ECE650 - METHODS AND TOOLS FOR SOFTWARE ENG

UNIVERSITY OF WATERLOO

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Analysis of Minimum Vertex Cover Using Various Algorithms

Authors:
Urja Panchal (ID: 21101631)
Rushi Raval (ID: 21086166)

Date: December 6, 2023

1 Introduction

According to graph theory, a set of vertices that has at least one endpoint for each edge in the graph is called a vertex cover [1].

Formally, a vertex cover ***V'*** of an undirected graph $G = (V,E)$ is a subset of V such that every edge in the set of vertices ***V'*** has at least one endpoint in the vertex cover ***V'***, meaning that $uv \in E \Rightarrow u \in V' \vee v \in V'$.

A minimum vertex cover is a vertex cover of smallest possible size. The problem of finding a minimum vertex cover is a classical optimization problem. It is NP-hard, so it cannot be solved by a polynomial-time algorithm if $P \neq NP$.

Here we have used three approaches in an attempt to solve minimum vertex cover problem and analyzed the optimization achieved by each algorithm.

- ***CNF SAT***: Minisat SAT solver API-based Boolean satisfiability.
- ***ApproxVC1***: Greedy approach for selecting the most weighted vertex each time until all edges are marked.
- ***ApproxVC2***: Bruteforce approach for selecting every vertex from each edge without duplicating the vertex.

2 Solution

Lets discuss three solutions to the minimum vertex cover problem.

2.1 CNF-SAT

CNF-SAT means Conjunctive Normal Form Satisfiability, is a problem in computer science and logic. It involves representing a logical expression in conjunctive normal form, where clauses are connected by conjunctions (AND), and then determining if there exists an assignment of truth values to variables that makes the entire expression true.

CNF is a specific way of representing boolean satisfiability (SAT) problems. In CNF, a formula is expressed as a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. A literal is either a variable or its negation.

A SAT solver is used to determine if it is possible to find assignments to boolean variables that would make a given expression true, if the expression is written with only AND, OR, NOT, parentheses, and boolean variables.

If it's satisfiable, most SAT solvers (including MiniSAT) can also show a set of assignments that make the expression true.

Using *CNF-SAT* to solve minimum vertex cover problem:

Consider a graph G with vertices V and Edges E inputed to our program. The program then translates the graph into a boolean formula in CNF. Each vertex is represented by a boolean variable.
Then we perform the encoding in order to obtain boolean formula in cnf format. Following are the conditions that are used for encoding:

- At least one vertex is the ***ith*** vertex in the vertex cover.
- No one vertex can appear twice in a vertex cover.
- No more than one vertex appears in the mth position of the vertex cover.
- Every edge is incident to at least one vertex in the vertex cover.

Pseudocode:

Algorithm 1 doCNF

```
1:  function DOCNF
2:      for i in [1,k] do
3:          Solver.addClause([x_i1(i) ∨ x_i2(i) ∨ ... ∨ x_in(i)])
4:      end for
5:      for m in [1,n] do
6:          for p,q in [1,k], p < q do
7:              Solver.addClause([¬x_mp(m) ∨ ¬x_mq(m)])
8:          end for
9:      end for
10:     for m in [1,k] do
11:         for p,q in [1,n], p < q do
12:             Solver.addClause([¬x_pm(p,m) ∨ ¬x_qm(q,m)])
13:         end for
14:     end for
15:     for ⟨i,j⟩ in E do
16:         Solver.addClause([x_i1(i) ∨ x_i2(i) ∨ ... ∨ x_ik(i) ∨ x_i1(j) ∨ x_i2(j) ∨ ... ∨ x_ik(j)])
17:     end for
18:     if Solver.solve() then
19:         return true
20:     end if
21: end function
```

2.2 ApproxVC1

ApproxVC1 is a greedy approach that selects a vertex that has maximum incident edges and removes every edge from that set and adds that vertex cover.

Greedy is an algorithm that builds up a solution step by step and always chooses the next step that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are the best fit for Greedy.

Pseudocode:

Algorithm 2 ApproxVC1

```
1:  function APPROXVC1(graph)
2:      vertex_cover ← empty set
3:      while graph has edges do
4:          max_vertex ← vertex with maximum incident edges in graph
5:          Add max_vertex to vertex_cover
6:          incident_edges ← set of edges incident to max_vertex
7:          Remove max_vertex from graph
8:          for each vertex in graph do
9:              Remove incident_edges from edges of vertex
10:         end for
11:     end while
12:     return vertex_cover
13: end function
```

2.3 ApproxVC2

ApproxVC2 is a bruteforce approach that selects each edge and then adds its vertices to our vertex cover solution such that there are no duplicate vertices added to our vertex cover set and such that all edges are traversed at the end.

Pseudocode:

Algorithm 3 ApproxVC2

```
1:  function APPROXVC2(graph)
2:      vertex_cover ← empty set
3:      while graph has edges do
4:          (u, v) ← graph.getRandomEdge()
5:          Add u and v to vertex_cover
6:          Remove attached edges from edges of vertex u and v
7:      end while
8:      return vertex_cover
9:  end function
```

To execute all of these algorithms concurrently, we multithreaded our program. For multithreading, We used thread and pthread.h lib for multithreading.

Threading

Threading is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of the CPU. Each part of such a program is called a thread, and threads are lightweight processes within a process.

Threading can be used in C++ to improve the performance and responsiveness of an application, especially when it involves resource-intensive or I/O-intensive tasks that can be parallelized.

3 Analysis

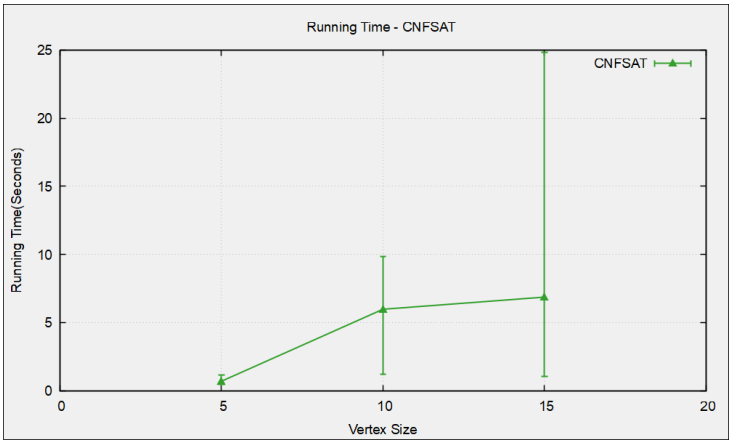
To measure the efficiency, we generated 10 graphs for every vertex size and computed running time with our algorithm, we can analyze our algorithm based on two premises:

- ***Running Time***: Total time taken by the algorithm to generate its vertex cover.
- ***Approximation Ratio***: Ratio of the size of the computed vertex cover to the size of an optimal (minimum-sized) vertex cover.

3.1 Running Time

We recorded the running time of our algorithm by using $pthread\_getcpuclockid()$ library functions. By placing a $clockid$ at the beginning of our algorithm, we can begin recording the algorithm's running time.

After calling $pclock()$, we get the exact time it takes our algorithm to compute vertex cover, in milliseconds, at the end of our algorithm.

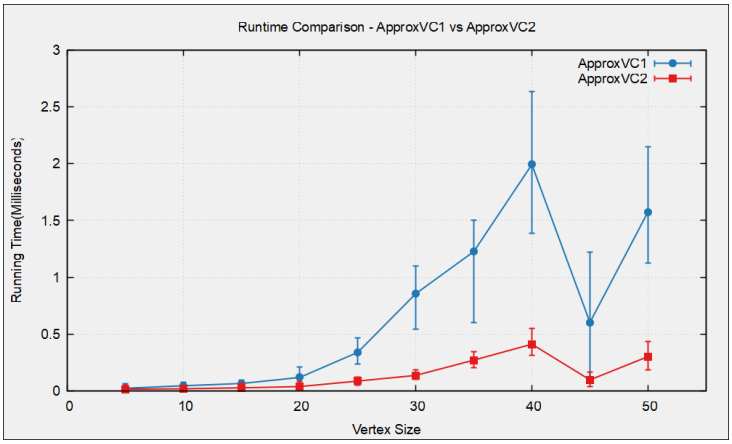


Running time of CNF-SAT algorithm

The efficacy of a ***CNF-SAT*** implementation in solving the minimum vertex-cover problem across varying vertex sizes. Results indicate a notable increase in mean execution times from 0.727 seconds for vertex size 5 to 5.999 seconds for size 10 and 6.889 seconds for size 15.

Minimum and maximum times further reveal the variability in computational effort, with the maximum time reaching 24.834 seconds for size 15.

We could only determine the minimum vertex cover of a graph till vertex size 15, beyond that only a few cases can be solved. To solve every case, a reduction can be done by modifying our encoding steps. We later talk about the approaches for reduction.



Running time of ApproxVC1 and ApproxVC2

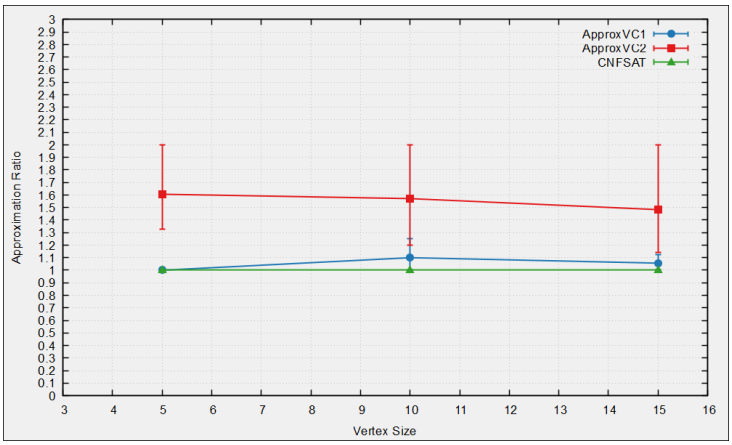
The running time of ***ApproxVC1*** increases exponentially from vertex size 20 to vertex size 40, then drops again for vertex size 45. The rising trend until Vertex size 40 can be attributed to the Graph Structure, which has a significant number of edges generated by *graphGen*. The same cause accounts for the drop in vertex size 45.

The ***ApproxVC2***'s running time, on the other hand, increased linearly until vertex size 40. ***ApproxVC2***'s execution time decreased slightly for vertex size 45 owing to the simpler graphs formed by *graphGen*.

3.2 Approximation Ratio

The approximation ratio is calculated as the ratio of computed vertex cover to the minimum optimal vertex cover. The Approximation plot has a number of vertices on the X-axis and approximation ratio on the Y-axis.

The plot shows that the approximation ratio for ***CNF-SAT*** is constant and equal to one for all graphs from vertex size 5 to 15. The standard deviation for the CNF is not evident, indicating that no variance was observed, confirming the constant ratio.



Running time of ApproxVC1 and ApproxVC2

We may conclude that ***CNF-SAT*** provides the best answer; nevertheless, for vertex counts more than 15, the ***CNF-SAT*** algorithm exhibited exponential growth in running time, making it difficult to compute vertex cover and difficult to realize in polynomial time.

For ***ApproxVC1***, the ratio follows a linear trend, with the majority of values close to 1 for all vertices from 5 to 15. We detect a minor divergence for vertices 10 and 15, but it is not significantly different from the ideal value.

The approximation ratio is shown to be higher for ***ApproxVC2***, nevertheless it trends linearly from vertex 5 to 15. We find a large variance as a result of the algorithm used to compute vertex cover.

Based on this plot, we may conclude that the ***ApproxVC1*** is the closest to the optimal solution and has a polynomial running time, making it suitable for larger graphs.

4 Analysis

This study assisted us in analyzing the time complexity of three distinct techniques for computing vertex cover. The ***CNF-SAT*** found to be the most efficient in terms of optimal solution; however, its running time increases exponentially for larger graphs.

The ***CNF-SAT*** technique is built on boolean satisfiability clauses. Because of the exponential time complexity, the larger the graph, the more clauses. One potential remedy to this problem is to reduce the CNF-clauses.

According to the ***"Applying Logic Synthesis for Speeding Up SAT"*** [2] research report, Tseitin transformation may allow the ***CNF-SAT*** to compute the vertex cover more efficiently.

The Tseitin transformation's primary purpose is not to reduce the number of clauses, but to convert a formula into an equivalent CNF form that can subsequently be efficiently handled by SAT solvers.

The procedure entails adding new auxiliary variables to the original formula to represent subformulas. These auxiliary variables aid in the division of complex formulations into simpler clauses.

There are other advanced SAT solvers that can provide results. ***ApproxVC1*** and ***ApproxVC2*** are faster, however they do not provide an optimal answer.

References

[1] Vertex cover. pages 2

[2] Niklas S¨orensson Niklas Een, Alan Mishchenko. Applying logic synthesis for speeding up sat. pages 7