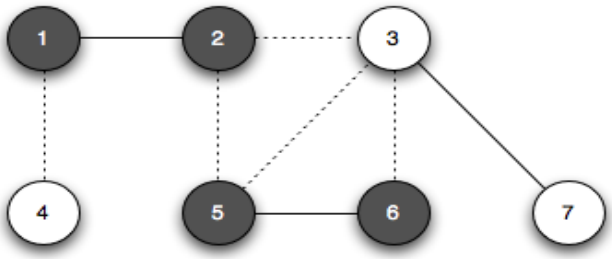

Figure 3: Iteration 3 - Choose edge [3,7] resulting in Vertex Cover set $C = \{1, 2, 3, 5, 6, 7\}$ (edge [3,6] removed) [1].

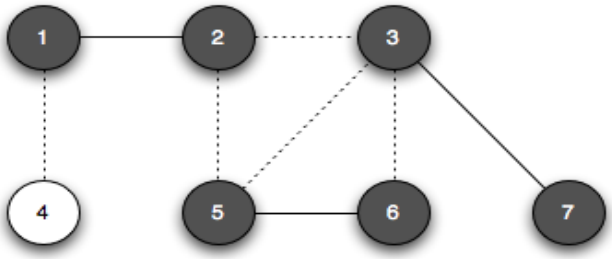

Figure 4: Approximate Vertex Cover $C_{approx} = \{1, 2, 3, 5, 6, 7\}$ with size 6 [1].

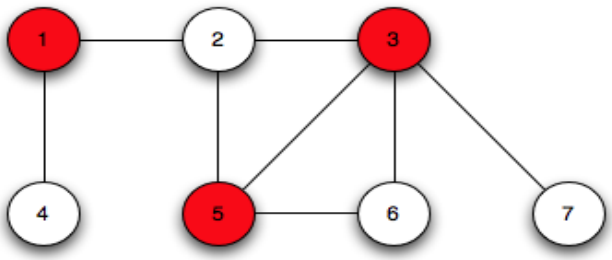

Figure 5: Considering the minimum vertices needed so that we touch all edges will give us the minimum Vertex Cover $C_{min} = \{1, 3, 5\}$ with size 3 [1].

2 Algorithms

a) CNF-SAT-VC : This algorithm is the primary solution we will use to find the minimum size k such that our CNF formula F is Satisfiable (SAT). Our CNF formula F is the reduction provided in assignment 4's encodings [5] and in order to find if F is SAT, we utilize the MinSat solver [6]. If F is SAT this implies our solver have found a vertex cover for our graph for a specific size k. In order to find the minimum vertex cover we employ a `BinarySearch` algorithm to find the smallest value of k such that F is SAT. This search algorithm works with our CNF reduction and sat solver algorithm (denoted `cnfReduction`) as follows:

Algorithm 1: Binary Search for Vertex Cover using cnfReduction

Input : Graph size n, Edges E
Output: Minimum Vertex Cover size and set $\{k, C\}$

Initialize: $low \leftarrow 1$ $high \leftarrow n - 1$;
while $low < high$ **do**
    mid $\leftarrow$ low $+ \frac{(\text{high}-\text{low})}{2}$;
    result $\leftarrow$ `cnfReduction(`n, mid, E`)`;
    if $result = True$ **then**
        low $\leftarrow$ mid $+ 1$;
    else
        high $\leftarrow$ mid;
$low \leftarrow k$;
return $\{k,$ `cnfReduction(`n, k, E`)`$\}$;

b) APPROX-VC-1 : This approximation algorithm is created by first choosing the vertex that occurs with maximum frequently in a set of edges E such that it is with highest degree of vertex V. This vertex is added to the vertex cover set C and we remove the edges that are attached to this vertex. This process is repeated until no edges remain. Below is the general step by step process for this algorithm:

Algorithm 2: Greedy Vertex Cover Algorithm 1: APPROX-VC-1

Input : Graph $G = (V, E)$
Output: Vertex cover C

Initialize an empty map M and degree $d = 0$;
while $\neg E.empty()$ **do**
    for u, v *in* E **do**
        $M[u]$ += 1;
        $M[v]$ += 1;
    for *entry in* M **do**
        `// Check if the count > current maximum degree`
        if *entry.second* $> d$ **then**
            `// Update max degree and corresponding vertex`
            d = entry.second;
            $value$ = entry.first;
    $C \leftarrow C \cup \{value\}$;
    Remove all edges in E incident on $value$;
$k \leftarrow C$.size();
Sort C in increasing order;
return *Vertex cover* C;

c) APPROX-VC-2 : This algorithm is created in which we choose an edge {u, v} from set of edges E and add them to Vertex Cover then we remove all edges {u, v} that are connected to both vertices. This process is repeated until we selected an edge and removed all of its adjacency until there exists no edges remaining.

Algorithm 3: Greedy Vertex Cover Algorithm 2: APPROX-VC-2

Input : Graph $G = (V, E)$
Output: Vertex cover C

Initialize unordered set C;
while $\neg E.empty()$ **do**
    Pick an edge (u, v) from E;
    Add both u and v to C;
    Remove edges attached to u and v from E;
$k \leftarrow C$.size();
Sort C in increasing order;
return $\{k, C\}$;

3 Experimental Details

For our project, we will take as input the output that is generated by the program `/home/agurfink/ece650/graphGen/graphGen` on eceubuntu. `graphGen` generates graphs G with edges E for a particular number of vertices V, but not necessarily the same edges each time it is executed. A shell script was created to generate graphs G for a particular vertex value V, where $V \in [5, 50]$. We increment V by a value of 5 for each output and use a minimum of 10 graphs per value of V [1]. For our main program, which will be named `ece650-prj`, we will create a multi-threaded script that will contain 4 threads [2], one for our input/output (I/O), and three for each of our algorithms. The I/O thread of our code should process the given inputs generated from the `graphGen` script and store the vertices and edges in two variables. We will then create three threads where each thread will run one of our three algorithms (i.e. one thread for each of CNF-SAT-VC,

[1] This means we should generate a minimum of 100 graphs for our values of V

[2] 5 threads in total if we consider the attempted improvement on our CNF SAT algorithm. This will be discussed in the Bonus section

APPROX-VC-1 and APPROX-VC-2). The output of our main program will do two things:

- We will print to console the vertex cover set for each of our algorithms for a particular input of the graph $G = (V, E)$.
- We will save to an output file, denoted *output.csv*, the performance of our algorithms. This will include the calculation of the run-time for each algorithm and the ratio of the vertex cover size for our two approximation algorithms relative to the base CNF SAT output.

4 Analysis

To validate and analyze how each algorithms is efficient in comparison to each other we calculate the running time and approximation ratio with each approach as mentioned in section 3. As mentioned, we generate graphs G for $V \in [5, 50]$ where V is incremented by 5 and we repeat this 10 times per V. Therefore, we will then calculate the mean run-time and approximation ratio and standard deviation of the mean for each measurement x_i :

$$\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{1}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \boldsymbol{\mu})^2}{n-1}} \tag{2}$$

The standard devation of the mean is then:

$$\sigma_{\boldsymbol{\mu}} = \frac{\sigma}{\sqrt{n}} \tag{3}$$

Let us describe how we implemented our two observables $x_i \in \{$Running Time, Vertex Cover Size Ratio$\}$.

4.1 Running Time

To calculate the running time of each algorithm we utilize the `chrono` package in C++. This tool is used within the *void* functions we create for each of our algorithms threads (`cnfSatThread()`, `approxVC1Thread()` and `approxVC2Thread()` in `ece650-prj`). The timing happens with in the *Critical Sections* of each of these functions. The output of our run-time for each

V and algorithmic approach is sent to *output.csv* to then be analyzed. We created a python script to handle the output data and plot the mean run-time for each algorithm as a function of V. We find that the CNF-SAT-VC algorithm takes a much longer to run as V increases compared to the greedy algorithms of APPROX-VC-1 and APPROX-VC-2. It is worth noting that as $V \geq 20$ the CNF-SAT-VC efficiency exponentially decreases and our CNF formula used may not be the best approach for graphs of this size. This can be due to the number of clauses that our algorithm needs to iterate through which grows according to the following equation from [5]:

$$k + n\binom{k}{2} + k\binom{n}{2} + |E| \tag{4}$$

Here k is the size of the vertex cover, n is the number of vertices in our graph and $|E|$ is the size of our edges. For APPROX-VC-1 and APPROX-VC-2 algorithms the results that we have got are much faster than calculation of CNF-SAT-VC algorithm for vertex cover problem. After calculating our mean run-time and the error, we plotted our results as seen in Figure 6. As we can see, the time complexity of the CNF SAT algorithm grows exponentially, however, after $V \geq 20$ we notice this pleteaus, which is a result of the attempted improvement we made to our algorithm that we will discus in Appendix A. Let us now analyze the two approximation algorithms in terms of time-complexity using big O notation. These are both greedy algorithms that scale as follows:

- **APPROX-VC-1** This algorithm as described in section 2 contains a sorting algorithm for our vertex cover C, and a search and erasing mechanism of E. Thus, considering how these algorithms scale we have the following:

  $$\Theta\big(C \log C + |E|\big) \tag{5}$$

  Let us denote $n = |E|$ and $m = C$, then we have:

  $$\Theta\big(m \log m + n\big) \approx O\big(n\big) \text{if } n > m \tag{6}$$

- **APPROX-VC-2** This algorithm described in section 2 contains a sorting algorithm as well on our vertex cover C. Within the while loop, in the worst case, each edge E is considered once, and during each iteration, edges are removed. Therefore, the complexity of the while loop scales linearly in E. Thus we also have:

  $$\Theta\big(m \log m + n\big) \approx O\big(n\big) \text{if } n > m \tag{7}$$

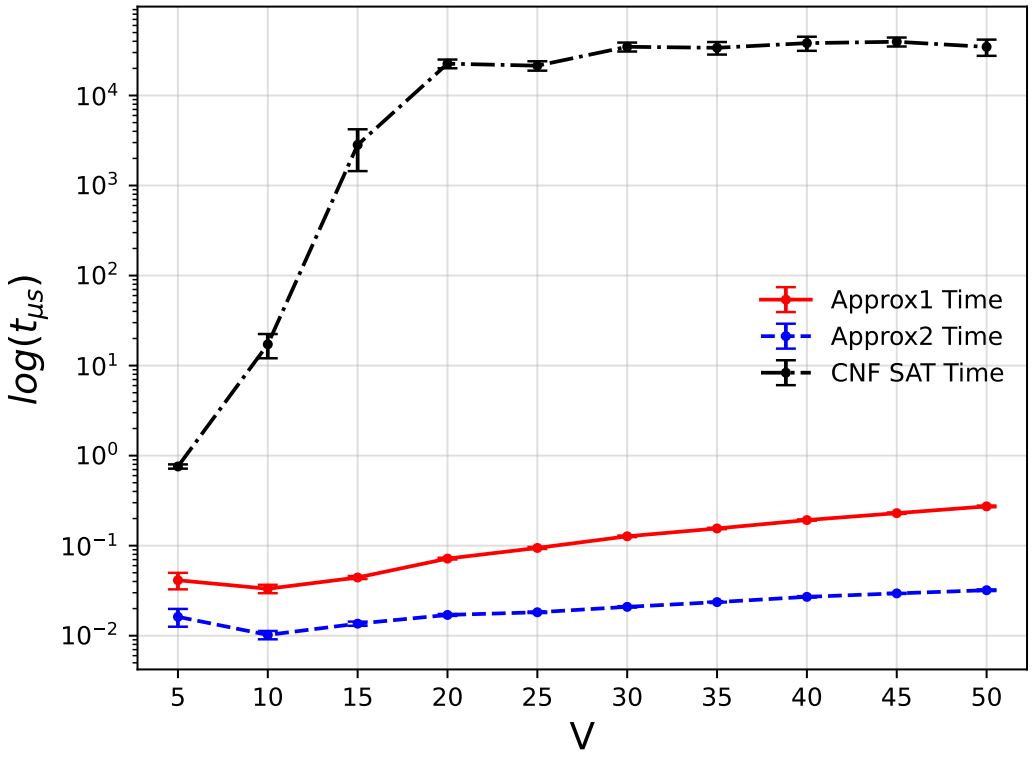

Figure 6: Runtime analysis of our three algorithms. Plotted is the logarithmic run-time in microseconds $log(t_{\mu s}$ as a function of the number of vertices in our graph V. Errorbars are calculated as the standard deviation of the mean.

note: When we execute both approximation algorithms we do make a copy of our input E initially since we are erasing from E. The operation of copying E also scales like $O\big(|E|\big)$.

4.2 Vertex Cover Size Ratio

In this analysis, we assume that the CNF-SAT-VC algorithm yields the minimum-sized vertex cover k. Therefore, a meaningful metric to assess the efficiency of the APPROX-VC-1 and APPROX-VC-2 algorithms is to calculate the ratio of their vertex cover sizes, denoted as s_1 and s_2, in relation to the CNF-SAT-VC size k. Thus this ratio has the following form:

$$s_i = \frac{\text{VC}_i}{\text{VC}_{\text{cnf-sat}}} \tag{8}$$

Here, VC_i is the vertex cover size of our i-th approximation and $\text{VC}_{cnf-sat}$ is the vertex cover size of our CNF-SAT-VC (i.e. this is k). A ratio closer to 1 suggests that the algorithm is more adept at approaching the minimum vertex cover size. This approximation ratio serves as an inverse measure of correctness, wherein a higher value indicates a less accurate vertex cover.

Figure 7 details our approximation ratios for the APPROX-1 and APPROX-2 algorithms as a function of the number of vertices used in the graph V. We notice that APPROX-VC-1 algorithm appears t perform better than APPROX-VC-2 in terms of a comparison of their vertex cover sizes with the optimal CNF-SAT-VC algorithm. The error for each measurement as a function of V does indicate however that their is some variability in the vertex cover sizes that our greedy approximation algorithms provide us since these are not guaranteed to be exact solutions of the vertex cover.

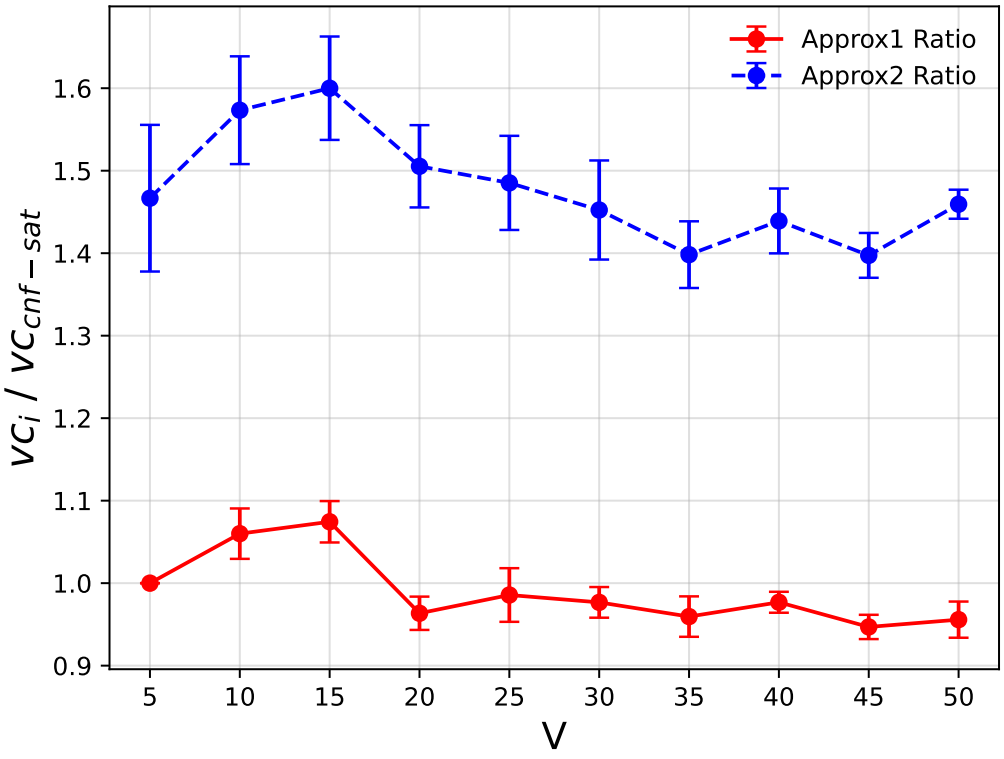

Figure 7: Approximation Ratio of APPROX-VC-1 and APPROX-VC-2 algorithm

5 Conclusion

While the CNF-SAT-VC algorithm yields optimal results for minimum vertex cover, its efficiency diminishes for larger graphs, $G = (V, E)$, generated by `graphGen`. Notably, the CNF-SAT-VC algorithm exhibits exponential scaling in running time as the graph size increases. In contrast, our approximation algorithms display linear scalability, making them more suitable for larger graphs. In the comparison between the two approximation algorithms, APPROX-VC-2 demonstrates marginal superiority in terms of running time performance compared to APPROX-VC-1, albeit similar time complexity scalings. Further analysis of the vertex cover sizes reveals that APPROX-VC-1 offers a better measure of correctness based on our calcu-

lated ratios.

In summary, our evaluation considered three distinct algorithms designed for computing the minimum vertex cover of a given graph. We presented findings based on their running time and approximation ratios. Notably, APPROX-VC-1 emerged as the preferred choice, offering the best balance across these metrics.

References

[1] Vertex Cover Example CS360 Approximation Algorithm lecture-29 2015: http://ycpcs.github.io/cs360-spring2015/lectures/lecture29.html

[2] Advanced Linux Programming, Mark Mitchell, Jeffrey Oldham, and Alex Samuel

[3] Introduction to Algorithms, Second Edition ,Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest and Clifford Stein

[4] Minisat Example: https://git.uwaterloo.ca/ece650-f23/minisat-example

[5] A4 encoding, A4 Assignment, Lecture notes, Threads example in Gitlab.

[6] MiniSat - A Minimalistic, Open-Source SAT Solver `http://minisat.se/`

A Appendix: Bonus

In order to improve the time-efficiency of the CNF-SAT-VC algorithm we made the following adjustment to our `cnfReduction()` function:

- First, we initialize our SAT solver variable as False and we start a new thread where we call `Minisat::solver.solve()`. In this second thread we will attempt to find a solution using our implementation of the CNF reduction from assignment 4. For each value of k we wait T seconds before interrupting the solver and moving to the next value of k using `BinarySearch`. The reason for this termination of the solver is that we notice when we lack a valid solution for as specific k on larger graphs (i.e. $V \geq 20$) the SAT solver will remain 'stuck' for a long time in the **for** loops we built for our clauses trying to find a valid solution. This will affect the rate at which we converge to the proper solution for a minimum vertex cover size k.

- The main variable we need to assess then in this situation is what is a suitable value T for which we can assure no solution will be found for a specific k. To do this the following test case was ran. We consider a graph in which the edges follow the below formula:

$$E = \{< i, i+1 >| \ 1 \leq i \leq V-1\} \tag{9}$$

  The reason for this, is we can assure that the minimum vertex cover will always be $k_{min} = \frac{V}{2}$ and so we will know the proper solution before-hand. We then perform the following two steps:

  - First for $k = \frac{V}{2}$ we see how long it will take our solver to find a solution for the maximum size graph we will be using (i.e. $V = 50$). This should give and indication to us how long it usually takes for a larger graph to solve a valid solution. When running this on 10 separate occasion our solver was able to provide us our solution in under 1 second each time for a minimum cover size of $k = V/2 = 25$. This indicates the solver is pretty efficient at getting a valid result if one exists[3].
  - Next we will add in a timeout variable that is some factor γ times a minimum of one second to find a valid solution for our

[3]This was also manually tested on some prior graphs from assignment 4 where we knew what the minimum value k was. Similarly, providing the solver this valid k we converged to a solution very quickly

maximum sized graph for a valid k. The agreed upon value was to atleast be large enough to not terminate anything that may be a valid solution and so a value of 10 was chosen. Thus we use a timeout of 10 seconds.

As seen from Figure 6 our run-time plateaus when we reach values of $V \geq 20$ as a result of this adjustment and we are able to obtain a valid solution for our vertex cover. This reduces the run-time from being exponential to linear in nature for $V \geq 20$.