1 Abstract

A vertex covers a set of vertices that includes at least one endpoint of every edge of the graph. Identifying the smallest possible vertex cover presents itself as an archetypal optimization problem in computer science. Such problems embody NP-hard complexities and can find resolution through approximation algorithms. Our project delves into solving this problem via three methods. In Section 2, we analyze the key issue of the project and in Section 3, we introduce methodologies along with the algorithm. In order to evaluate each method's effectiveness, section 4 presents and discusses running time and approximation ratio. Finally, in the last section, we will present a comprehensive summary of our final project.

2 Problem Analysis

After studying deeply through provided documents and professor's lectures, it's not hard to find our goal is to write a program that solves the Minimum Vertex Coverage (MVC) problem in graph theory, using multiple algorithms. What's more, we are required to evaluate the performance of these different algorithms and that involves in data analysis. We find that the MVC dilemma embodies a main goal which is identifying and implementing as few vertices as possible to cover all edges in the given graph efficiently.

It would be wise to use the recently completed code in Assignment #4 to implement our solution. We are asked to ensure that the software has a multi-threaded structure, with at least one thread dedicated to I/O and others assigned to the various MVC solutions.

We also have to analyze the software quantitatively for different inputs. We use two metrics, running time and approximation rate, to describe the efficiency of each method.

For each graph with a number of vertices between 5 and 50, we measure the running time and the approximation rate. Additionally, the code should calculate the mean and standard deviation for each vertex count over a hundred runs, which can be an important step.

We evaluate our code based on three dimensions: the correctness of the implemented algorithms and the quality of the graphs produced. For sure we must pass all text cases and think of different input conditions. If we want to extend CNF-SAT-VC scale to larger instances through some specific methods, that of course requires significant algorithmic optimization.

3 Methodologies and Algorithm

In the project, we delve into three distinct methodologies designed to address the minimum vertex cover problem: CNF-SAT, APPROX-1, and APPROX-2. Each solution presents a unique approach to solve this classic optimization problem.

1: CNF-SAT

We utilize CNF-SAT as the first method for solving the VARTEX-COVER problem The CNF Satisfaction Questionnaire (CNF-SAT) is a variation of the Satisfaction Questionnaire that defines a Boolean formulation of Correlation Normal Form (CNF), thus indicating that each sentence is either a dissection of letters or a collection of sentences which is only one lexical sentence Variations or omissions are such letters. An algorithmic strategy for solving a problem by reducing it to another problem by polynomial time reduction. In this example, we transform the VERTEX-COVER problem into a CNF-SAT problem. We designed the minimization process to run in polynomial time with respect to the size of the input. After successfully converting the problem to CNF-SAT, we use the SAT solution called “Minisat”. The unique software created by the researchers is minimal and open source for all students, and its main purpose is to discover new methods and techniques in solving the SAT. They recognized how effective and complex it was and saw it as an appropriate tool for this approach.

2: APPROX-1

The implementation of APROX-1 is a second solution that takes a more empirical approach to the problem. The APPROX-1 method must first identify the maximum degree level in the graph. Vertex angles represent the number of incident edges. Firstly we find a vertex which has the highest degree and then we add it to the vertex cover and throw away all edges incident on that vertex. After that we just repeat upper steps until no edges remain.

3: APPROX-2

The third method is a modification of the APPROX-1 algorithm. Because we can see with the name ”APPROX-2”, most of its core idea is the same with the former one. Instead of starting from the highest point, we start by picking a random edge (u, v) and subtract from it. ”u” and ”v” sections are added and these edges are joined together to form a top layer. All edges belonging to ”u” or ”v” are to be removed again and this process is repeated until there are no edges, similar to APPROX-1.

4 Data Analysis and Discussion

To evaluate the efficacy of the algorithms, we conducted an assessment of their execution duration and the approximation quality. This evaluation involved the generation of ten unique graphs for vertex counts ranging from 5 to 50, in increments of 5, the run of each set of vertex with edge value 10 times, and calculated the mean and the standard deviation for each set edge of vertex. We will analyze the algorithms in two different aspects

1: Running Time

We took careful notes on how long each part of the program ran when we tested different methods. After, we looked at these numbers to find out their average and how much they changed from this by calculating the standard deviation for each vertex. We used tools in Python, NumPy

for doing math calculations, and Matplotlib to draw a chart with these values. To show our results well, we made two different charts: Figure 1 shows how long APPROX-1 and APPROX-2 take, and Figure 2 is for looking at CNF-SAT's running time. In the two pictures, we use the x-axis to show how many vertices there are and the y-axis for how long it takes in milliseconds. This helps us see very well how different algorithms perform when comparing them.

In the first picture with graphs, we can see a clear pattern for APPROX-1 and APPROX-2. Both ways of solving problems take longer when there are more points to connect. This makes sense because their rules make them naturally slower as vertex get bigger. The yellow line, which is for APPROX-2, takes more time to run and uses more of the computer's power because APPROX-2 tries every possible solution one by one. On the other side, APPROX-1 which you see as the blue line, shows us a shorter time for running. This is because its way of doing things is simpler - it carefully picks out only those points with many connections.

When we look at the CNF-SAT algorithm, the second picture shows a big change in how long it takes to run when there are more points. The time to finish increased very fast, like many other NP-complete problems that take too much time and can't be finished quickly. With the input vertices increasing, we can see that the running time climbs dramatically on graphs with a logarithmic scale. The points of data show when we get close to having 20 vertices, then taking time to work out the CNF-SAT algorithm starts to be not doable for our test - that's why no more data after this is placed on the picture.

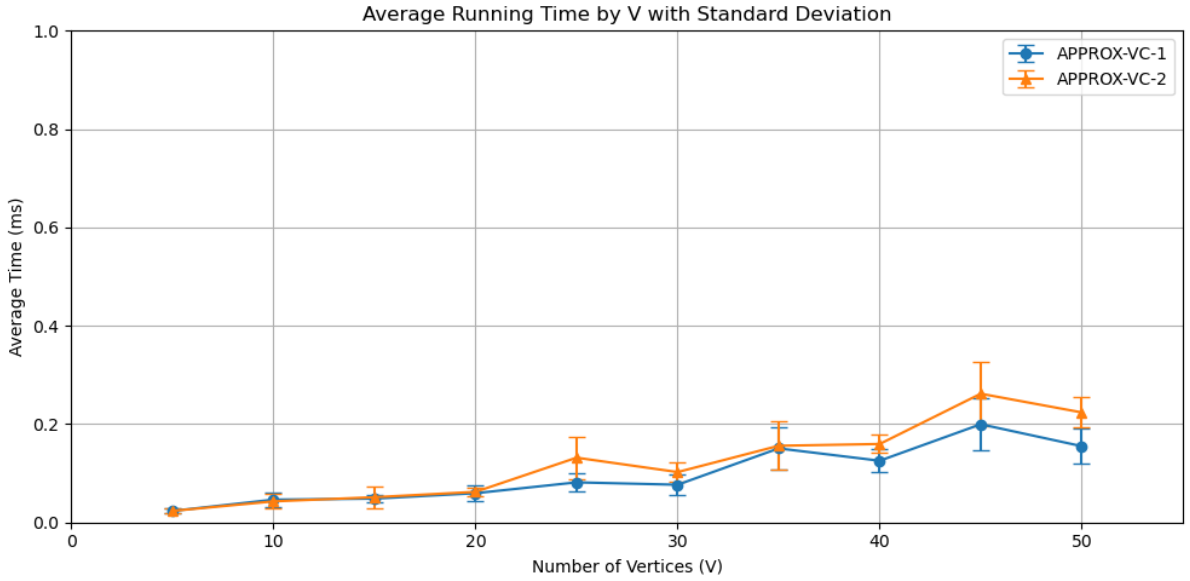

Figure 1: Running time of APPROX-1 and APPROX-2

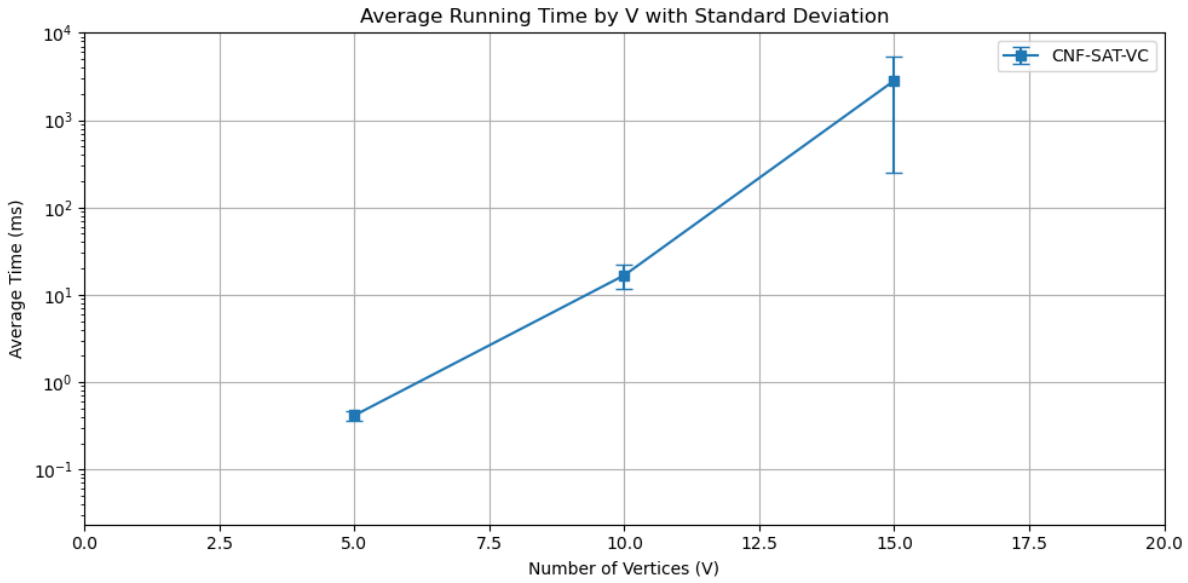

Figure 2: Running time of CNF-SAT.

2: Approximation Ratio

In our research, we carefully checked how two smart methods, named APPROX-1 and APPROX-2, perform by figuring out their approximation ratios. These numbers are very important because they tell us how well each method gets close to the best possible answer. The approximation ratio is a measure that compares how big the vertex cover from the heuristic is to the smallest one possible. This thing we look at is very important when we want to know if these shortcut methods are good for tricky math problems about covering points with lines.

In the Figure 3, we show pictures of these approximation ratios. The x-axis counts how many vertices are in our graphs. The y-axis tells us the number for how close our estimates come to the exact value. We picked points for our graph study every five steps, so we looked closely at 5, then 10, and finally 15 points. We did this on purpose because, after 15 vertices, there is no output for the CNF-SAT method.

From the picture in Figure 3, we see that for different sizes of graphs, APPROX-1's approximation ratio stays near the perfect one-to-one ratio. This closeness means APPROX-1 can make vertex covers almost as small as the smallest ones possible, showing it works very well with all graph sizes looked at here.

However, the vertex covers made by APPROX-2 show a bigger difference from the best answer produced by CNF-SAT. The chart shows that when there are more vertices, the ratio trend from APPROX-2 goes down, but it seems to be a not-so-good estimate than what we get with APPROX-1. The result shows that APPROX-2 does not keep the same correctness when producing the vertex cover answer.

By examining deeply, we understand important things about the balance between two shortcuts. APPROX-1 looks to care more about being correct, but APPROX-2 perhaps uses speed or easy calculations even if it means its estimate is less precise. Insights like these are very important to know how using these algorithms works in real life, where choosing between quickness and being right is usually a key thing to think about.

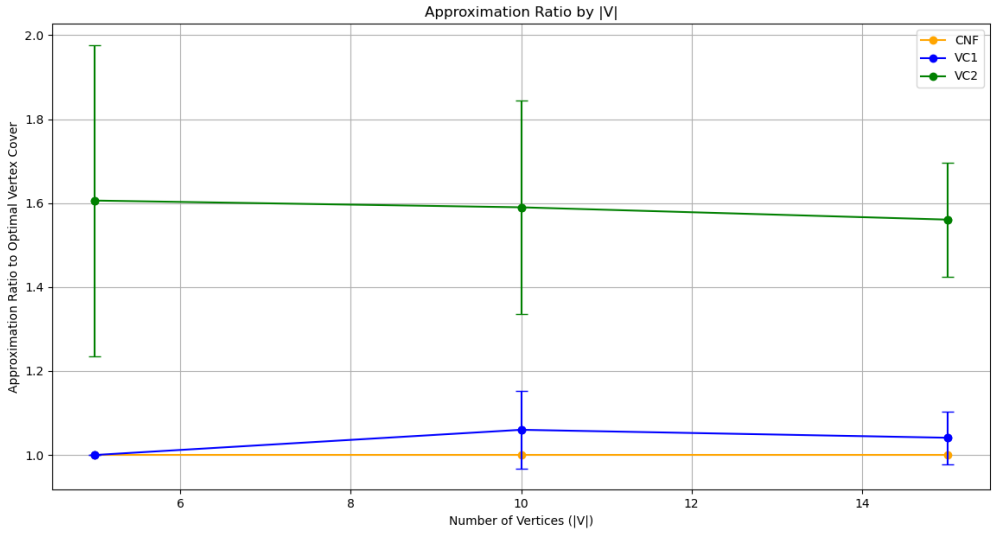

Figure 3: Approximation Ratio

Conclusion

In this report, we discussed the algorithm implementation for CNF-SAT, APPROX-1, and APPROX-2 in detail. In addition, we tested each algorithm's performance by counting the running time for each algorithm and comparing the approximation ratio for APPROX-1, and APPROX-2 with the CNF-SAT solver, we give the conclusion which CNF-AST is the NP problem which we need to optimize it for get the result when input vertex is large, also the APPROX-1 is the related good algorithm to solve the vertices coverage problem compare to other two algorithms in terms of balance the waiting time and the computational resource.