Analzying Different Approaches for finding Vertex-Cover ECE 650 - Project Report[⋆]

Uditya Laad[20986041] (ulaad@uwaterloo.ca)
Swapnil Baluja[20933848] (s3baluja@uwaterloo.ca)

University of Waterloo, Waterloo, ON, N2L 3G1, CA

Abstract. We are comparing the time and output effieciency of different algorithms used for finding Minimum Vertex-Cover of a Graph. The algorithms being considered are CNF-SAT-VC, APPROX-VC-1, and APPROX-VC-2.

Keywords: Running Time · Approximation Ratio · Number of Vertices

Abbreviations:

n	Number of Vertices
k	Size of Proposed Vertex Cover
E	Edges of Graph
Std. Dev.	Standard Deviation
VC-1	APPROX-VC-1
VC-2	APPROX-VC-2
CNF	Conjunctive Normal Form
SAT	Satisfiable

1 Algorithms Being Considered

1.1 CNF-SAT-VC

```
Input: G=(n,E): Graph with $n$ number of Vertices and $E$ Edges
Output: VertexCover[]

Initialize k = 1
while $k \leq v$ do
    minisat_CNF = createClausesForVertexCoverProblem()
    SATISFIABILITY, vertexCover = Minisat.Solve(minisat_CNF)

    if $SATISFIABILITY == true$ then
        return vertexCover
    end

    k++
end
```

Algorithm 1: CNF-SAT-VC

.

This algorithm makes use of Minisat Solver [5] to find the optimal Vertex-Cover (i.e. the Minimum possible Vertex-Cover, required to cover all edges)

[⋆] Supported by University of Waterloo

1.2 APPROX-VC-1

```
1. Set: v_high = Vertex with highest degree (most incident edges).
2. Add 'v_high' to VertexCover[]
3. Throw away all edges incident to 'v_high'.
4. Repeat $'Steps\ 1-3'$ till no edges remain.
```

Algorithm 2: APPROX-VC-1 [2]

1.3 APPROX-VC-2

```
1. Set: first_edge = First available 'Edge(src,dest)' in the Graph
2. Add both 'src' and 'dest' to VertexCover[]
3. Throw away all edges attached to 'src' and 'dest'.
4. Repeat $'Steps\ 1-3'$ till no edges remain.
```

Algorithm 3: APPROX-VC-2 [2]

2 Analyzing with Original Encoding[1]

2.1 Running-Time

We compute the Mean and Standard Deviation of CPU-Time taken by each algorithm for multiple graph inputs, grouped over 'No. of Vertices (v)'.

CNF-SAT-VC .

From Fig. 1, we can make out that 'CNF-SAT-VC' takes the longest to find the Vertex Cover. It competes well with the other 2 algorithms till $n \leq 10$, as compromising on a few milliseconds is no big deal, especially when we take the optimal vertex-cover into consideration.

But for $n \geq 10$, the running-time drastically shoots up. This is because, with increasing number of Verticies, we also have more and more edges to cover; leading to a higher value of 'k' to find an optimal solution. This increases the number of clauses that are being input to the Minisat-Solver to determine satisfiability. Also, the clauses being added are not definite, but rather complex.

The total number of clauses required for each value of 'k' is given by the following equation: [1]

$$Number\ of\ Clauses = k + n\binom{k}{2} + k\binom{n}{2} + |E| \tag{1}$$

Where: 'k' is the size of proposed Vertex-Cover, 'n' is the Number of Vertices, and 'E' contains all the edges in the Graph

This leads to a CNF with a lot of complex clauses, that require more resources and time to be solved by Minisat. (Note: that this is an NP-Complete problem; i.e: it cannot be solved in poynomial time.)

Also, we start with k=1, and keep using the minisat solver until we get a satifiable result. Hence, even for a slight increase in satifiable 'k', the required CPU-time shoots up by a huge margin; as we are solving '$\forall$ k $\in$ [1, Satisfiable_k]'.

However, we notice that for n $\geq$ 20, the running time is exactly 10 seconds, which contradicts the above reasoning. This constant run-time is the result of 'timeouts'. For large enough 'n', it is possible that 'CNF-SAT-VC' may take a very long time (could be hours or days or even weeks), which is obviously not feasible. Hence, we timeout the algorithm once a specified amount of CPU-time has been utilised for any run.

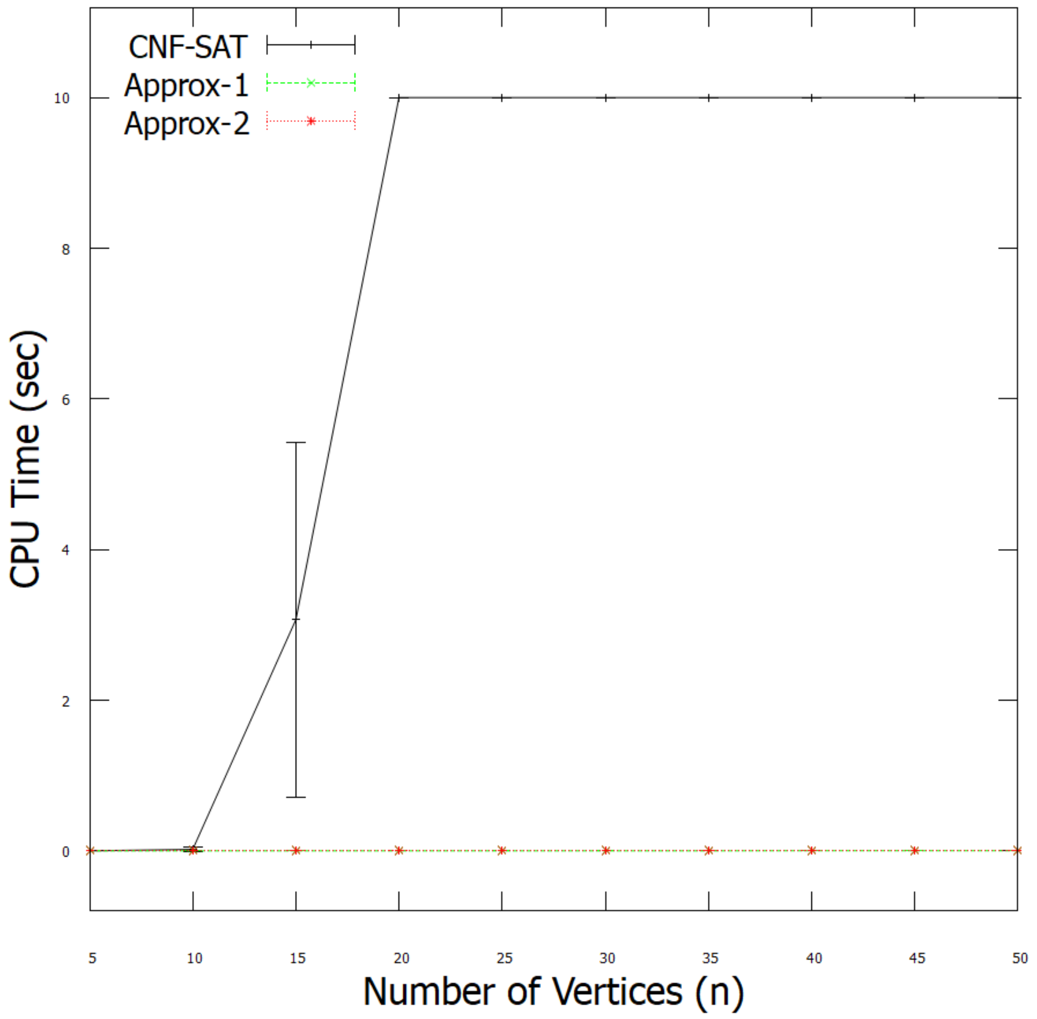

Fig. 1. Graph of 'Running (CPU) Time (sec)' v/s 'Number of Vertices (n)', for each algorithm being considered and using Original Encoding [1] in CNF-SAT-VC

Standard Deviation, which is represented by a vertical bar at each marked coordinate in Fig. 1, shows the amount by which the running-time of each input at 'n' differs from the mean running-time at that 'n'.

- At n≥ 20, we see that the standard deviation is Zero. This is because 'CNF-SAT-VC' times out for each input provided at n≥20. And since the timeout value is considered when performing analysis, there is no difference between the running times of each input; resulting in no standard deviation between those runs.
- Table-1 shows the values we've plotted on the graph (Fig. 1) for 'CNF-SAT-VC'. When compared against their respective mean values, we will observe that the standard deviations are quite large. This is because we have good variations in our input graphs. This also means that the time taken by CNF-SAT-VC depends heavily on the complexity of Graph input and that of the clauses derived from it.

Table 1. Mean and Standard Deviation in CPU Time - for CNF-SAT-VC

No. of Vertices	5	10	15	20
Mean (CPU Time)	0.000202	0.018633	3.066866	10.000000
Standard Deviation	0.000074	0.030873	2.354415	0.000000

APPROX-VC-1 and APPROX-VC-2 .

By looking at the graph (in Fig. 1), it appears as if APPROX-VC-1 and APPROX-VC-2 take the same amount of CPU-time to find the vertex cover. Some may even say they both take 0 seconds to produce the output. But, this is not true. The time taken by either algorithm does not exceed 1 millisecond for the tested data. And since our scale for 'Running Time' is in seconds, the graph gives an impression that they both take the same time. This is expected, since both these algorithms run in polynomial time, unlike 'CNF-SAT-VC' which was previously discussed.

Table-2 shows the values we've plotted for these two algorithms in Fig. 1. On comparing their mean CPU-time at each 'n', we can make out that the time taken by APPROX-VC-2 is about 45-55% less than that of APPROX-VC-1. The difference could possibly be a result of the overhead involved in finding the Vertex with highest degree for each iteration in APPROX-VC-1, which leads to more CPU-time; compared to APPROX-VC-2 which simply picks the first available edge in the list.

Table 2. Mean and Standard Deviation in Time - APPROX-VC-1 & APPROX-VC-2

No. of Vertices		5	10	15	20	25	30	35	40	45	50
APPROX-VC-1	Mean	0.000019	0.000042	0.000079	0.000139	0.000203	0.000273	0.000351	0.000469	0.000569	0.000686
	Std. Dev.	0.000005	0.000003	0.000006	0.000006	0.000014	0.000014	0.000018	0.000019	0.000029	0.000033
APPROX-VC-2	Mean	0.000013	0.000025	0.000041	0.000067	0.000096	0.000128	0.000155	0.000204	0.000245	0.000282
	Std. Dev.	0.000001	0.000002	0.000004	0.000006	0.000008	0.000010	0.000013	0.000009	0.000018	0.000017

A realtively smaller Standard Deviation at each 'n', also means that the Mean values are good enough to draw concrete conclusions with respect to CPU-time taken by APPROX-VC-1 and APPROX-VC-2.

Overall

– On the basis of above analysis, we belive it's safe to say that APPROX-VC-2 is the fastest of all the three algorithms being considered, with APPROX-VC-1 being the next best.
– CNF-SAT-VC still performs reasonably fast for n ≤ 10, although still not as fast as the other two. But falls behind for further values of 'n' and that too by a huge margin.
– This is not surprising, as CNF-SAT-VC is NP-Complete, whereas the other 2 algorithms run in polynomial time.

2.2 Approximation Ratio

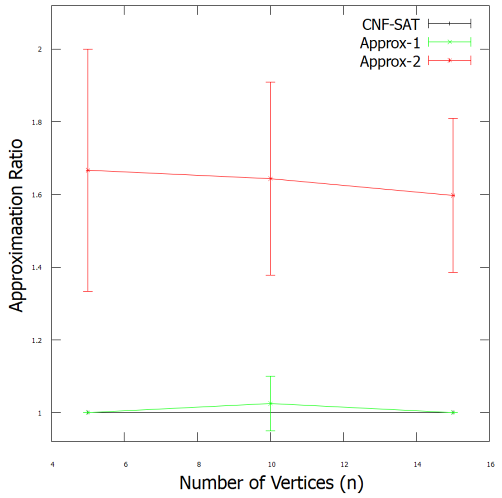

Fig. 2. Graph of 'Approximation Ratio' v/s 'Number of Vertices (n)', using Original Encoding [1] in CNF-SAT-VC

This will be explained in Section 3.2, since we have a larger set of data in the graph (to compare against) with the Optimized version of CNF-SAT-VC.

3 Optimizing CNF-SAT-VC

Note: Pendant Edge refers to an edge, whose either or both end-point(s) is a pendant-vertex (Vertex with only one incident edge).

The above analysis (in Section 2) was performed for the Encoding that had been provided by Professor Arie in Assignment_4.pdf [1]. The original encoding can be improved by revising it.

```
Input: G=(n,E): Graph with 'n' number of Vertices and 'E' Edges
Output: redundantVertices[]: Vertices that are not required in minimum vertex-cover.
        mandatoryVertices[]: (Unique) Set of vertices that must be a part of the minimum
vertex-cover.

Initialize x = 0
while $x < |E|$ do
    if E[x] is a Pendant Edge then
        if $|Incident\_edges\ of\ E[x].src| == 1$ then
            redundantVertices.add(E[x].src)
            mandatoryVertices.add(E[x].dest)
        end
        else
            redundantVertices.add(E[x].dest)
            mandatoryVertices.add(E[x].src)
        end
    end
    x++
end
mandatoryVertices.removeDuplicates();
.
```

Algorithm 4: Finding Redundant and Mandatory End-points in Pendant Edges

.

The revised reduction will be as follows: [1]
Given a pair (G, k), where $G = (V, E)$, denote $|V| = n$. Assume that the vertices are named $1, ..., n$. Construct F as follows.

$$Let\ A\ =\ Set\ of\ Mandatory-Verticies\ (which\ must\ be\ a\ part\ of\ Minimum\ Vertex-Cover)$$

$$Let\ B\ =\ Set\ of\ redundant\ vertices,\ (which\ are\ not\ required\ to\ be\ a\ part\ of\ Minimum\ Vertex-Cover)$$

$$(Both\ 'A'\ and\ 'B'\ are\ found\ using\ Algorithm-4)$$

$$Let\ N\ = Set\ of\ Vertices\ with\ no\ incident\ edges\ at\ all$$

$$Let\ 'R'\ be\ a\ set\ of\ verticies\ i \in \{[1,n] : such\ that\ i\ has\ no\ incident\ edge,$$
$$or\ i\ is\ a\ redundant\ endpoint\ in\ a\ pendant\ edge\};$$
$$i.e.\ \ R\ = \{N\ \cup\ B\}$$

NOTE: 'Eligible Vertices' are vertices that do not belong to $\{A\ \cup\ R\}$; and 'Remaining Spaces' refer to the spaces remaining in vertex-cover after adding all elements of 'Set A'.

The reduction uses '$(n - |R|)\, x\, k$' atomic propositions, denoted $x_{\mathrm{i,j}}$, where $i \in \{[1,n] - R\}$ and $j \in [1,k]$. A vertex cover of size k is a list of k vertices. An atomic proposition $x_{\mathrm{i,j}}$ is true if and only if the vertex i of V is the jth vertex in that list.

```
if $k < |A|$ then
    Cannot have a vertex-cover of size 'k' for Graph 'G', i.e. (G,k) is unsatisfiable.
end
else
    Use the following set of clauses to determine satisfiability:
    1] All Mandatory Vertices (in Set A) must be included in the Vertex Cover.
        $\forall i \in [1, |A|] :\ \ a\ clause\ (x_{\mathrm{A[i],i}});$
        $and\ \forall i \in [1, |A|],\ \forall j \in \{[1,n],\ \ j \neq A[i]\} :\ \ a\ clause\ (\neg x_{\mathrm{j,i}})$
        $and\ \forall i \in [1, |A|],\ \forall j \in \{[1,k],\ \ j \neq i\} :\ \ a\ clause\ (\neg x_{\mathrm{A[i],j}})$
    2] At least one eligible vertex is the ith vertex in the remaining spaces of Vertex Cover.
        $\forall j \in [|A| + 1, k] :\ \ a\ clause\ (x_{\mathrm{i1,j}}\ \vee\ x_{\mathrm{i2,j}}\ \vee\ ......)$
        $where :\ \ \{i1, i2, .....\} = \{[1,n] - R - A\}$
    3] No one (eligible) vertex can appear twice in the remaining spaces of Vertex Cover.
        $\forall m \in \{[1,n] - R - A\},\ \forall p, q \in [|A| + 1, k]\ \ with\ \ p < q\ \ :\ \ a\ clause\ (\neg x_{\mathrm{m,p}}\ \vee\ \neg x_{\mathrm{m,q}})$
    4] No more than one eligible vertex appears in the mth position of the remaining spaces
       of Vertex Cover.
        $\forall m \in [|A| + 1, k],\ \forall p, q \in \{[1,n] - R - A\}\ with\ \ p < q,\ :\ \ a\ clause\ (\neg x_{\mathrm{p,m}}\ \vee\ \neg x_{\mathrm{q,m}})$
    5] Every **Uncovered-Edge** is incident to at least one vertex in the vertex cover.
        $\forall\,(i,j) \in E, such\ that : i \notin A\ \ and\ \ j \notin A :$
        $a\ clause\ (x_{\mathrm{i},|A|+1}\ \vee\ x_{\mathrm{i},|A|+2}\ \vee\ ....\ \vee\ x_{\mathrm{i,k}}\ \vee\ \ x_{\mathrm{j},|A|+1}\ \vee\ x_{\mathrm{j},|A|+2}\ \vee\ ....\ \vee\ x_{\mathrm{j,k}}\,)$
end
```

Algorithm 5: CNF-SAT-VC - Revised Encoding

The condition $k < |A|$ in Algorithm 5 helps to skip running the minisat-solver for some UNSATISFIABLE values of 'k' (in certain cases), which help save time, specific to those cases.

The revised encoding uses less atomic propositions, compared to the original one. Also we have a lot of definite clauses (with one atomic proposition). This helps simplify the CNF, which then helps Minisat solve it faster.

3.1 Running Time

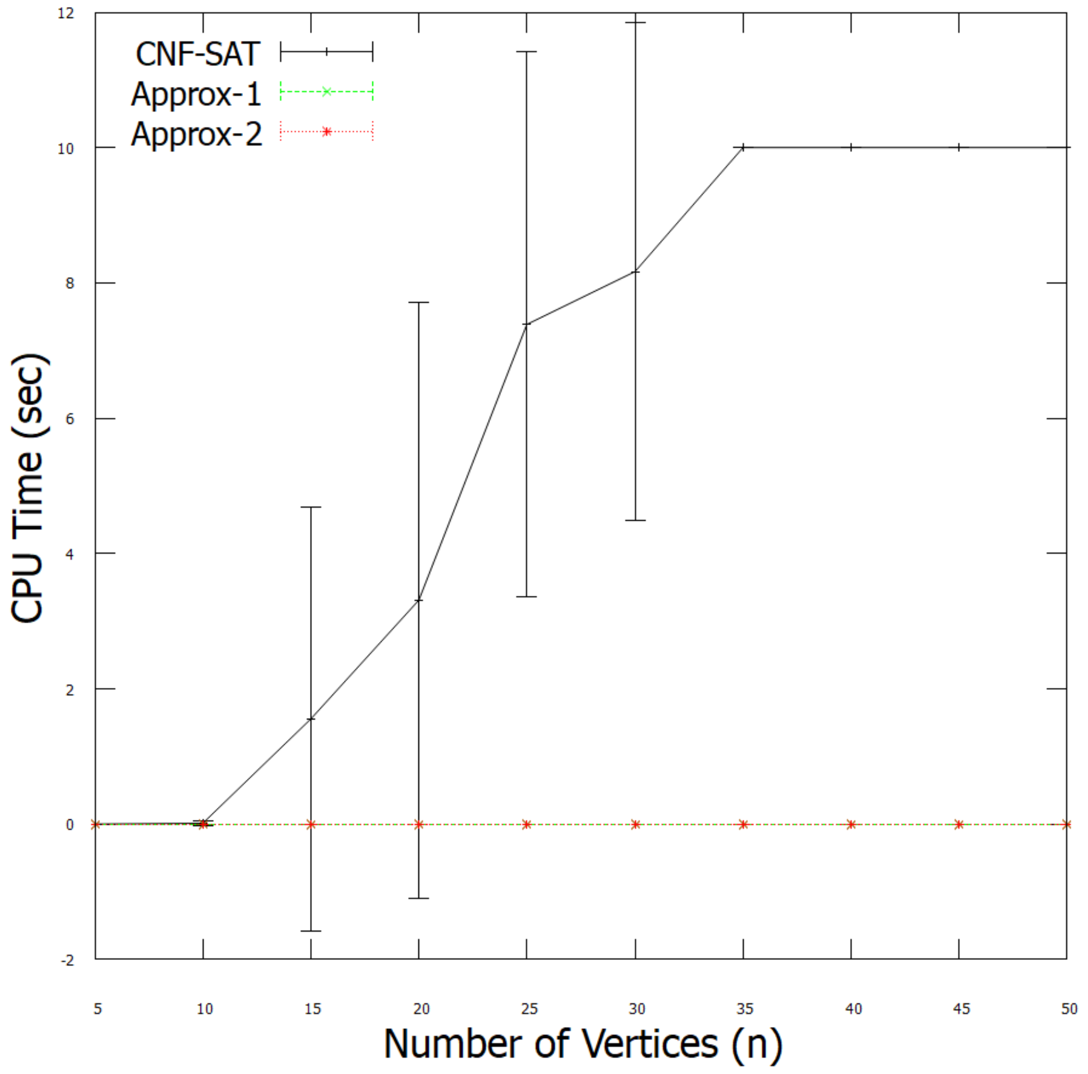

Fig. 3. Graph of 'Running (CPU) Time (sec)' v/s 'Number of Vertices (n)', for each algorithm being considered and using Optimized Encoding in CNF-SAT-VC

On comparing Figure 1 with Figure 3, it's quite evident that the Optimized Encoding does scale to larger instances. The original encoding could not produce any output for graphs with $n \geq 20$, when timeout was kept to 10 sec. But the limit is now raised to 'n = 35', for the same time-out.

As expected, we see that the CPU Time shoots up, for graphs with $n > 10$. But the rise is steeper, when compared with the original encoding. At v=15, we get no timeouts; hence the only reason for the large standard deviation can be attributed to good amount of variations in the input data. For $n \geq 25$, the optimized version does timeout for a lot of runs; and hence the large standard deviation.

For $n \geq 35$, the optimized version times out for all runs, hence we see a consistent CPU time of 10 seconds (which is the timeout itself), with no standard deviation at all.

Due to the large sample size of test data, it was not practical to try this analysis with a large value of timeout. But we did try executing certain input graphs with $n \geq 20$ (using the original encoding) one by one, and none of them could produce an output within 30 mins. This gives us a good indication of how significant the improvement (in Optimized Encoding) is.

3.2 Approximation Ratio

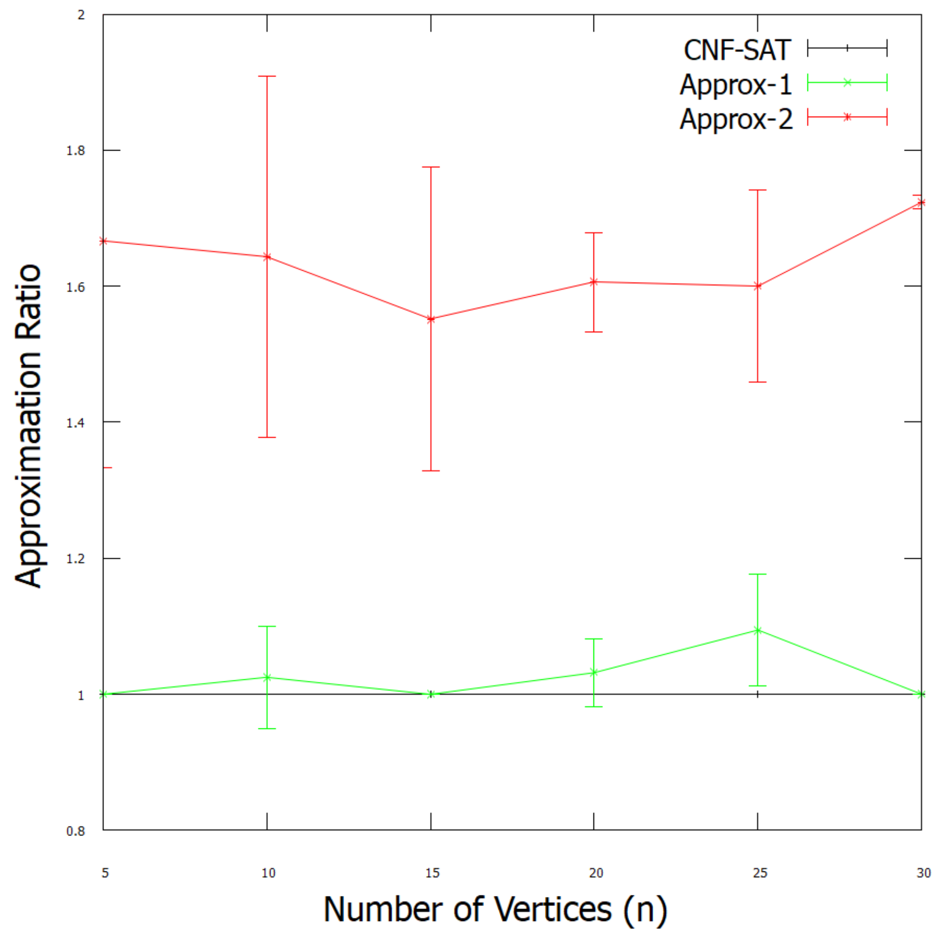

Fig. 4. Graph of 'Approximation Ratio' v/s 'Number of Vertices (n)', for each algorithm being considered and using Optimized Encoding in CNF-SAT-VC

Approximation Ratio is computed using the following formula: [2]

$$ApproximationRatio(Algo) = \frac{|VertexCover(Algo)|}{|OptimalVertexCover|} \quad (2)$$

Where: 'Algo' is the current algorithm being considered, 'VertexCover(Algo)' is the Vertex-Cover found using 'Algo' and 'OptimalVertexCover' is the Minimum possible Vertex Cover.

CNF-SAT-VC .

CNF-SAT-VC always produces the Minimum (Optimal) Vertex Cover, since it starts looking for a satisfiable 'k' beginning k=1, and returns it once found. Hence, the approximation ratio for CNF-SAT-VC, by above definition, will always be equal to 1, as is evident from the graph.

Due to this, the Standard Deviation, will always be Zero at each value of 'n'. This is proved by the absence of vertical bars at each coordinate (for CNF-SAT-VC) in the graph (Fig. 4).

Fig. 4 shows data only till $n = 30$. This is because, it's not possible to find Approximation-Ratio for $n > 30$, since CNF-SAT-VC is the only source of finding an optimal vertex-cover, but it times out for larger instances.

Also, note that the Mean Approximation Ratio that we see in the Graph are only for those inputs, that did not time-out. Those that time-out are not considered at all, which is also the reason for no data beyond n=30 (where all runs time-out).

APPROX-VC-1 and APPROX-VC-2 .

APPROX-VC-1 performs better than APPROX-VC-2 due the difference in selection of vertices by respective algorithms.

APPROX-VC-1 picks the highest-degree vertex in each iteration, covering maximum possible edges for that particular iteration [2]. There are chances that this may not give the most optimal solution, as it is always possible that the edges covered by the highest-degree vertex would have anyway been covered by other vertices, later added to the vertex-cover in subsequent iterations. Another case where it might fail to produce an optimal solution is when the algorithm leads to a combination of vertices, each of which covers at least one unique edge by itself, but still a different combination (with a smaller size) is sufficient to cover all edges. Yet, the number of redundant vertices (in Vertex-Cover) are highly unlikely to be greater than those produced by APPROX-VC-2.

APPROX-VC-2 picks the first available uncovered edge and simply adds its 'source' and 'destination' to the vertex-cover. It is very much possible that one or both of these vertices do not have any other incident edge(s). In any case, this algorithm will surely end up adding atleast one redundant vertex (i.e. the end-point of last uncovered edge).And since it is possible for our input graphs to contain many such edges, this can add a lot of redundant vertices to the vertex-cover, depending on the selection of edges.

The graph backs the above analysis. We see that APPROX-VC-1 performs better and even returns optimal solutions for all inputs at n=5 and n=10, as is evident by the absence of Standard Deviation at both the coordinates. A reasonable standard deviation at $n = \{15, 20, 25\}$ also means, there are chances that the algorithm could have produced an optimal solution for some graph inputs at these 'n'. Although we see an approximation ratio of '1' with no standard deviation at n=30, this need not necessarily be true if we had a larger timeout; as we are not considering timed-out runs (in Optimized CNF-SAT-VC) as discussed earlier (and there are high chances that majority of runs at n=30 could have timed-out).

APPROX-VC-2 on the other hand, is nowhere near to being optimal, as was expected. Also, the Standard Deviation is very large, which backs our claim of getting different number of redundant vertices (in vertex-cover), depending on the order of edges being selected.

Overall .

- From the above analysis, it's obvious that CNF-SAT-VC produces the most optimal solution.
- APPROX-VC-1 also performs reasonably well, even providing an optimal solution in a lot of cases. Overall, it is very close to optimal.
- APPROX-VC-2 is very unpredictable and will never produce an optimal solution. (Since it will have atleast one redundant vertex in the vertex-cover).

4 Conclusion

- If time-efficiency is the only requirement, than APPROX-VC-2 comes across as the most suitable option.
- When output-efficiency (optimal result) is the only requirement, CNF-SAT-VC (with optimized encoding) is the best fit, as it is guaranteed to produce minimum vertex-cover.

- In cases, where both time-efficiency and optimality are important , we can explore the following options:
  1] Use optimized CNF-SAT-VC first. If it times out (adjust the timeout beforehand, as per requirement), then use APPROX-VC-1.
  2] Use only APPROX-VC-1: Although it is not guaranteed to produce an optimal solution; it is safe to say that the solution will be very close to being optimal (on the basis of above analysis)

References

1. Author, Prof. Arie Gurfinkel.: uw-ece650-1221-a4.pdf (2022)
2. Author, Prof. Arie Gurfinkel.: uw-ece650-1221-prj.pdf (2022)
3. Owner, Prof. Arie Gurfinkel.: graphGen
4. GNU Plot Home Page, `http://gnuplot.sourceforge.net`.
5. Minisat Home Page, `http://minisat.se/`.