Project 3 Report

CSCI 338

Camille Custer & Avery Jacobson

Introduction

Detailed in this report are algorithms used to return vertex covers and independent sets of specified graphs. There is an algorithm to return the optimal (smallest) vertex cover, optimal (largest) independent set, and non-optimal VC and IS. We compared running times from tests on different graphs for each algorithm.

Overview of Algorithms

vcVerifier(`HashSet<Integer> set, Graph inputGraph`**)**

VcVerifier takes as input a set of vertices and a graph. Think of the vertices in the set as marked vertices. It first checks that the length of the set is smaller than the length of the graph. It will return false if it is not. If the size of the set is less than the size of the graph, we check each edge, e = (a, b), to see if either a or b is in the set (marked). For each vertice in the graph, if that vertice is not in the set (unmarked), we will check to see if its neighbors are in the set (marked). If a neighbor exists that is not in the set, return false because that shows there is an unmarked vertice connected to an unmarked vertice, which is not a vertex cover.

PrintPowerset(`int[] set, int n, Graph inputGraph`**)**

This function takes an array of vertices, integer n, and a graph as input. The initial function was taken from GeeksforGeeks.com and modified for our purposes. This function takes the array of vertices and returns every permutation of every possible length, starting with the smallest. In a for loop, each of these subsets is passed to our VcVerifier function that will return true if it is a valid VC, and false if not.

InexactVC(`Graph inputGraph`**)**

We start by creating integer k that is the length of the input test graph. A for loop is run from 0 up to k, creating the array of vertices. The inexact vertex cover is initialized as an empty ArrayList(). For each vertex in the vertice array, if the vertex is not in the vertex cover ArrayList, add all of its neighbors. Do this until there are no more vertices to check. We send the array to VcVerifier to confirm it is a VC.

NumVertices	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average Seconds
5	6.2E-05	2.8E-05	5.97E-05	0.0001	8.77E-05	0.0000675
18	6.5E-05	6.1E-05	4.69E-05	0.000101	0.000138	0.00008238
19	0.00015	0.00014	0.000109	0.000128	0.000126	0.00013108
20	0.00011	9.5E-05	7.09E-05	0.000121	0.000124	0.00010336
25	8.8E-05	8.2E-05	5.06E-05	8.81E-05	8.07E-05	0.00007798
450	0.00413	0.00602	0.002546	0.00425	0.00204	0.0037972
945	0.0031	0.00255	0.002806	0.00656	0.00498	0.0039988
1272	0.0051	0.00769	0.006185	0.007639	0.008633	0.00704968
1534	0.00522	0.00598	0.005969	0.006969	0.006822	0.0061936

InexactVC() ran in $O(n^3)$.

ExactVC(`Graph inputGraph`)

We start by creating integer k that is the length of the input test graph. A for loop is run from 0 up to k, creating the array of vertices. This array, integer k, and inputGraph are passed to the printPowerset function. The following table shows all running times with varying graphs:

NumVertices	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average Seconds
5	0.0169	0.00035	0.000329	0.000485	0.000416	0.00369616
18	0.0889	0.0532	0.0431	0.03921	0.05716	0.056314
19	0.0807	0.063	0.0517	0.05704	0.0465	0.059788
20	0.1205	0.1253	0.131	0.1214	0.1087	0.12138
25	2.1743	2.0209	2.1721	2.0081	1.935	2.06208

As NumVertices increases, Average Seconds of running time increases exponentially, $O(2^n)$.

OptimalIS(`Graph inputGraph`)

Because we know that IS is the complement of VC, we run exactVC on the inputGraph and return an array of everything that is *not* in VC and *is* in the inputGraph.

NumVertices	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average Seconds
5	0.016	0.00037	0.000419	0.000303	0.000444	0.00350698
18	0.09	0.0349	0.0583	0.03563	0.04796	0.053358
19	0.0591	0.0471	0.0511	0.0556	0.0434	0.05126
20	0.1331	0.1158	0.1165	0.12321	0.10816	0.119354
25	2.2654	2.3195	2.2811	2.0862	2.23044	2.236528

As NumVertices increases, Average Seconds of running time increases exponentially, $O(2^n)$.

InexactIS(`Graph inputGraph`)

Because we know that IS is the complement of VC, we run inexactVC on the inputGraph and return an array of everything that is *not* in VC and *is* in the inputGraph.

NumVertices	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Average Seconds
5	0.0001	0.00011	0.000115	0.000236	0.000197	0.00015202
18	0.00021	0.00017	0.000493	0.000251	0.000245	0.00027338
19	0.00015	0.00049	0.000164	0.000285	0.000328	0.00028296
20	0.00021	0.00015	0.000108	0.000416	0.000286	0.00023514
25	0.00014	0.00013	0.000139	0.000445	0.000295	0.00023068
450	0.00894	0.00828	0.002937	0.0092	0.00684	0.00723914
945	0.0051	0.00641	0.002818	0.00421	0.00371	0.00444934
1272	0.00648	0.00699	0.007836	0.00749	0.00624	0.0070087
1534	0.00954	0.01037	0.007277	0.00744	0.0132	0.0095654

InexactIS() ran in $O(n^5)$. (InexactVC plus n^2).

Validity

Both our inexact and optimal VC and IS are valid because we programmed and ran a verifier to check them for us. We know that our exactVC function returns the smallest (optimal) VC because it is programmed to do so. The permutations are ordered and verified by smallest to largest cardinality. As soon as a VC is found, it is guaranteed to be the smallest one, and powerset creation is halted. On the flipside, this is the same reason that we know optimalIS is the largest, as it is the complement of the smallest possible VC.

Our estimated VC and IS (using the non-optimal algorithms) were within reason of the optimal VC and IS. See the tables below to compare.

NumVertices	Inexact Vertex Cover Size	Optimal Vertex Cover Size	Accuracy (Expected-Observed)/Expected * 100
5	3	3	0%
18	10	9	10%
19	11	9	18%
20	12	10	17%
25	13	11	15%
450	428	420	2%
945	912	900	1%
1272	1236	1219	1%
1534	1493	1475	1%

NumVertices	Inexact Independent Set Size	Optimal Independent Set Size	Accuracy (Expected-Observed)/Expected * 100
5	2	2	0%
18	8	9	-13%
19	8	10	-25%
20	8	10	-25%
25	12	14	-17%
450	22	30	-36%
945	33	45	-36%
1272	36	53	-47%
1534	41	59	-44%

Testing

ExactVC() was tested with graphs of sizes 5, 18, 19, 20, and 25. These graphs are visualized at the end of this report.

This was done using a Driver file containing a main() function. Each algorithm was timed using System.nanoTime(), and **tested with 5 iterations** using a for loop for consistency and accuracy. We noticed that there was a relatively large jump in running time between graphs of 20 and 25 vertices. We can see the exponential running time in the plots below.

The non-optimal functions were tested with our **self-generated graphs** and the provided 450+ node graphs. We were not able to test the 450+ node graphs with our optimal functions as they run exponentially and would take too long. The largest graph we ran optimally was 30 nodes which took approximately 206 seconds.

We generated graphs by hand. We did this so we could predict the optimal VC and IS and cross-check with our computed outputs. Graph 18 and 19 are the same besides an additional node in the 19 node graph. This node was purposely placed to not change the vertex cover, which it did not. We then added a node we knew would change the optimal VC in the 20 node graph, which resulted in a larger VC.

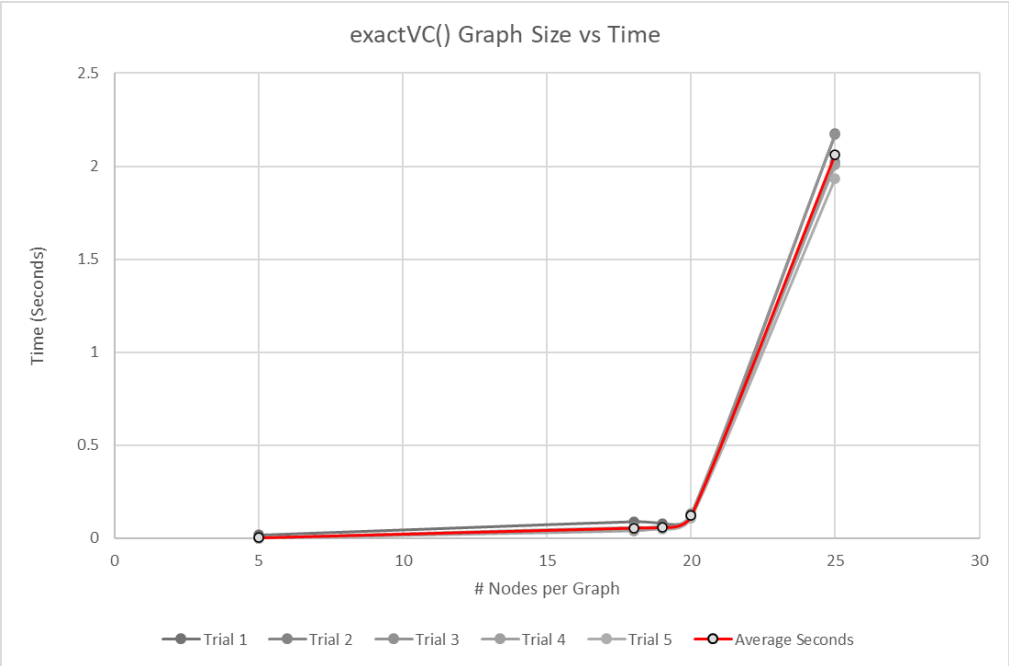
The 25 node graph was entirely different from the rest of the graphs and we used it to show the exponential time increase.

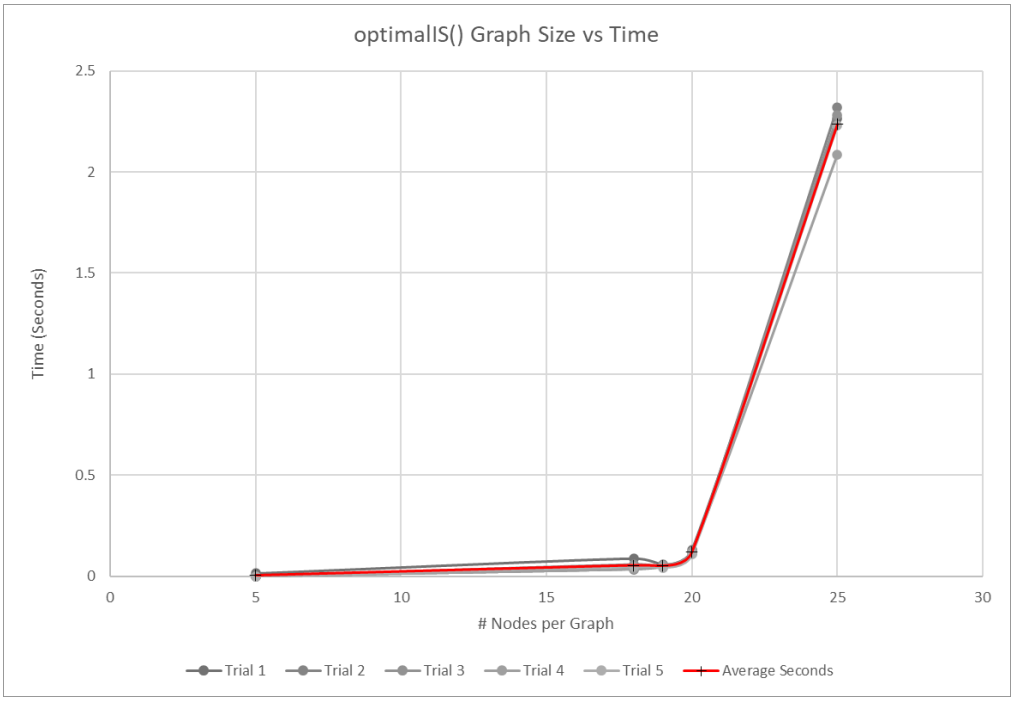
Conclusion

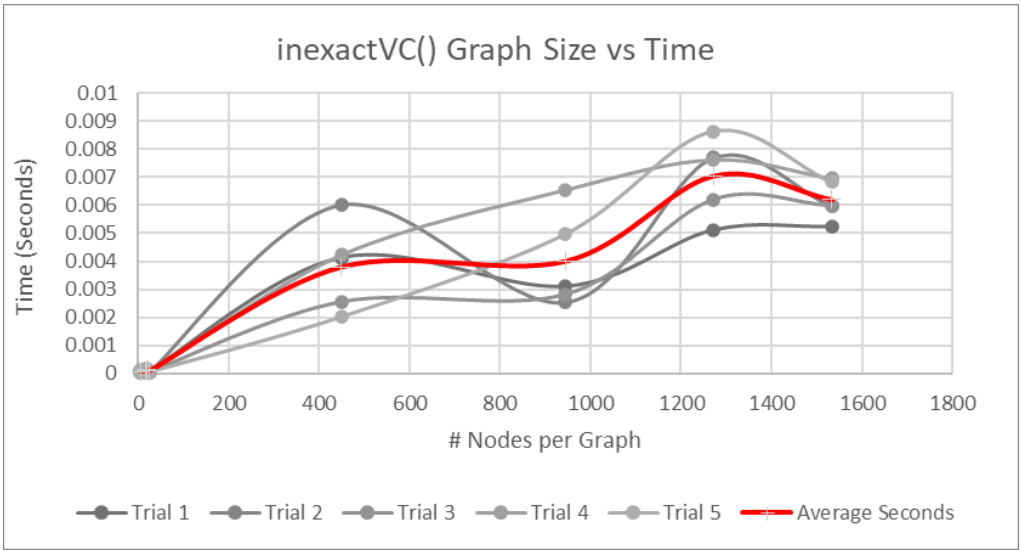
The inexact algorithms both function polynomially and quickly on graphs of large size. The largest graph we tested polynomially was 1534 vertices which evaluated in less than one hundredth of a second (0.0096 Seconds). Since we can build an Estimated Vertex Cover algorithm polynomially, inexactVC and inexactIS are in P.

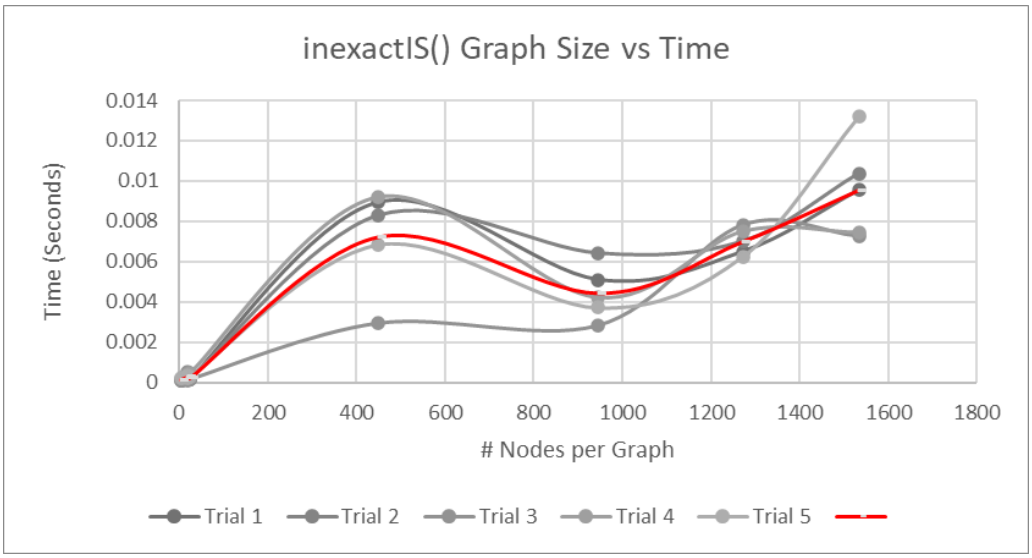
The ideal graph size for the exponential exactVC algorithm is less than 30 nodes. Finding Optimal Vertex Cover solves exponentially since we build the powerset of the vertices of size $O(2^n)$, and then polynomially check each set using a verifier. Because we can verify the Vertex Cover polynomially using a candidate solution (subset of powerset), exactVC and optimalIS are in NP. However, exactVC and optimalIS cannot be solved polynomially. Therefore, P/=NP.

Plots

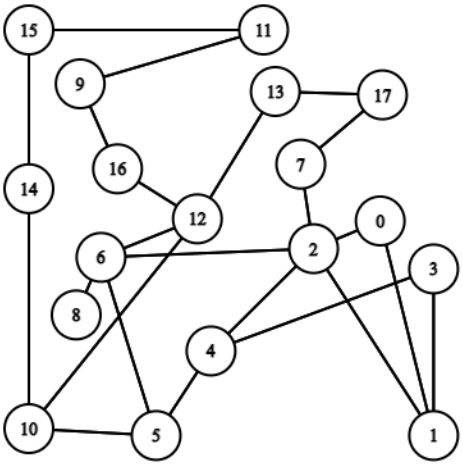

exactVC() Graph Size vs Time
Time (Seconds)
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
Nodes per Graph
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average Seconds


optimalIS() Graph Size vs Time
Time (Seconds)
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
Nodes per Graph
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average Seconds

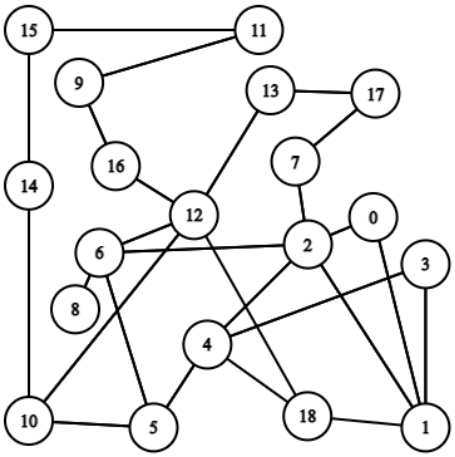

inexactVC() Graph Size vs Time
Time (Seconds)
0.01
0.009
0.008
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0
0
200
400
600
800
1000
1200
1400
1600
1800
Nodes per Graph
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5
Average Seconds


inexactIS() Graph Size vs Time
Time (Seconds)
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
200
400
600
800
1000
1200
1400
1600
1800
Nodes per Graph
Trial 1
Trial 2
Trial 3
Trial 4
Trial 5

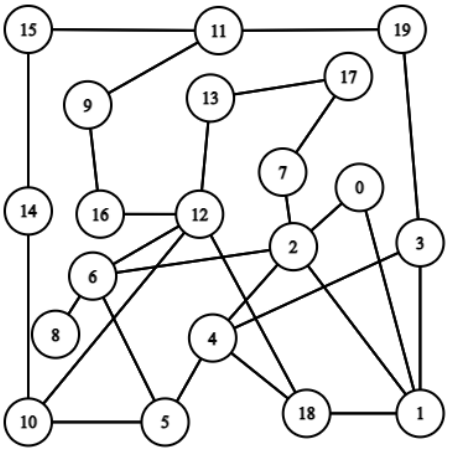
Self-Generated Testing Graphs



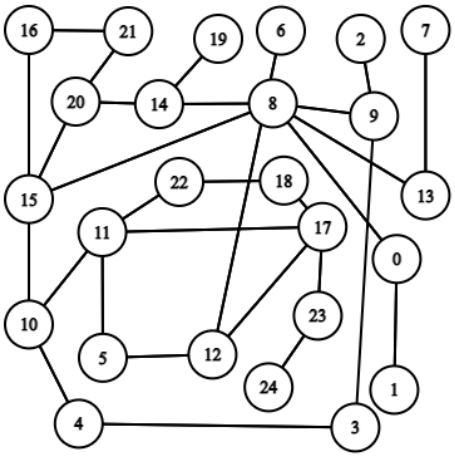
18 node graph



19 node graph



20 node graph



25 node graph