Performance Analysis of SAT solver with other apporoaches in finding Vertex Cover of graph

Name: Yican SHI

Department of Electrical and Computer Engineering

y233shi@uwaterloo.ca

Abstract. Three different approaches are analyzed in this report to find Vertex Cover of a gragh. Three approaches include SAT Solver and other two aproximation algorithms and I then use CPU running time and approximation ratio to analyze efficiency of each approach.

1. Introduction

In the mathematical discipline of graph theory, a vertex cover (sometimes node cover) of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The problem of finding a minimum vertex cover is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm[1]. In this assignment, I use the approach which is based on a polynomial time reduction to CNF- SAT, and with a 64-bit SAT solver (SAT). Also the other two approximation algorithms are developed to settle out the minimum vertex cover problem. As in these three approaches, The SAT solver checks every combination of vertices in order to find minimum vertex cover set, so SAT solver is guaranteed to be optimal thus giving minimum vertex cover set. The approximation algorithms cannot always be optimal and give me the vertex cover set that is not minimum sometimes. The SAT solver checks every combination of vertices in order to find minimum vertex cover set.

In the following part of this report, I analyze the performances of each algorithm using running time and approximation ratio to present how result change for different approaches.

2. Analysis: running time and approximation ratio

Polynomial time reduction of vertex cover is used to settle the problem. Except for the main (I/O) thread, there are three more threads, each for a different approach of the minimum vertex cover problem: the SAT Solver, first approximation algorithm and second approximation algorithm. I explain my result of these approaches in the following subsections.

2.2 Experimental Environment

[1] https://en.wikipedia.org/wiki/Vertex_cover

The code is run in ecelinux sever. The machine is a NoMachine server with a 4-core 3.8GHz AMD FX CPU with 32G of ECC RAM (ecelinux sever website). The output result could vary in different sever or computer with different configurations.

2.1 Running time comparison

Among all three approaches, SAT solver takes the maximum running time to get the final vertex cover set. I display the running time result with three graphs. In order to get CPU run time I use pthread's time function pthread_getcpuclockid(). I also use Gnuplot to draw the graphs. Average running time is calculated from vertices four to twenty. We use Graphgen to produce graphs with different vertices, and we test ten graphs of a certain number of vertices. Fig.1 and Fig.2 is for the running time of SAT solver, the former without logscal and the latter with logscal so as to give a better understanding of running time. Fig.3 is for the average running of other two approximation approaches.

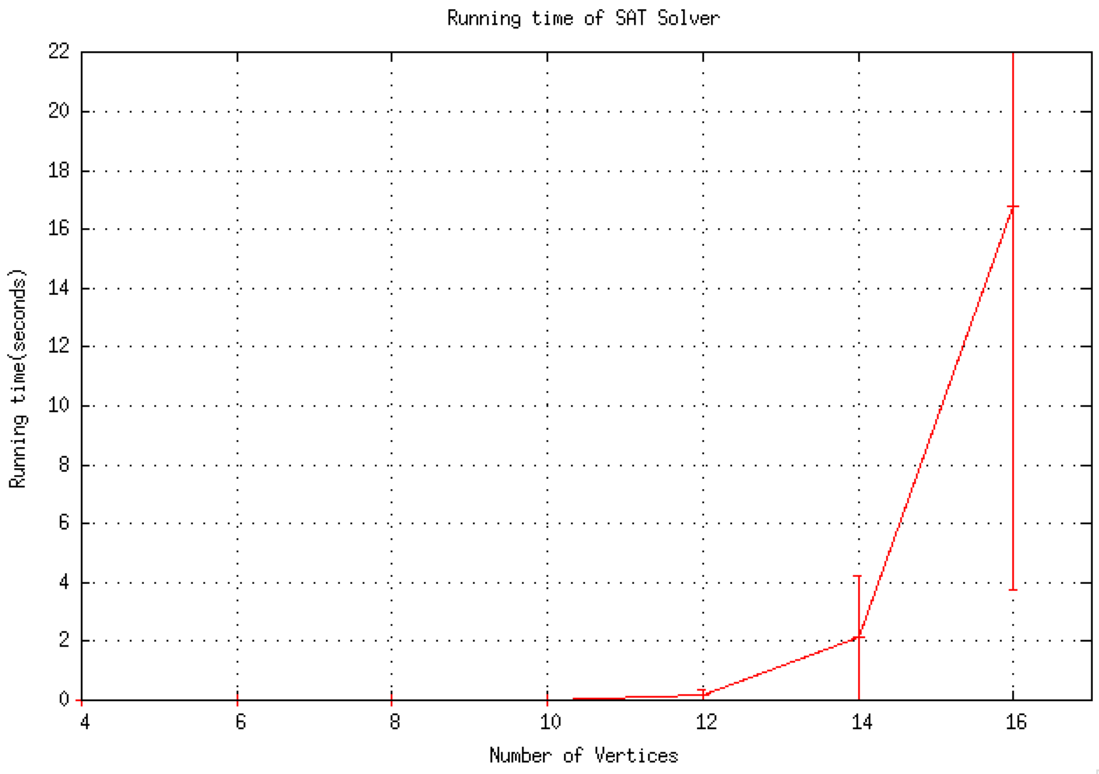

Fig.1 Running time of SAT Solver (without Log Scale)

From Fig.1 (Running time of SAT Solver) we can clearly see that SAT solver run very fast when there is less number of vertices, for example from 4..10. As vertices number increases, the performance time is getting longer. It takes much more time to find the vertex cover when number of vertices increases to fourteen. We can figure out there is an exponent increase in running time from vertices number fourteen to sixteen. The increase is expected to continue as vertices number increase to be larger. When the vertex number reaches larger than 18, it will take more than hours to calculate the vertex cover. The main reason for this running time increase is that the number of clauses will increase very fast as vertices number gets larger. And we can calculate that the number of clauses is $v * (v - 1) * k/2$ where k is the minimum size of vertex cover set. In addition to the vertices number increase, the edge number also increases causing the SAT solver to satisfy even more clauses. On the other hand, with less number of vertices, there are less clauses and less number of variable combinations thus taking less running time to find the vertex cover set. The above reasons give justifiable explanation for the exponential growth of the SAT solver.

From Fig.1 and Fig.2, I also notice that for certain number of vertices, the standard deviations, which can be shown by errorbars, are very big. The reasons for these high standard deviations are that for different graphs with the same vertices number, there are different connections between edges. Because of that, the vertex number and exact vertices found varies much. In some cases, the variation can be even more. This makes the standard deviations' variation justified.

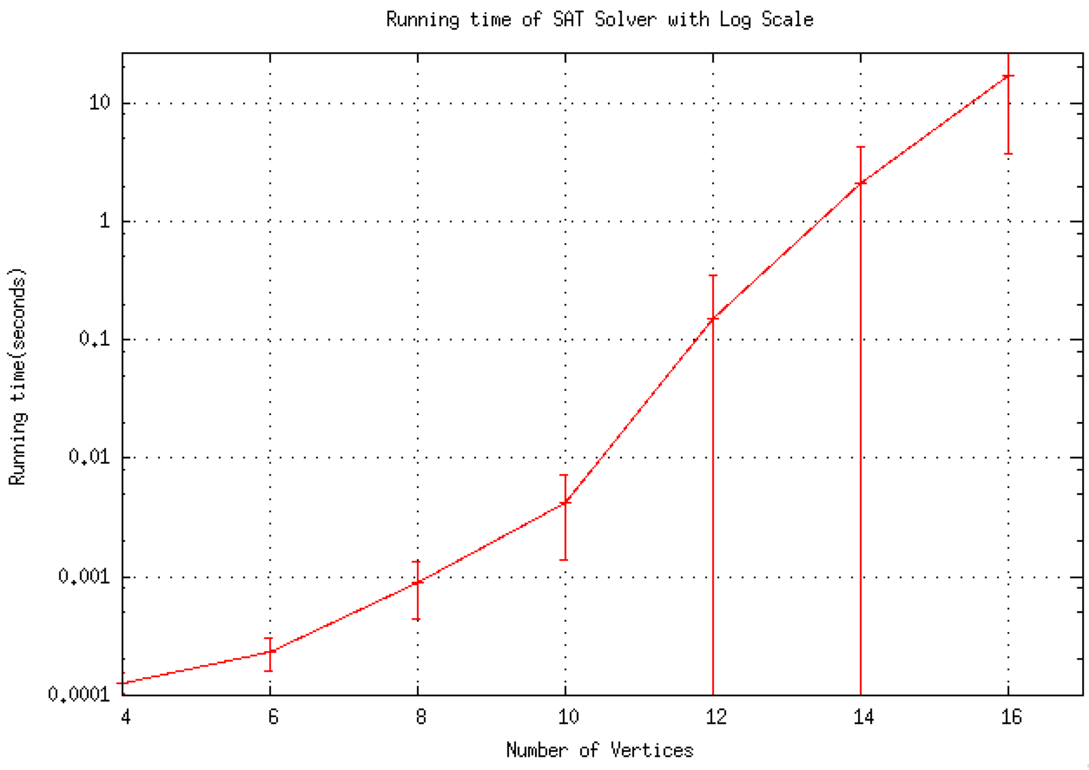

Fig.2 Running time of SAT solver (with Log Scale)

From Fig.3 we can see that in most cases running time for both approximation algorithms increase as number of vertices grows. In most cases, running time of Approx1 is much more than Approx2. There is a basic difference between SAT solver and the approximation algorithms. Namely neither of the two approaches shows exponent increase as the SAT solver does. The curves increase smoothly. This is because SAT solver has to satisfy every possible variable combination in order to find out the minimum vertex cover set. For the two approximation approaches, there is no need to check all possibilities. This makes the substantive discrepancy between the approximation approaches and SAT solver. To analyze the two approximation algorithms separately, we need to look into the time complexity. In Approx-VC -1, we take vertex with most incident edges then delete the edges related to it. I need to check all edges to find the vertex that is with maximum incident edges thus taking v^2 comparisons in the loop. Its time complexity is $O(v^2)$. In order to delete edge, I use a two dimensional array. We suppose the number of edges denoted by E. We conclude that time complexity is $O(E) + O(v^2) + O(v^2)$. I keep a two dimensional array to store graph and two one-dimensional arrays to store vertex. So the space complexity is $O(v)$. For Approx-VC-2, I pick the first edge and delete edges related to both vertices of it. This process repeated until there is no edge remained. In this case, the time complexity is $O(E)$. There is also a temporary array to keep graph, as Approx-VC-1, it takes $O(v^2)$. The time complexity is $O(E) + O(v^2)$. Compared to Approx-VC-1, it is no need to do the v^2comparisions loop so as to search for the maximum incident vertex thus taking less running time to find the vertex cover as shown in Fig.3.

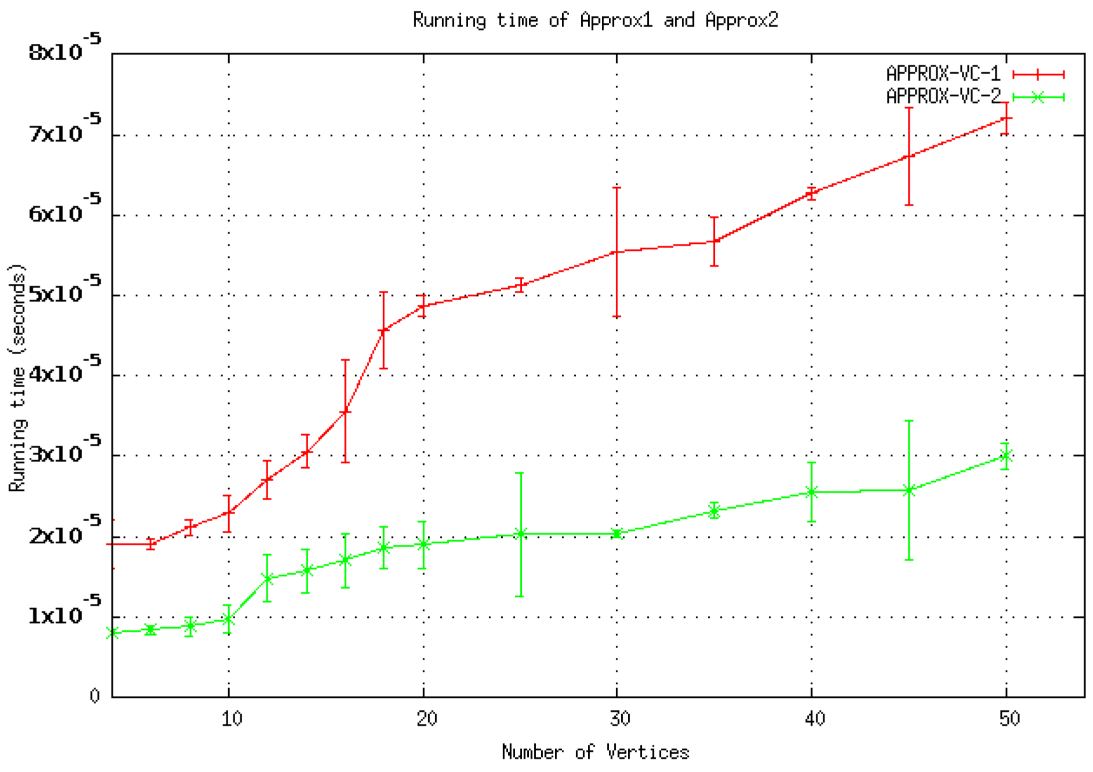

Fig. 3 Running time of Approx1 and Approx2

2.3 Approximation Ratio

The approximation ratio is calculated using the following formula:

$$Appro\ Ratio\ = \ ^{sizeofvertexcoverofCNF-SAT}/_{sizeofcomputedvertexcover_i}$$

We can see the result in Fig.4. As can be shown, the algorithm whose approximation ratio is close to 1 is the better one because it is more accurate than the other. We can figure out the Approx-VC-1 is more closer to 1 than Approx-VC-2. So Approx1 is performing better than Approx2 in terms of approximation ratio. By observing the data I gathered from all the generated graphs, I find that the result calculated by Approx-VC-1 is very much close to the SAT solver's, which is the optimal vertex cover solution.

From the curve of Approx-VC-2, we can see the approximation ratio is much higher than Approx-VC-1. In this case, Approx2 does not search for most incident edges and throw away edges one by one in a row. So the output is not very accurate.

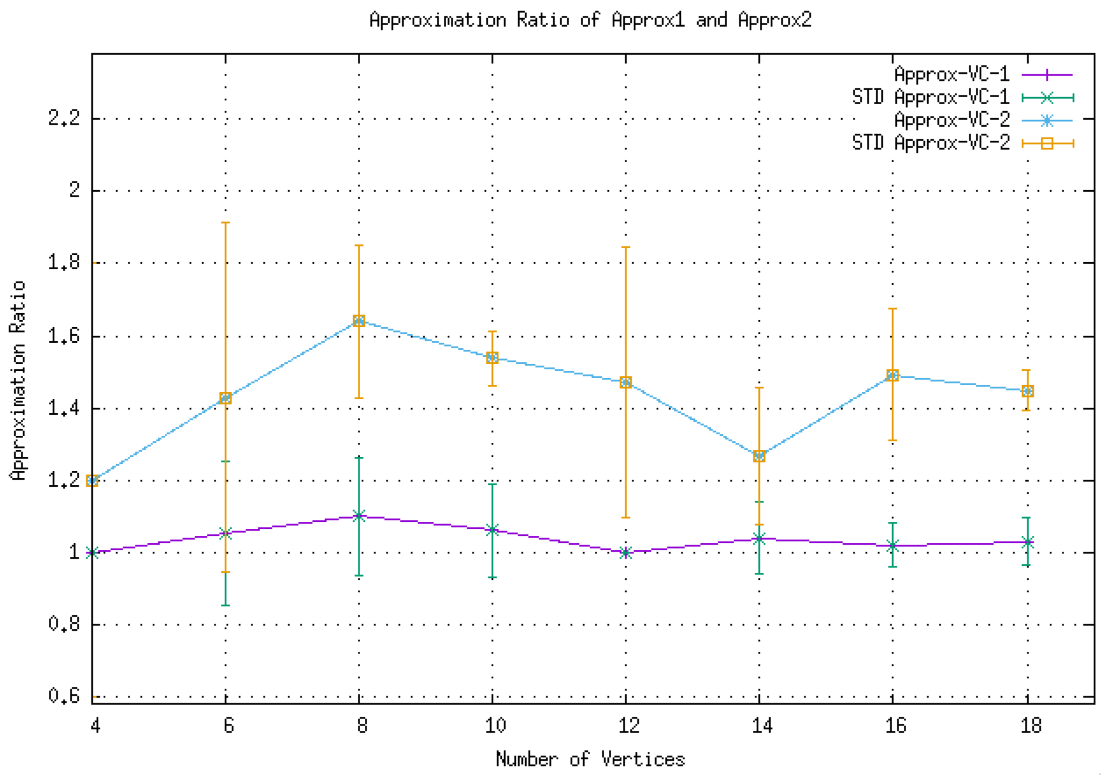

Fig.4 Approximation ratio of Approx1 and Approx2

3. Conclusion

As known to all, CNF-SAT is always optimal result, which is the minimum vertex cover set. Its drawback is that SAT solver is not very efficient when number of vertices increase to certain degree. On the other hand, the approximation algorithms are efficient both in time and space complexity, but they are not very accurate and cannot guarantee optimal.