ECE 650 - METHODS AND TOOLS FOR SOFTWARE ENGINEERING

UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Analytical Report on Three Vertex Cover Approaches

Project Members:
Seturaj Matroja (ID: 21064444)
Hritika Kalghatgi (ID: 21068377)

Date: December 7, 2023

1 Introduction

The identification of a minimum vertex cover within a graph is a well-established problem in computer science, characterized by its status as a paradigmatic example of an NP-hard optimization challenge. This report aims to present three distinct approaches for solving the minimum vertex cover problem, accompanied by a comprehensive exploration of their methodologies, interrelationships, and experimental intricacies, as delineated in sections 2 and 3. A meticulous analysis of each approach's efficiency will be conducted in section 4, where parameters such as running time and approximation ratios will be systematically elucidated and discussed. Section 5 will encapsulate the findings with a summary.

2 Methodology

1. Vertex Cover Algorithms

We mainly focus on three different types of algorithms which have been explained below in a short manner.

1. CNF-SAT - Polynomial Time Reduction:
In the CNF-SAT solution, we employ a polynomial time reduction from the VERTEX-COVER problem to CNF-SAT. A polynomial time reduction is an algorithm that operates in time polynomial in its input size. Here's an overview of the process:
Reduction Process: We transform the given instance of the VERTEX-COVER problem into an equivalent instance of the CNF-SAT problem, ensuring that a solution to the CNF-SAT problem corresponds to a solution for the original VERTEX-COVER problem.
Minisat SAT Solver: Following the reduction, we utilize a Minisat SAT solver to efficiently determine a satisfying assignment for the CNF-SAT problem, which, in turn, provides a solution to the original VERTEX-COVER problem.

2. APPROX-1 - Greedy Vertex Cover Approximation:
In the APPROX-1 solution, we employ a greedy approach to approximate the Minimum Vertex Cover by iteratively selecting vertices with the highest degree. The procedure unfolds as follows:
Step 1: Initial Vertex Selection: Identify a vertex with the highest degree in the graph.
Step 2: Vertex Cover Expansion: Add the selected vertex to the vertex cover and remove all edges incident on that vertex.
Step 3: Iterative Expansion: Repeat Steps 1 and 2 until no edges remain unprocessed, continuously selecting vertices with the highest degree.

The time complexity of APPROXVC-1 can be expressed as $O(V^2)$. In this algorithm, we iterate through each vertex value (V) to identify the vertex with the highest degree, resulting in V^2 comparisons. To facilitate this, we utilize an adjacency list implemented using a vector of vectors in C++, continuously checking the adjacency list for the highest degree. During each iteration, we traverse the entire list (column) to find the vertex with the highest degree and then loop over the particular row value of vertex v to locate its neighbors and delete the incident edges. In the worst case, this results in V^2 comparisons. Since the adjacency list uses a space of V^2, the space efficiency is also $O(V^2)$.

3. APPROX-2 - Edge-Based Vertex Cover Approximation:
The APPROX-2 solution utilizes an edge-centric approach to approximate the Minimum Vertex Cover. The process involves:
Step 1: Initial Edge Selection: Pick an arbitrary edge u,v from the graph.
Step 2: Vertex Cover Expansion: Add both vertices u and v to the vertex cover and discard all edges connected to u and v.
Step 3: Iterative Expansion: Repeat Steps 1 and 2 until no edges remain unprocessed, consistently selecting and adding vertices associated with the chosen edges. These three approaches present distinct strategies for finding approximate solutions to the Minimum Vertex Cover problem, each with its own characteristics and efficiency considerations.

In the case of APPROXVC-2, we employ an array and systematically select an edge, deleting all incident edges on the chosen (u, v) pair. Here, we make V comparisons in the worst case and iterate over the edges (E). Consequently, the running time can be expressed as O(V * E), which remains polynomial relative to the size of the input.

The space complexity remains consistent with that of the APPROXVC-1 algorithm. Notably, the APPROXVC-2 algorithm demonstrates enhanced efficiency, as the removal of edges incident on two different vertices with each edge selection accelerates the graph reduction compared to the APPROXVC-1 algorithm. However, it is important to acknowledge that this accelerated approach may not consistently yield an optimal solution, a topic we will delve into in the next section.

2. Multithreading

Within the realm of computer architecture, the concept of multithreading denotes the inherent capability of a central processing unit (CPU) or a solitary core embedded within a multi-core processor to simultaneously manage multiple threads of execution. This intrinsic capability is intricately coordinated by the underlying operating system, orchestrating a seamless and concurrent execution of diverse threads. This report outlines the strategic implementation of multithreading within our project, specifically addressing two pivotal aspects:

1. Input/Output Thread:
A dedicated thread has been employed to manage input and output operations throughout the execution of the program. This thread is designed to persist until the entirety of the output has been successfully generated and printed.

2. Algorithmic Threads:
To optimize the processing of the CNF (Conjunctive Normal Form) algorithm, as well as the VertexCover1 and VertexCover2 algorithms, individual threads have been instantiated. This approach ensures concurrent execution of these algorithms, thereby enhancing the overall efficiency of the system.

3. Timeout for CNF Thread:
Notably, the CNF thread has been configured with a timeout constraint of 60 seconds. This deliberate limitation is in response to the observed computational demands, particularly when processing instances involving 15 or more vertices. Empirical results indicate that the CNF algorithm necessitates substantial computational time, manifesting as approximately 6 minutes for 20 vertices and an extended duration of 45 minutes for 25 vertices.

The incorporation of multithreading in the aforementioned capacities is instrumental in optimizing the program's execution, ensuring efficient handling of input/output operations, and managing the computational complexity associated with intricate algorithms.

3. Optimization

We employed binary search optimization to determine the minimum vertex cover value (k) in our algorithm. This application significantly reduced the search time for CNF-SAT by eliminating numerous false minimum possibilities of vertex cover size. Initially, the code performed effectively for vertices up to 15, producing outputs in under a minute. However, as we extended its application to higher vertices (e.g., 20, 25), we observed a substantial increase in execution time. For instance, the algorithm ran for approximately 6 minutes for 20 vertices and 45 minutes for 25 vertices, prompting us to terminate execution for higher vertex values due to excessive runtimes.
The primary cause of this performance degradation lies in the third constraint, which generates an extensive number of clauses. This constraint operates in exponential time, and the overall time efficiency of the CNF-SAT algorithm is inherently exponential. While the Vertex Cover problem is NP-complete and cannot be solved in polynomial time, we sought to enhance the efficiency of CNF-SAT by devising a more intelligent and efficient encoding strategy.
To address this issue, we introduced modifications to the CNF-SAT algorithm to optimize

its performance. The Minisat Solver, in its attempt to find a solution for all possibilities, explores permutations of the vertex cover size. We recognized that the correctness of certain permutations could be determined by the correctness of others, allowing us to streamline the search process.

Addressing this issue, consider this situation; in the execution process, the Minisat Solver systematically explores various permutations when attempting to find a solution for the minimum size Vertex Cover. For instance, if the initially identified optimal vertex cover is represented by the sequence 2, 5, 7, 9, 11, the solver subsequently evaluates all possible permutations, such as 5, 7, 2, 9, 11. Crucially, this exhaustive exploration accounts for the inherent symmetry in the Vertex Cover problem; thus, permutations like 7, 9, 5, 2, 11 are equally valid solutions if the initial sequence is correct. Conversely, if the initial sequence proves to be an incorrect solution, the same holds true for all its permutations, underscoring the symmetric nature of potential vertex cover outputs. In essence, this observation establishes that all permutations of the vertex identifiers within the identified solution set are either collectively correct or collectively incorrect for a given Vertex Cover size.

The modified code introduces an additional loop to check specific conditions within the third clause. Specifically, for each vertex A(i)(j), if A(2)(j) is correct, then all other A(i)(j) should be false for i!=2 and j < k (minimum vertex cover size we are trying to find). This adjustment helps avoid exploring permutations that do not contribute to the correct solution, effectively reducing the search space.

Below is the code snippet originally:

```
for (int i = 0; i < k; ++i) {
    for (int p = 0; p < n - 1; ++p) {
        for (int q = p + 1; q < n; ++q) {
            solver->addClause(~lits[p][i], ~lits[q][i]);
        }
    }
}
```

Now, here is the modified code:

```
for (int i = 0; i < k; ++i) {
    for (int p = 0; p < n ; ++p) {
        for(int j=i; j<k; j++) {
            for (int q = p + 1; q < n; ++q) {
                solver->addClause(~lits[p][i], ~lits[q][j]);
            }
        }
    }
}
```

With these adjustments, we achieved a significant improvement in the algorithm's efficiency. The modified code now delivers outputs for vertex values as high as 50 in a matter of minutes, as demonstrated by the plotted graph.

3 Experimental Details

Our workflow commences with the utilization of the graphGen tool on the eceubuntu platform, generating graphs characterized by a consistent number of edges for a given quantity of vertices. Subsequently, these generated graphs serve as inputs processed within the Input/Output (I/O) thread. The I/O thread is responsible for extracting pertinent information from the generated graphs and storing the vertices and edges in two distinct vectors.

Each of the algorithmic threads receives this graph data as input, wherein the parameters provided include an integer representing the number of vertices (V) and a vector storing the edges of each graph. This structured input facilitates the independent execution of three distinct algorithms—CNF, VertexCover1, and VertexCover2—in concurrent threads.

Upon the completion of computation by each approach within their respective threads, the program outputs the minimum vertex cover. This output is encapsulated in a vector, where vertices are sorted in ascending order. Notably, to circumvent potential conflicts arising from modifications, we have implemented three separate vectors to store the output vertex covers corresponding to each algorithmic approach. This design choice ensures the integrity of the output data and enables a comprehensive analysis of the results obtained from each algorithm.

Result Analysis

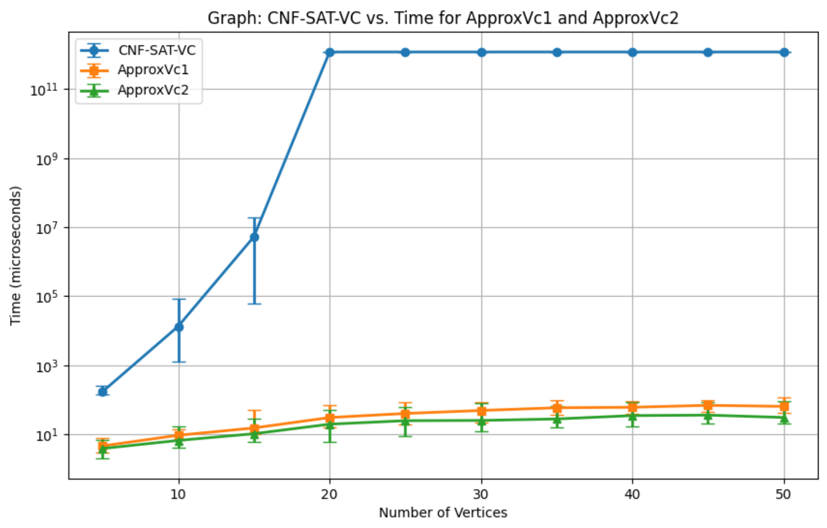

Figure 1: The three algorithms exhibit the above running time when we take 50 vertices.

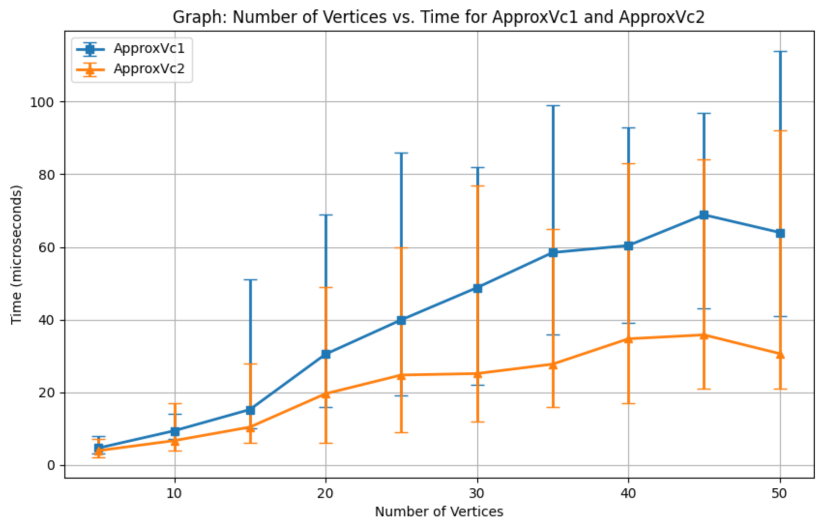

Figure 2: APPROXVC1 and APPROXVC2 exhibit the above running time based on vertices up to 50.

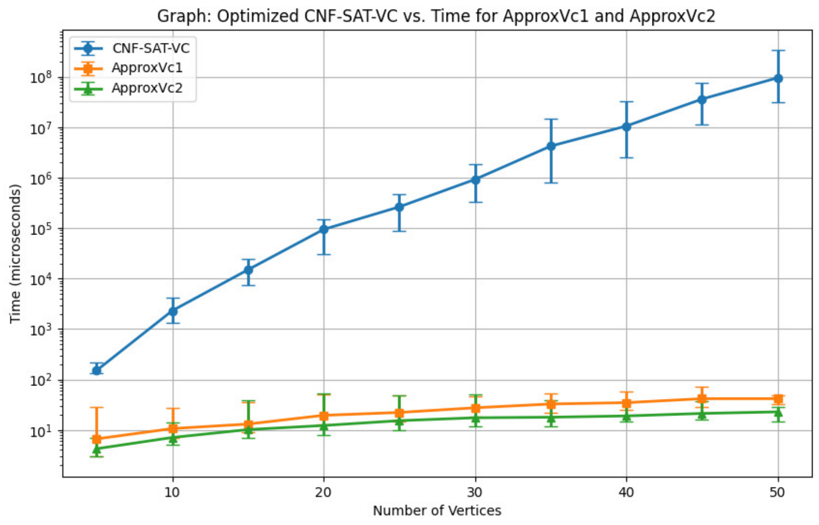

Figure 3: The three algorithms exhibit the above running time with an optimized CNF-SAT algorithm.

To assess the efficacy of each algorithmic approach, we conducted a comprehensive analysis encompassing both running time and approximation ratio. This evaluation involved the generation of datasets consisting of 10 graphs for each distinct value of vertices (V), ranging from 5 to 50 in increments of 5. For each graph, we executed the program 10 times, accumulating a dataset of 100 runs for every unique value of V. Subsequently, we computed the mean and standard deviation across these 100 runs, thereby providing a robust statistical foundation for evaluating the performance of each algorithm across a spectrum of vertex values.

(a) Running Time

We meticulously documented the individual running times of each algorithm thread, subsequently deriving their respective averages and standard variances. Recognizing the significant disparity in execution times, particularly evident in the case of CNF-SAT, which demands a substantially longer duration for computation compared to the first two approaches, we opted for a differentiated visualization strategy. But Figure 3 has our optimized CNF-SAT algorithm which shows a better run time.
Reaching up to 45 minutes for 25 vertices, we can see from our analysis in Figure 1 that the time taken by CNFSAT becomes exponential above 15 vertices.

Furthermore, for a more detailed exploration of the APPROXVC-1 and APPROXVC-2 algorithms, we curated an additional graph (Figure 2) presenting results for vertices spanning the entire spectrum, from 5 to 50. This approach contributes to a comprehensive understanding of algorithmic performance across varying input sizes.

In Figure 2, the graphical representation illustrates a growth in both APPROXVC-1 and APPROXVC-2 algorithms with an increase in the number of vertices. A comparative analysis between these two algorithms reveals that APPROX-1 incurs a higher running time compared to APPROX-2. That is, APPROXVC-1 methodology involves always picking the highest degree vertex and removing all the edges incident on that vertex. Keep doing this until all edges related to the graph are removed. APPROXVC-2 methodology involves picking an edge(u,v) and removing all the edges incident on the u,v in the graph. We keep picking edges until all edges are removed from the graph. This discrepancy arises from the programmatic structure, wherein APPROXVC-1 necessitates additional sequential loops and computational steps for selecting the vertex with the highest degree. In contrast, APPROXVC-2 adopts a more efficient strategy by randomly selecting edges, resulting in a comparatively lower computational cost.
In the context of the old CNF-SAT algorithm, the running time exhibits a substantial increase with a higher number of vertices and edges in the graph. Experimental observations indicate that attempting to execute the program with graphs containing 15 vertices or more incurs an exceptionally prolonged duration, surpassing 45 minutes. But when we look at Figure 4, which has our optimized CNFSAT algorithm, there is an improvement in the running time.

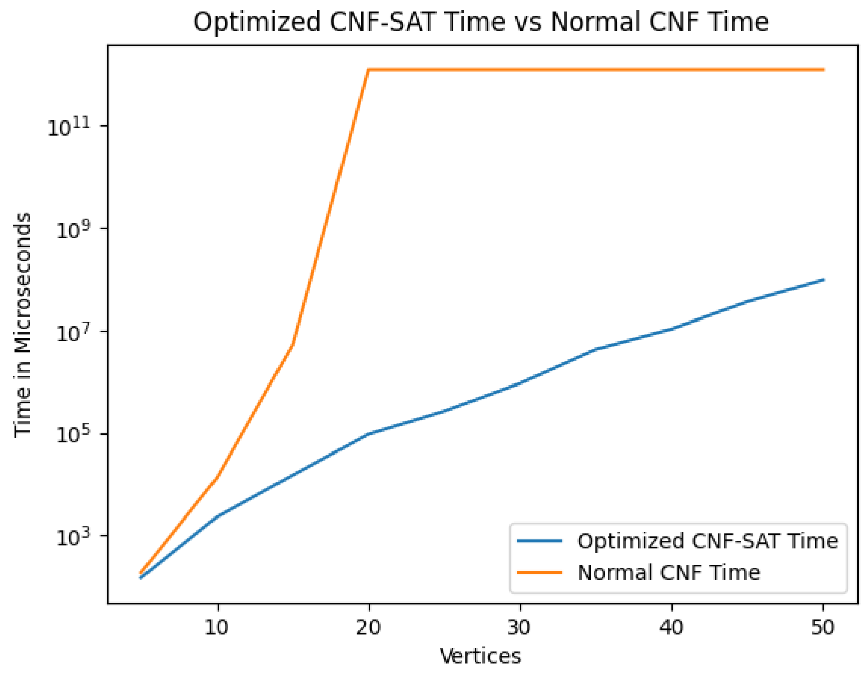

Figure 4: The graph shows the improvement we achieved.

Figure 4, which exclusively focuses on the old and improvised CNF algorithm, provides a discerning analysis, showcasing an exponential rise in running time as the number of vertices increases and an improvement as well. This behavior is inherently attributed to the NP-complete nature of the problem, signifying that solutions cannot be obtained within polynomial time constraints. The pronounced increase in running time serves as a testament to the computational complexity inherent in solving NP-complete problems.

(b) Approximation Ratio

We conducted an assessment of the approximation ratios for APPROX-1 and APPROX-2, defined as the ratio between the size of the computed vertex cover and the size of an optimal (minimum-size) vertex cover. We can confidently assert that the vertex cover obtained from CNF-SAT is consistently an optimal solution. This is because CNF-SAT systematically examines all possible solutions and returns one. Consequently, we consider the vertex cover generated by our optimized CNF-SAT as the benchmark for an optimal solution and proceed to compare its size with the vertex covers produced by the APPROXVC-1 and APPROXVC-2 algorithms.

$$\text{Approximation Ratio} = \frac{\text{Size of Vertex Cover (Approximation Algorithm)}}{\text{Size of Vertex Cover (CNF-SAT)}}$$

Figure 5 visualizes these approximation ratios, where the x-axis corresponds to the number of vertices (V), and the y-axis depicts the approximation ratio.

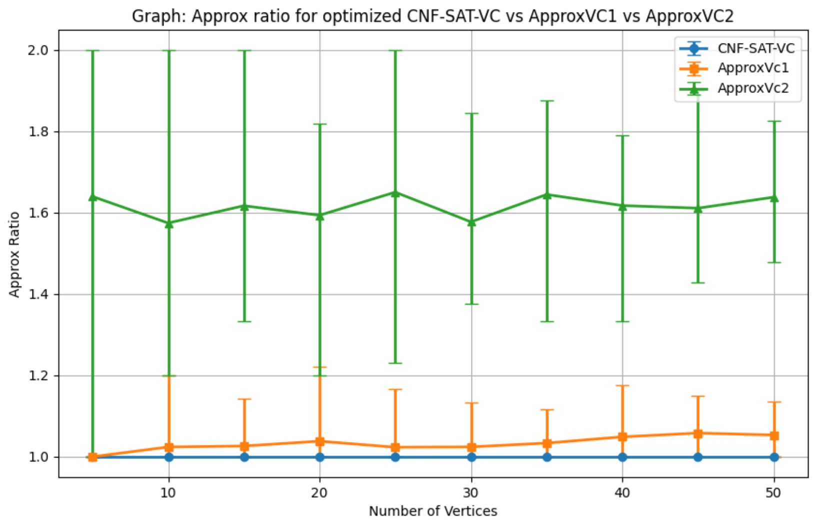

Figure 5: We compare the approximation ratio of all our algorithms with the optimized CNF-SAT.

The graphical representation in Figure 5 distinctly illustrates that APPROXVC-1 yields results closely aligned (essentially 1:1) with the optimal solution, reflecting its effectiveness in approximating minimum-size vertex covers. Conversely, APPROX-2 tends

to produce larger vertex covers, indicative of its inability to consistently guarantee the accuracy of minimum-size results.
However, despite being the quickest algorithm in solving our vertex cover, APPROXVC-2 exhibits suboptimal performance in delivering an optimal solution. In the worst-case scenario, the solution provided by APPROXVC-2 is nearly twice the size of the optimal solution, which is undesirable in terms of correctness. Additionally, the error bars indicate a considerable range in the variation of the ratio for graphs with the same number of vertices. Notably, in specific instances, APPROXVC-2 demonstrates spikes. This phenomenon can be attributed to the graph's high connectivity, where the removal of an edge results in the elimination of a significant portion of edges, bringing the solution closer to the optimal outcome.
In summary, the CNF-SAT algorithm exhibits superior efficiency in identifying minimum vertex covers for smaller datasets. However, its running time significantly escalates as the number of vertices and edges increases, rendering it less practical for larger datasets. APPROX-2, while boasting the shortest running time, exhibits an inconsistent approximation ratio, implying a diminished accuracy in generating minimum vertex covers. On the other hand, APPROX-1 combines a relatively short running time with an approximation ratio consistently close to optimal results, positioning it as a more efficient solution as data scales.

4 Conclusion

In summary, it is evident that the running time of the CNF-SAT solver exhibits an exponential correlation with the input size. While both approxvc1 and approxvc2 boast shorter execution times compared to the optimized cnf-sat-vc, it is crucial to acknowledge that optimized cnf-sat-vc consistently provides the optimal solution for the minimum vertex cover, ensuring the smallest vertex cover size.
Upon comparing these results with the two approximation algorithms, it becomes apparent that APP1 VC manages to yield an almost identical optimal solution within a polynomial time frame. In such scenarios, preference leans towards the approximation algorithms over CNF-SAT. However, it's noteworthy that the other approximation algorithm, APP2 VC, despite running in polynomial time and faster than APP1 VC, doesn't consistently deliver the desired optimal vertex cover solution.
Therefore, in an anticipated scenario, selecting APP1 VC emerges as a favorable choice for solving the vertex cover problem, primarily due to its superior running time. However, if the need for a highly efficient and correct output prevails, the optimized CNF-SAT VC algorithm remains the preferred option.

References

1. a4-encoding.pdf

2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.