Efficiently Solving Vertex Cover: An In-Depth Analysis of Different Approaches to VC Problem

Tonghe Bai
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
t4bai@uwaterloo.ca

Leo Zhang
Department of Systems Design Engineering
University of Waterloo
Waterloo, Canada
l536zhang@uwaterloo.ca

***Abstract*—This report presents an in-depth analysis of various approaches to solving the minimum vertex cover problem in graphs. The software, developed as part of the ECE650 course project, encompasses existing CNF-SAT-VC methodology, augmented with two additional algorithms APPROX-VC-1 and APPROX-VC-2, and multithreading capabilities. This report provides insights into the implementation of each algorithm, the multithreading strategy, and the program's input/output structure. A comprehensive quantitative analysis is conducted, evaluating the running time and approximation ratios for different input scenarios considering varying graph sizes. The findings are presented through plots for the efficiency of each approach and the impact of graph size on performance. The report concludes with discussions on observed patterns and outliers, and provides an understanding of the software's capabilities and limitations.**

***Index Terms*—CNF, SAT, VC, approximation**

I. Introduction

The central objective of this project is the refinement and evaluation of software dedicated to addressing the minimum vertex cover problem. It involves augmenting existing code from Assignment 4, introducing multithreading capabilities, and implementing novel algorithms to enhance the problem-solving process.

A. Problem Statement

The minimum vertex cover problem involved identifying the smallest set of vertices within a graph, ensuring that each edge has at least one endpoint in this selected set. The project's primary focus lies in employing diverse approaches to solve this problem, with an emphasis on overcoming challenges such as the incorporation of multithreading, the introduction of innovative algorithms, and the analysis of performance across varying graph sizes.

B. Objectives

- **Approximation Algorithms:** Implement two additional algorithms namely APPROX-VC-1 and APPROX-VC-2 to tackle the minimum vertex cover problem alongside the existing CNF-SAT-VC approach.
- **Multithreading:** Incorporate multithreading by utilizing at least four threads for the I/O operations as well as the vertex cover solving methods.
- **Performance Evaluation and Quantitative Analysis:** Perform a detailed quantitative analysis of the software's efficiency with graphs, evaluating running time and approximation ratios for various inputs with varying numbers of vertices.

II. Implementation

A. CNF-SAT-VC

The code for CNF-SAT-VC in Assignment 4 focused on solving the minimum vertex cover problem using the MiniSat solver. The developed code utilized various rules to formulate constraints for the SAT problem, in order to determine the minimum-sized vertex cover for a given graph. The following key components were present in the a4 code:

1) **Solver Initialization:** The code initialized the MiniSat solver and set up necessary structures to represent the graph.
2) **Rule Definitions:** Several rules were defined to establish constraints for the SAT problem. These rules included:
   - `ruleAtLeastOneVertex`: Ensuring that at least one vertex is chosen from the vertex cover for each position.
   - `ruleNoRepeatedVertex`: Prohibiting the selection of the same vertex in multiple positions of the vertex cover.
   - `ruleNoMoreThanOneVertexPerPosition`: Ensuring that no more than one vertex is selected for each position in the vertex cover.
   - `ruleEdgeIncidence`: Guaranteeing that every edge in the graph is incident to at least one vertex in the vertex cover.
3) **Solver Execution:** The solver iterated over different values of k (size of the vertex cover) until a solution was found or all possibilities were exhausted. For each k, the rules were applied, and the SAT problem was solved. If a solution was found, the corresponding vertex cover was extracted and output.
4) **Input Handling:** The code read input commands from standard input, including 'V' (vertex count) and 'E' (edge specification). The 'V' command set the number

of vertices in the graph, while the 'E' command specified the edges between vertices.

5) **Input Error Handling:** The code included error-checking mechanisms to ensure the validity of input commands. It handled cases where the number of nodes was less than or equal to 1 and discarded invalid or repeated 'E' commands.

For the current project, the code from a4 serves as the foundation for the CNF-SAT-VC approach. In the augmentation process, the code was modified to accommodate multithreading and integrate two additional algorithms: APPROX-VC-1 and APPROX-VC-2 as shown below.

B. APPROX-VC-1

The APPROX-VC-1 algorithm is implemented based on the following steps:

1) **Initialization:** The algorithm begins with the initialization of the set of remaining edges, denoted as `remainingEdges`, and an empty vector to store the vertices in the vertex cover (`vertexCover`).
2) **Main Loop:** The main loop continues until no edges remain in the graph. Inside the loop, the degree of each node is counted by creating an unordered map (`degreeMap`) where the keys are node IDs, and the values are their respective degrees. The node with the maximum degree (`maxDegreeNode`) is identified based on the degree map.
3) **Adding Nodes to Vertex Cover:** The node with the maximum degree is added to the vertex cover (`vertexCover`). All edges incident to this node are identified and stored in `edgesToRemove`.
4) **Removing Incident Edges:** The identified edges (`edgesToRemove`) are removed from the set of remaining edges (`remainingEdges`).
5) **Sorting and Output:** The vertex cover is sorted in ascending order. The final result is output in a standardized format, showing the nodes in the computed vertex cover.
6) **Promise and Lock Mechanism:** The algorithm uses a promise (`approxVC1Promise`) to signal the completion of the computation. A lock (`std::lock_guard`) is employed to ensure thread safety during the execution of this algorithm.

The structure above implements the described APPROX-VC-1 algorithm. The output includes the nodes in the vertex cover, presented in sorted order. The promise mechanism (`approxVC1Promise`) is used to signal the completion of the computation in a multithreaded environment.

C. APPROX-VC-2

The implementation of the APPROX-VC-2 algorithm involves the following steps:

1) **Initialization:** The algorithm begins with the initialization of an unordered set (`vertexCover`) to store the vertices in the vertex cover and a set of remaining edges (`remainingEdges`).
2) **Main Loop:** The main loop continues until no edges remain in the graph. Inside the loop, the algorithm picks an arbitrary edge $< u, v >$ from the set of remaining edges.
3) **Adding Nodes to Vertex Cover:** Both u and v are added to the vertex cover (`vertexCover`).
4) **Removing Incident Edges:** All edges attached to u and v are identified and stored in `edgesToRemove`.
5) **Removing Edges from Remaining Edges:** The identified edges (`edgesToRemove`) are removed from the set of remaining edges (`remainingEdges`).
6) **Sorting and Output:** The vertex cover is sorted in ascending order, and the final result is output in a standardized format, showing the nodes in the computed vertex cover.
7) **Promise and Lock Mechanism:** The algorithm uses a lock (`std::lock_guard`) to ensure thread safety during the execution of this algorithm.

The structure above implements the described APPROX-VC-2 algorithm. The output includes the nodes in the vertex cover, presented in sorted order.

D. Multi-threading

The multithreading implementation in the program involves the allocation of separate threads for various tasks, enhancing overall efficiency. At least four threads are utilized, including one for I/O operations and one for each vertex cover solving approach. The presented structure below demonstrates the multithreading implementation, enhancing the program's efficiency by concurrently executing I/O and vertex cover solving tasks.

1) **I/O Thread:** A dedicated thread (`ioThread`) is created for handling input/output operations. It reads the graph information either from the `graphGen` tool or standard input. This thread must be joined until the algorithm threads are created to ensure completed graph specifications.
2) **Algorithm Threads:** Three additional threads are created for each vertex cover solving approach: `cnfSatVCThread` for the CNF-SAT-VC methodology; `approxVC1Thread` for the APPROX-VC-1 algorithm; `approxVC2Thread` for the APPROX-VC-2 algorithm.
3) **Thread Creation:** Threads are created and launched simultaneously to execute their respective tasks concurrently.
4) **Synchronization with Promises and Futures:** Promises and futures are used for synchronization between threads. Promises are reset before each iteration, and futures are waited upon to ensure that specific tasks are printed before the others in the specified order.
5) **Thread Joining:** After launching the threads, the main thread waits for the completion of the CNF-SAT-VC and APPROX-VC-1 threads using `cnfSatVCFuture` and `approxVC1Future`.

6) **Thread Termination:** All threads, including I/O and algorithm threads, are joined to the main thread to ensure proper termination and synchronization.
7) **Delay:** A delay of one second (`sleep_for`) is introduced at the end of each iteration to manage thread synchronization and avoid potential race conditions.

E. Timer

The `std::chrono` timer mechanism is used to measure thread execution time, the exact code is as follows:

```
auto start_time = high_resolution_clock::now()
    ↪ ;
// thread execution here
auto end_time = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(
    ↪ end_time - start_time);
```

Listing 1. Timer mechanism code

This is used because it provides the highest possible resolution with the smallest tick period. We set a start_time using the above function at the beginning of each thread's critical section, and an end_time immediately after the vertex cover is calculated at each function. In the end, we calculate duration. We use these 3 lines to wrap around the vertex cover calculation. This should in theory produce the most accurate result.

III. Program I/O

A. Input

The input for the program is sourced from the output of */home/agurfink/ece650/graphGen/graphGen* on *eceubuntu*. This particular program consistently generates graphs with a fixed number of edges for a given number of vertices, although the specific edges in each run may differ. The input format of a single graph is structured as follows:

- `V 5`
- `E {<3,2>,<3,1>,<3,4>,<2,5>,<5,4>}`

The above example represents a graph with 5 vertices and the corresponding edges specified within the set notation. To be more specific, it is formatted as:

- Letter "V"; a space char; a positive integer.
- Letter "E"; a space char; an open curly bracket; a sequence of edge representations separated by commas; a close curly bracket.
- Each edge is represented by an open angle bracket; a positive integer as the starting vertex; a comma separation; another positive integer as the ending vertex; a close angle bracket.

B. Output

The program is expected to produce output that denotes the vertex cover computed by each approach. The results should be presented in sorted order. Following the example input above, the expected output is as follows:

- `CNF-SAT-VC: 3,5`
- `APPROX-VC-1: 3,5`
- `APPROX-VC-2: 1,3,4,5`

This output format helps in assessing and comparing the results obtained by different algorithms for the minimum vertex cover problem. Each line in the output follows a specific structure:

- The algorithm name, followed by a colon.
- A space char.
- The computed vertex cover, presented as a sorted sequence of vertices, separated by commas.

IV. Observations and Initial Analysis

In this section, we conduct a quantitative analysis of the software's performance across diverse input scenarios. The evaluation encompasses running time analysis for each approach, with a focus on assessing efficiency based on the number of vertices ($|V|$). "Efficient" is characterized in one of two ways: (1) running time, and (2) approximation ratio. We characterize the approximation ratio as the ratio of the size of the computed vertex cover to the size of an optimal (minimum-sized) vertex cover.

More specifically, the results, plots, and graphs showing the running time and approximation ratios for various values of $|V|$ will be demonstrated for each approach.

A. Running Time Analysis

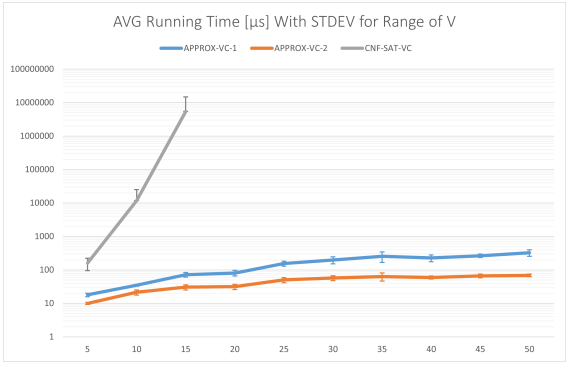

Fig. 1. Running Time (log scale) vs Range of V (normal scale)

In the above plot, we have included the three plots in the logarithmic scale for the average running time of APPROX-VC-1, APPROX-VC-2, and CNF-SAT-VC solvers. The runtime of the SAT solver is increasing exponentially with the increase in the number of vertices. This is because SAT is an NP-complete problem, and finding a solution is expected to take exponential time in the worst case, as no known polynomial-time algorithm exists for solving arbitrary instances of SAT. For a smaller number of vertices, there are fewer literals. For a larger amount of vertices, it would take longer to solve as there are more literals. The average running time for over 20 vertices is not shown for CNF-SAT-VC, as it takes too long to solve. The standard deviation is also the highest around V=15. This is because the exponential increase in time will yield varied results, therefore producing the highest standard deviation.

In the plot, it is also shown the plots for APPROX-VC-1 and APPROX-VC-2. APPROX-VC-2 has a faster runtime than APPROX-VC-1. This is because APPROX-VC-1 adds the minimum cover by picking a node of the highest degree, deleting its incident edges, and repeating until no edges remain. This means that vertexes in the vertex cover can be visited more than once in the linear search loop. APPROX-VC-2, append vertex list by picking two vertices from an edge, throw away all edges incident to the two, and repeat until no vertex remains. This is more efficient as no vertice in the vertex cover is being visited more than once in the linear search loop.

B. Approximation Ratio Analysis

The approximation ratios for APPROX-VC-1 and APPROX-VC-2 are shown in the following image. The approximation ratio for APPROX-VC-1 is close to 1 throughout. This is mostly because it a much more accurate alogorithm than APPROX-VC-2. It will always choose the highest degree vertex globally. This will often yield the minimum cover, as the highest degree vertex always cover the most amount of edges, and it would be effective for graph generated in general. APPROX-VC-2, however, picks an edge at random. It would be less accurate than APPROX-VC1 because it does not follow any particular algorithm, rather than just selects a random edge and add its vertexes. A detailed discussion of approximation ratios is included in section 5.

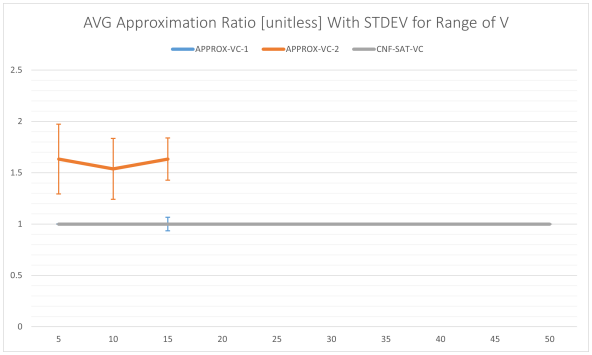

Fig. 2. Approximation Ratio vs Range of V, note that the APPROX-VC-1 plot is not quite visible as its overlapped with CNF-SAT-VC

There is no general trend for the approximation ratio in APPROX-VC-2. This is because the input vertex and edges are generated randomly. Therefore, the approximation ratios are also random. The standard deviation is decreasing, because graphGen has a constant V/E count ratio=1.5 (rounded down). This means for higher V count, the graph has less connectivity, which means the standard deviation would be smaller for higher V.

For APPROX-VC-1, the standard deviation is higher at V=15, this is likely caused by the increased vertex and edge size. Even though the graph is less connected, it would still mean an increase in vertex cover, based on vertice degrees. This means more varied results for standard deviation.

V. Further In-Depth Discussions

A. Running Time Discussion

The time complexity of APPROX-VC-1 can be analyzed based on the observed linear trend in Figure 3 below, note that now the y-scale is reverted to normal scale (i.e. not logarithmic) for a better comparison and visualization between APPROX-VC-1 and APPROX-VC-2.

As the number of vertices ($|V|$) increases, the running time of VC1 shows a linear growth pattern. This suggests that the algorithm has a time complexity of $O(|V|)$ or $O(|V| + |E|)$, where $|E|$ is the number of edges in the graph.

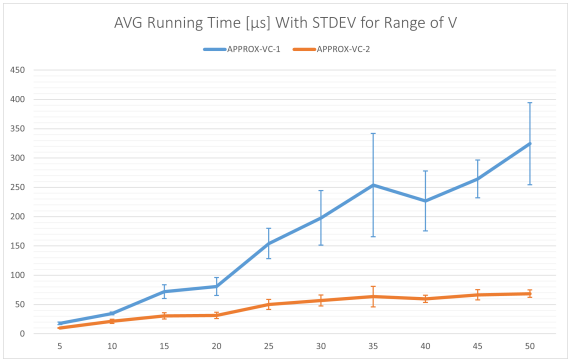

Fig. 3. Running Time (normal scale) vs Range of V (normal scale) For APPROX-VC-1 and APPROX-VC-2

The linear time complexity indicates that the algorithm's performance is proportional to the size of the input graph. Each additional vertex or edge contributes to a linear increase in the running time. This efficiency is advantageous for handling larger graphs without a significant increase in computation time.

It is worth noting that the observed linear trend might be influenced by the characteristics of the input graphs used for testing. Further analysis with a diverse set of graphs could provide additional insights into the algorithm's scalability, at least beyond what CNF-SAT-VC can handle for the current encoding.

The comparison with CNF-SAT-VC and VC2 suggests that VC1 achieves a balance between the faster but less accurate VC2 and the slower but more accurate CNF-SAT-VC. VC1's linear time complexity positions it as a viable option for scenarios where a reasonable approximation is acceptable within a limited time frame.

The time complexity of APPROX-VC-2 is $O(V + E)$ because, in the worst-case scenario, the algorithm needs to traverse all the edges and vertices to add the endpoints to the vertex cover. Each addition of two vertices to the vertex cover covers one edge at a time. Therefore, the increase of total vertices and edges increases runtime, as shown in Figure 1. The time complexity for CNF-SAT is unknown. However, it is known that its time complexity will increase exponentially

with the increases of literals in clauses and the increase in k. It should have a higher time complexity than APRROX-VC1 and APPROX-VC-2. It involves an exhaustive search for true assignments, from the smallest size of k to the actual minimum. As the number of literals in the clauses increases, the search space grows exponentially, leading to a higher time complexity.

B. Approximation Ratio Discussion

The approximation ratio for APPROX-VC-1 consistently remains around 1, indicating that the algorithm almost always provides solutions close to the optimal vertex cover size determined by CNF-SAT-VC. The comparison between the computed vertex cover (C) by APPROX-VC-1 and the optimal vertex cover (C^*) illustrates that $|C|$ is nearly equal to $|C^*|$ for various input scenarios.

The approximation ratio of APPROX-VC-1, a greedy algorithm for the vertex cover problem, is bounded between 1 and $\log n$. At each step, the algorithm adds vertices to the vertex cover, creating a cover of size roughly $n + \frac{n}{2} + \frac{n}{3} + \ldots$, which is of size $\Omega(n \log n)$ due to the harmonic series. Therefore, the worst-case approximation ratio can be expressed as:

$$\text{Approximation Ratio} = \frac{n}{\log n} = \log n$$

The tight approximation ratio observed in APPROX-VC-1 results from its algorithmic design. APPROX-VC-1 develops a strategy that tends to select vertices contributing to the minimum vertex cover by prioritizing nodes with high degrees. By iteratively choosing vertices with the maximum degree and removing their incident edges, APPROX-VC-1 will most likely build up a vertex cover that reasonably approximates the optimal solution and prevents unnecessary expansion of the vertex cover.

The consistently low approximation ratio of around 1 for APPROX-VC-1 reflects its effectiveness in providing solutions that closely align with the optimal vertex cover. This characteristic coupled with VC1's linear running time, makes it a valuable algorithm for scenarios where a quick and reasonably accurate solution is desired.

For APPROX-VC-2, it is evident from our graph in Figure 2 that APPROX-VC-2 has a tight upper bound of 2. This can be expressed in the following formula:

$$|C| \leq 2|C^*|$$

C denotes the size of the computed vertex cover, and C^* denotes the size of the optimal vertex cover. Let A denote the set of edges that the algorithm picked. $C*$ should be greater or equal to the count of A. This is because all edges in A share an endpoint, and no two edges in A are covered by the same vertex from $C*$. For every vertex in $C*$, it covers at most one edge in A. This gives: $|C| \geq |A|$.

For C, each endpoint in A should all be included. This is because each edge picked should have vertexes not included in C. This gives: $|C| = 2|A|$.

Combining the two equations above gives the first equation of $|C| \leq 2|C^*|$.

As shown in Figure 4 below, in contrast to APPROX-VC-1's consistent approximation ratio **around 1** (between 1 and log n) and APPROX-VC-2's inconsistent approximation ratio **between 1 and 2**, CNF-SAT-VC achieves an exact minimum vertex cover. CNF-SAT-VC explores the solution space exhaustively, ensuring an optimal result. However, this exhaustive search comes at the cost of increased running time, as observed in the running time analysis.

Fig. 4. Bounding Conditions for Approximation Ratios

While APPROX-VC-1 sacrifices optimality for speed, its approximation ratio of around 1 makes it a very compelling choice in scenarios where a balance between accuracy and efficiency is required. The algorithm's predictable performance and approximation ratio results makes APPROX-VC-1 as a practical solution for real-world applications where obtaining an exact optimal solution is less critical than achieving timely results.

In summary, APPROX-VC-1's approximation ratio analysis affirms its suitability for applications where a near-optimal solution is acceptable within a time-constrained environment.

VI. Conclusion

In conclusion, the comprehensive analysis and discussions on running time and approximation ratios provide important information into the performance and characteristics of the three vertex cover algorithms: CNF-SAT-VC, APPROX-VC-1, and APPROX-VC-2.

A. Algorithmic Trade-offs

Each algorithm presents distinct trade-offs. CNF-SAT-VC guarantees an exact minimum vertex cover but at the expense of exponential running time, especially as the number of vertices and edges increases. On the other hand, APPROX-VC-1 sacrifices optimality for speed, offering a linear time complexity and consistently providing solutions close to the optimal vertex cover size. APPROX-VC-2, while faster than CNF-SAT-VC, exhibits a variable approximation ratio between 1 and 2, indicating a trade-off between accuracy and efficiency.

B. Best Algorithm for Industry Needs

Considering the industry needs and the trade-offs presented by each algorithm, APPROX-VC-1 is considered as the most suitable choice for real-world applications. Although it does not guarantee an exact optimal solution like CNF-SAT-VC, its significantly faster running time, paired with a consistent approximation ratio close to 1, positions it as an efficient and practical solution.

The decision is further supported by the comparison with APPROX-VC-2, where the marginal difference in running time does not outweigh the significant difference in approximation quality. Given that both approximation methods APPROX-VC-1 and APPROX-VC-2 are in linear time complexity, the preference should be given towards the one with a much better close to optimal minimum vertex cover result.

C. Practical Applicability

The practical applicability of APPROX-VC-1 could be demonstrated in scenarios where obtaining an exact optimal solution is not a strict requirement, and a balance between accuracy and efficiency is essential. APPROX-VC-1's linear time complexity ensures scalability, making it very suitable for handling larger graphs without a substantial increase in computation time, which would be typical in industry cases. The algorithm's predictable performance, indicated by a consistent approximation ratio around 1, makes it a reliable choice in time-constrained environments.

In summary, the findings highlight the significance of considering specific industry needs and priorities when selecting a vertex cover algorithm. APPROX-VC-1, with its acceptable trade-offs and efficient performance, stands out as the best-suited algorithm for various practical applications.

ACKNOWLEDGMENT

We extend our gratitude to Dr. Albert Wasef and other teaching assistants for their invaluable support in ECE 650 Methods & Tools for Software Engineering. Their responsive engagement on Piazza, insightful in-class discussions, and clarifications during office hours were instrumental in the success of our project.

REFERENCES

[1] Methods and Tools for Software Engineering, ECE 650 Section 001, Course ID, University of Waterloo, Fall 2023. [Online]. Available: https://learn.uwaterloo.ca

[2] Propositional Satisfiability, ECE 650 Methods & Tools for Software Engineering (MTSE), Fall 2023, Presented by Dr. Albert Wasef, Used by permission from Prof. Arie Gurfinkel.

[3] Introduction to Algorithms, fourth edition, April, 2022, Written by T.H. Cormen, C.E.Leiserson, R.L. Rivest, C. Stein

[4] CSE421: Design and Analysis of Algorithms, May 3, 2019, Lecturer: Shayan Oveis Gharan, Lecture 15 Approximation Algorithms.