Burakhan CELENK

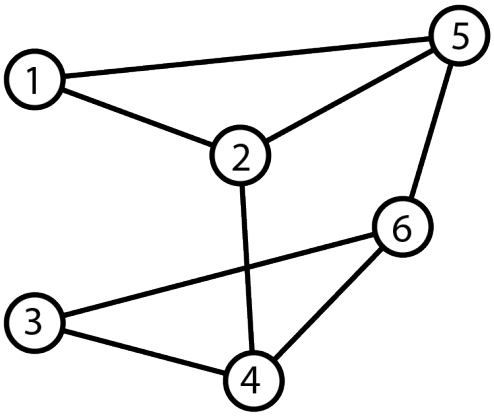
Study on Vertex Cover Problem

1. Introduction

What is vertex cover ?

A vertex cover is a subset of vertices belongs to an undirected graph such that for every edge(u,v) of the graph, either ‘u’ or ‘v’ is in the vertex cover. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. Here is an example in the following illustration.

Vertex cover problem is one of the NP-Complete problems. And there is no polynomial time solution for this problem. There are two type of this problem. The first one is, find a solution for a weighted graph. In this type, instead of counting vertices, to find the minimum cost vertex cover, we are calculating the sum of vertex weights and picking the minimum cost vertex cover. The second one is, find a solution for unweighted graph in which each vertex has same importance for the vertex cover. We call this type as **Cardinality Vertex Cover Problem**. In this report, we will only deal with cardinality vertex cover problem.



Minimum vertex covers

VC(1) = {2,4,5,6} VC(2) = {1,2,4,6}
VC(3) = {1,4,5,6} ...

There are some polynomial time approximate algorithms and branch-and-bound algorithms for solving this problem.

To make the report easy to read, here is the quick summary about what we will cover in the following sections.

Outlines

- Approximate algorithms for vertex cover problem
  - 1 - Algorithme Couplage
  - 2 - Algorithme Glouton
  - 3 - Custom approximate algorithm
  - 4 - Comparison of these algorithms

- Branch-and-bound algorithm for vertex cover problem
  - 1 - Branch-and-bound (BFS Approach)
  - 2 - Branch-and-bound (DFS Approach)
  - 3 - Improvements on branch-and-bound method
  - 4 - Comparison and evaluation of the improvements

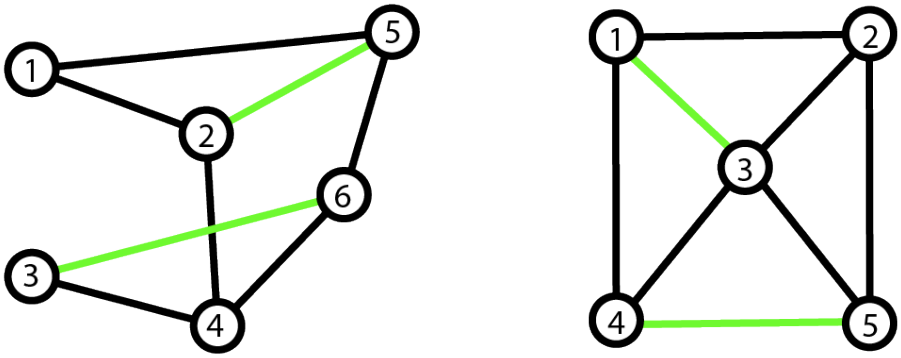
- General conclusion

2. Approximate Algorithms

In this section, we will only cover 3 algorithms and try to rediscover the quality of results and performance of these algorithms.

Algorithme Couplage

Before explaining this algorithm, let us give some definitions about the graphs. Given a graph G, A maximal matching is a matching M of a graph G that is not a subset of any other matching. A matching M of a graph G is maximal if every edge in G has a non-empty intersection with at least one edge in M. Here are some examples of maximal matching;



This is exactly what this algorithm does for finding an approximated minimum vertex cover. This algorithm is a 2 factor approximate algorithm which means, the cost of this algorithm for finding a vertex cover ≤ 2 * IOPTI.

Time Complexity : O(IVI + IEI)

Space Complexity : O(2 * IVCI), VC is optimal vertex cover.

Algorithme Glouton

The simple logic of this algorithm is briefly, find maximum degree vertex, delete it from the graph and add it to the vertex cover until there is no edge left in the graph. This algorithm ensures that returned vertex cover is valid as in the previous algorithm.

Time Complexity : O(IVI * IEI)
Space Complexity : O(IVI + IEI)

Custom Approximate Algorithm

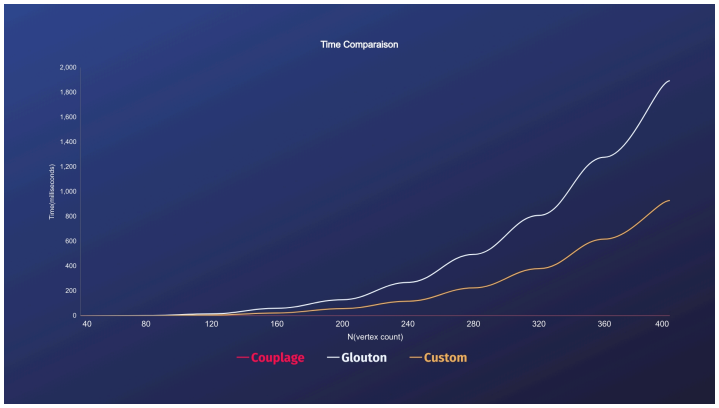
The simple logic of this algorithm is briefly, find maximum degree vertex and find an edge which doesn't contain previous vertex. Then find bigger degree vertex in this edge and delete these two vertex from the graph also add them to the vertex cover. If edge count = 0 then return vertex cover, otherwise return back to first step. This algorithm is also ensures that returned vertex cover is valid as in the Glouton and Couplage algorithm.

Time Complexity : O((|E| * |V|) ÷ 2)
Space Complexity : O(|V| + |E|)

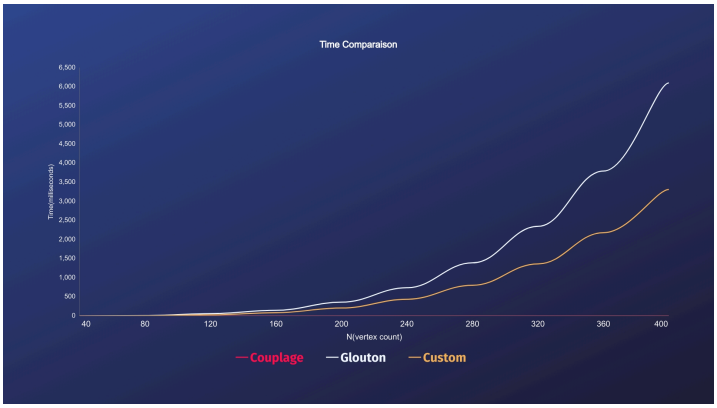
Comparison of Approximate Algorithms

We have always spoken theoretically, now it is time to look at the practical results. We will compare these 3 algorithms in terms of execution time and quality of returned vertex cover. Let us start with execution time.

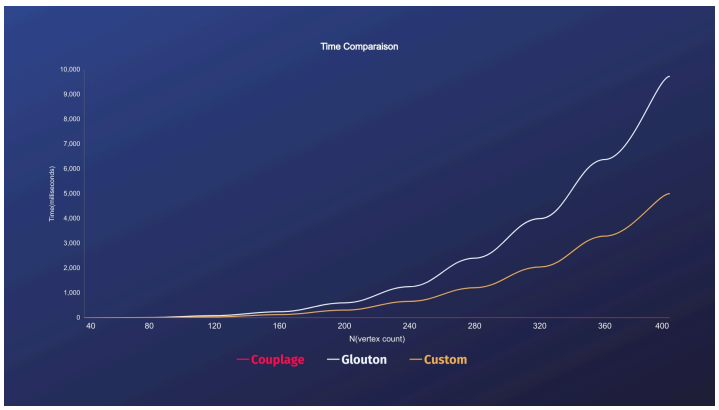
Execution Time Comparison



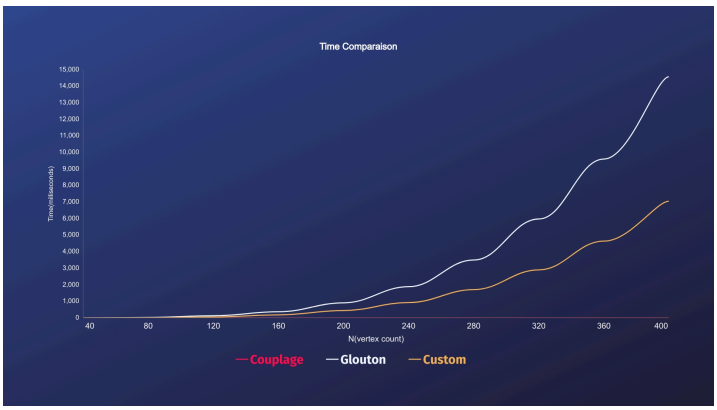
p = 0.1



p = 0.3



p = 0.5



p = 0.7

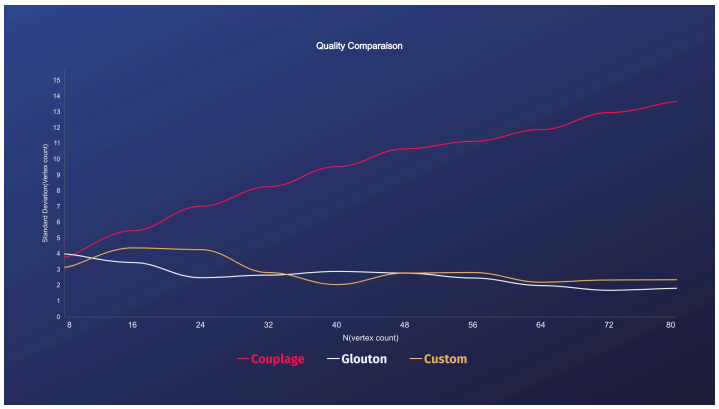
We have 4 different line graphs telling information about the execution time of these algorithms. Each line graph has different p value which is probability of edge existence between vertices(independent probability for every possible vertex couple).

According to the information in the line graphs, Couplage algorithm has a big advantage comparing to Glouton and Custom algorithms. While Custom and Glouton algorithms were reaching around 2-3 seconds, Couplage algorithm had not even reach 2 milliseconds yet. So it is obvious that Couplage algorithm is doubtlessly more and more faster than other two algorithms. If we compare Custom and Glouton algorithms between each other, we can clearly see the execution time relation between these algorithms which is generally Custom algorithm is twice faster than Glouton algorithm.

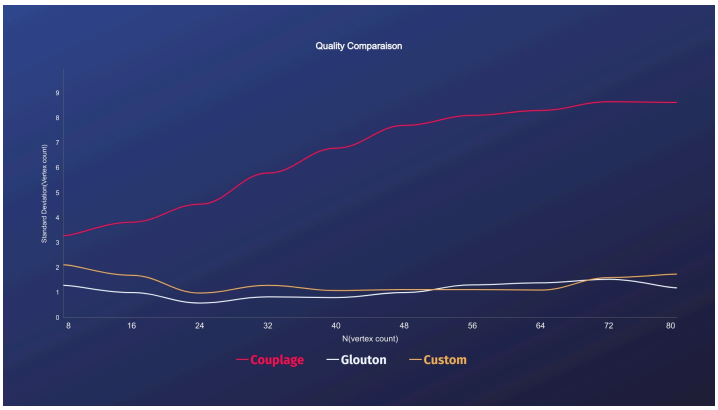
The relation between p value and execution time is different for these algorithms. For Couplage algorithm, increasing p value doesn't have a big impact on execution time. But for Glouton and Custom algorithms, while p value was 0.1, the execution time was around 1-2 seconds. While p value was increased to 0.7 from 0.1, the execution time has followed the same rate and reached to around 7-14 seconds.

As a conclusion, for Couplage algorithm, we don't have to worry about either vertex count or edge density. It will do its job quite fast. But unfortunately we should consider both vertex count and edge density if we are planning to use either Glouton or Custom algorithm.

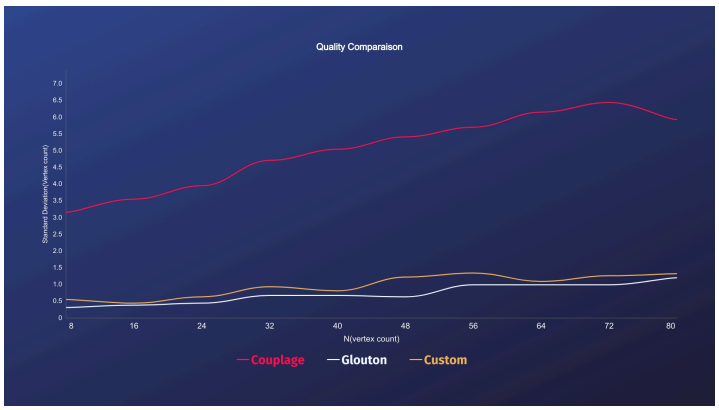
Comparison About Quality of Results



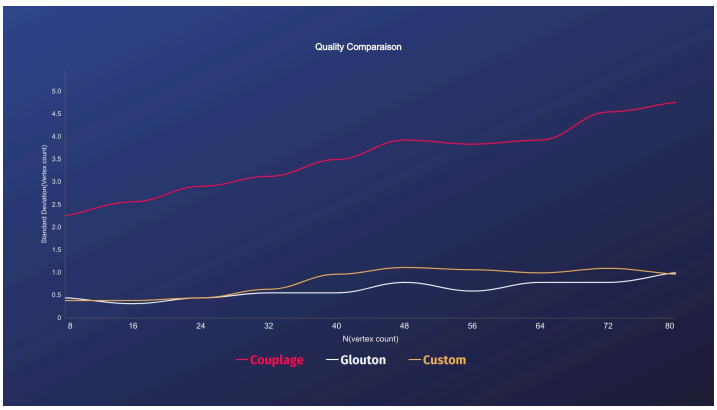
p = 0.1



p = 0.3



p = 0.5



p = 0.7

We will try to conclude on quality of result by considering standard deviation(given in the line graphs), percentage of correct results and worst results given by these algorithms.

As we can easily see from the line graphs, for Couplage algorithm, standard deviation increases while number of vertices in the graph increases. For Custom and Glouton algorithms, number of vertices in the graph doesn't have a significant impact on the standard deviation. it stabilizes itself around some fixed standart deviation value. If we examine p value, it is obvious that the increment of p value has a good effect on the standard deviation for all algorithms.

According to these examinations, we should be careful for Couplage algorithm if we have some graphs with big number of vertices. And lastly, for very small p values, all algorithms are in danger. One small suggestion on this side effect may be eliminate degree 1 vertices from the graph before execute the one of these algorithms (we will explain how it works in the further section).

Percentages of correct result given by algorithms ;

	p = 0.1	p = 0.3	p = 0.5	p = 0.7
Couplage	% 0-5	% 0-5	% 0-5	% 0-5
Custom	% 10-15	% 30-35	% 40-45	% 50-60
Glouton	% 5-15	% 40-50	% 50-60	% 60-70

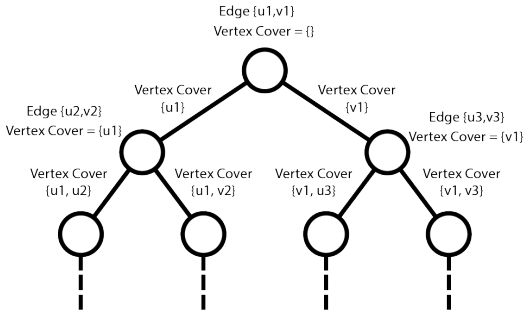
Worst results given by algorithms (vertex cover lengths) ;

	n = 10	n = 20	n = 30	n = 40
Couplage / Exact	10 / 5	20 / 11	28 / 16	38 / 24
Custom / Exact	5 / 3	5 / 2	13 / 9	13 / 9
Glouton / Exact	4 / 2	6 / 3	6 / 3	6 / 3

As a general conclusion, according to 3 different aspects on quality, we can say that the best results are given by Glouton algorithm. Custom algorithm follows Glouton with small difference and Couplage algorithm is obviously giving the worst quality results with huge difference comparing to other two algorithms.

3. Branch-And-Bound Algorithm

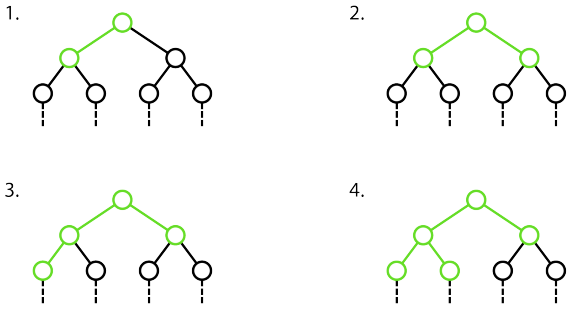
Let us consider a vertex in a graph. There are 2 possible case for a vertex in a vertex cover. It is either in the vertex cover or not. We can use this simple approach to transform the vertex cover problem to a binary search tree as shown in the following illustration. If we design the tree in this way until each path reach to a position such that there is no edge left to pick. One of these paths must contain the optimal vertex cover because we examine every possible combinations.



There are many ways to traverse a tree but in this section, we will review only 2 different approach which are Breadth-first-search(BFS) and Depth-first-search(DFS) but we will spend most of our time on DFS approach.

BFS Approach

The BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the tree layerwise thus exploring the neighbour nodes(nodes which are directly connected to source node). You must then move towards the next level neighbour nodes.



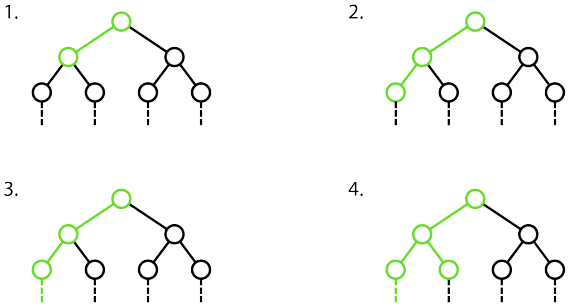
While traversing the tree, after the first encounter with the no-edge-left situation, there is no need to continue traversing anymore because after find a solution using BFS, we can only find a vertex cover with same length or bigger length if we continue to traverse.

Time Complexity : O(2^k) , k is length of the minimum vertex cover.
Space Complexity : O($2^{k-1} * k$), k is the length of the minimum vertex cover.

We will not cover the practical results of BFS approach but it is implemented in the source code and ready to test.

DFS Approach

The DFS is a recursive algorithm that uses the idea of backtracking. it involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes



along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

Opposite to BFS, we don't stop the execution when we first encounter with the no-edge-left situation. We traverse all the tree and pick the minimum vertex cover from the found solutions.

Time Complexity : O(2^{IVI})
Space Complexity : O(IVI)

As you can see easily, it is a terrible algorithm for vertex cover problem if we don't improve it. But the good news, it is very convenient for improvements. Now let us see how we can improve this algorithm.

Adding Minimum Bound

Let G be a graph, M be a maximal of G and VC be a vertex cover of G. Then,

if IVCI ≥ max(b_1, b_2, b_3), stop branching relevant node. With,

$$b_1 = \left\lceil \frac{m}{d} \right\rceil \qquad b_2 = IMI \qquad b_3 = \frac{2n - 1 - \sqrt{(2n-1)^2 - 8m}}{2}$$

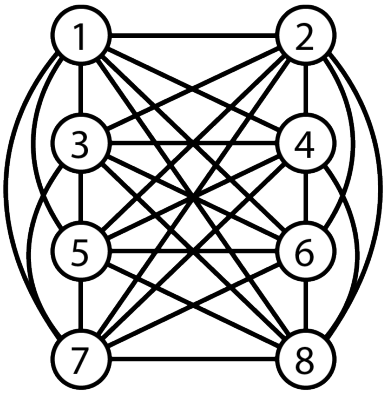
Where n is the number of vertices, m is the number of edges, d is maximum degree, M is maximal of the graph.

Proof of Bounds ;

Let us consider following graph ;

if you calculate b_1 for this particular instance, you will see that the result is equals to minimum vertex cover length of this instance. To prove this, let us compare b_1 and minimum vertex cover without specifying the number of vertices. d value for this type of graph is fixed to 2 and also m value is equal to $n - 1$. The length of the vertex cover is equal to $\left\lceil \frac{n-1}{2} \right\rceil$ and b_1 is equal to $\left\lceil \frac{m}{d} \right\rceil$ which is $\left\lceil \frac{n-1}{2} \right\rceil$. Thus no matter how much is n value, b_1 will always predict the minimum vertex cover length correctly. Now let us examine b_3. If you put $n - 1$ for m in the equation and solve it, you will see that no matter how much is n value, b_3 will always be equal to 1 which is absolutely false unless n is equals to 1 or 2.

Now let us consider following graph ;



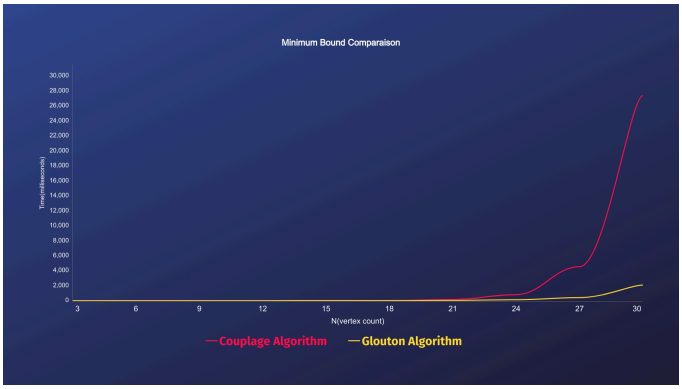
This is a special graph such that all possible vertex pairs are connected to each other. if you calculate b_3 for this graph, you will realize that the square root term is always equals to 1. This is not an exception, you can test it by replacing m value with $\frac{n*(n-1)}{2}$. If we write the equation again, we will get $\frac{2n-1-1}{2}$ which is equal to $n-1$. Thus we have proved that b_3 is equal to minimum vertex cover length. If you calculate b_1 for this graph, you will get $\frac{n}{2}$ which is absolutely false unless n is equal to 2. Now let us focus on the difference between these two graphs. You can easily see that the only difference between them is the edge density. Hence, we can say that b_1 is good at predicting low density graphs and b_3 is good at predicting high density graphs. But they are both bad at predicting graphs with average density.

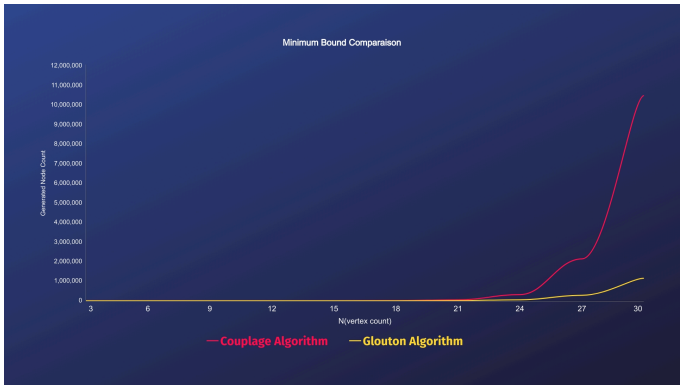
The easiest part is prove the validity of b_2. As we explained in the previous sections, the length of maximal of a graph is always bigger than or equals to the minimum vertex cover of the same graph. So b_2 will always give a valid bound for any edge density.

According to these informations, we can use b_2 alone as a bound but we can't use max(b_1, b_3). Because it may cause pruning the optimal solution for average density graphs. Besides, there is no need to use max(b_1, b_2, b_3) because if you calculate b_2 for low and high density graphs, you will see that b_2 will always give bigger value than b_1 or b_3. Instead of using 3 bounds in this manner, we can switch between the bounds according to the edge density. For example, if edge density smaller than some fixed value use b_1, if it is bigger than some another fixed value then use b_3, if it is between these 2 fixed value then use b_2 as a bound.

We said that we can use b_2 alone as a bound. But do we have to use maximal as a bound? Of course not! If you look at the quality comparison in the previous sections, you can see that Couplage algorithm have a bigger standard deviation than Glouton or Custom algorithm. Both Glouton and Custom guarantees that the result is bigger or equal to minimum vertex cover. Then we have still valid bound if we use Glouton or Custom algorithm as a bound. The following line graph shows us effect of using another algorithm as a bound.

Time Complexity : reduced to O(2^{IMI} or 2^{IGI}), IGI is length of vertex cover returned by the Glouton algorithm.

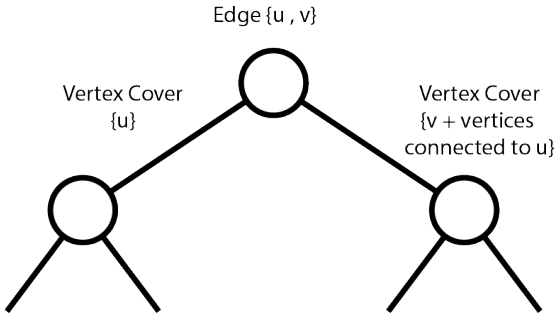




According to the line graphs, the results are expected just as we have discussed before. Because of Glouton has better quality result, b_2 bound has pruned more nodes, and the algorithm has finished its task quicker than the old version.

Advanced Branching

We know that there are 2 endpoints of an edge and the vertex cover must contain one of these 2 endpoints for cover this edge. Let us say that we have branched the tree for one of these endpoints and switched to other one. If there is an optimal solution which contains the first vertex, then the optimal solution must lay under that branch. Hence, there is no need to branch more combinations which contain any combination that contains first vertex. Since we will not take the first vertex, we will include the other vertices connected to the first vertex in the vertex cover to cover the relevant edges. Due to this reason, we will redesign the branching method illustrated in the following image.



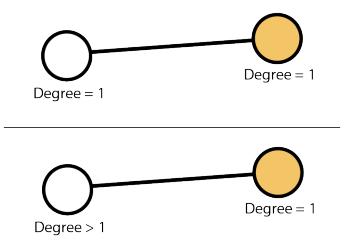
To add these vertices to the vertex cover, we can use may different ways. For example, add them in the ascending vertex degree order, descending vertex degree order or we can add them randomly. At each addition we are deleting edges connected to that vertex. Since our goal is to cover as many edges as possible. The smartest choice will be add them in descending vertex degree order(maximum degree first).

Time Complexity : reduced to O(IVI^2)

Elimination of Degree One Vertices

If there are degree one vertices in a graph, then there is an optimal vertex cover which doesn't contain these vertices. Let us prove this lemma,

For a degree one vertex, there are 2 possible cases as illustrated in the image. The first case is an exception(graph with 2 vertices) which can be handled by adding one line of code. Now let us focus on the second case,



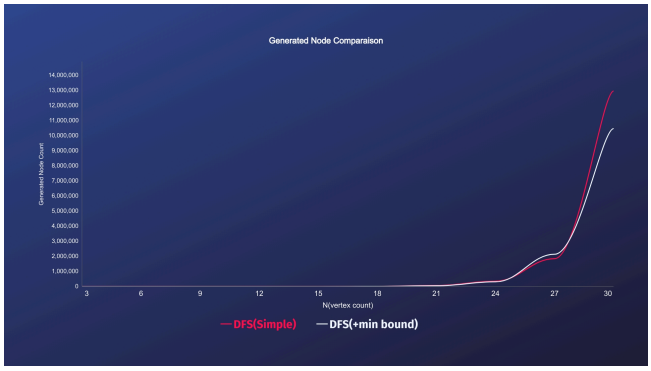
if we pick the degree one vertex, it is obvious that we have to pick another vertex or vertices for cover other edges connected to the vertex with degree > 1. To understand better, here are the possible vertex covers with including degree one vertex and without including degree one vertex ;

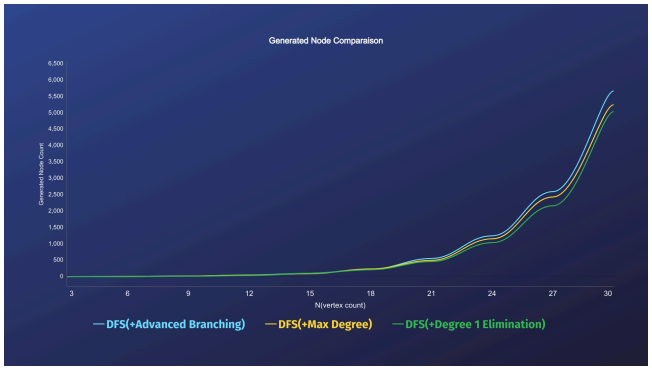
VC1 = { degree one vertex + all other vertices } length > 1
VC2 = { vertex with degree > 1 } length = 1

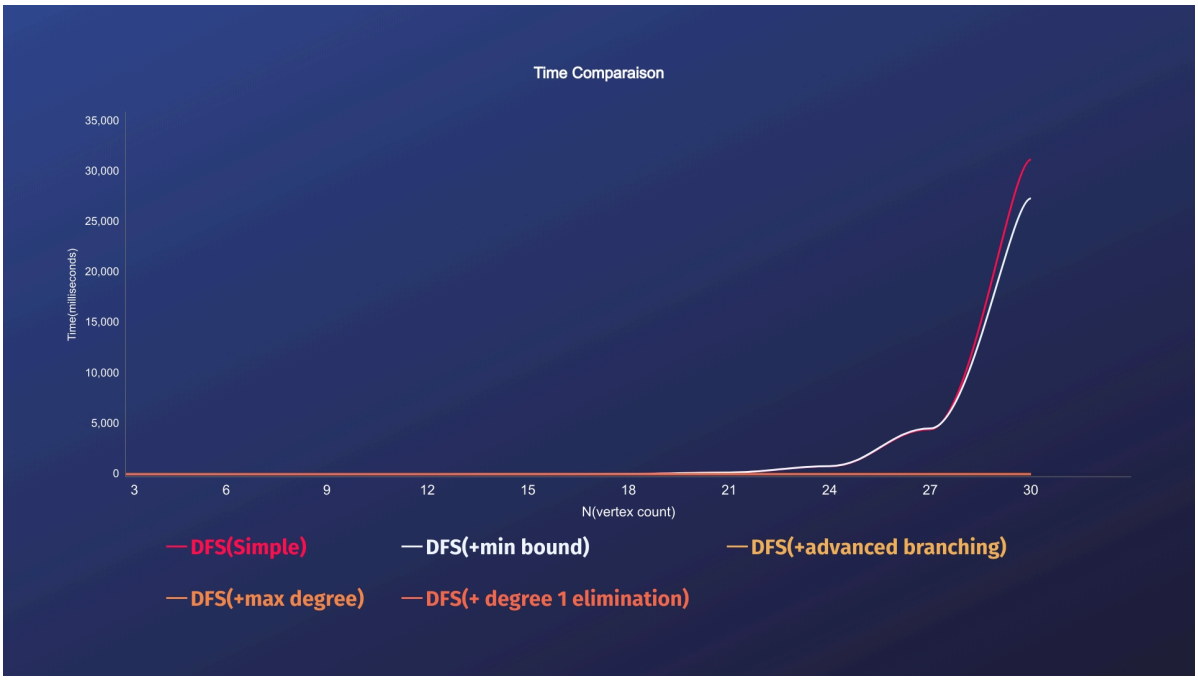
As our goal is find the smallest vertex cover, the vertex cover containing degree one vertex may be one of the optimal solution for some exceptional graphs(see the first graph in the bound proofs) but there will be always an optimal solution without the degree one vertex.

Practical Results of Improvements

We have talked about a lot of improvements. It is time to test them and see the effects of these improvements. Here are the results in terms of execution time and generated nodes (p = $\frac{1}{\sqrt{IVI}}$) ;







Generated nodes and execution time are highly related to each other. For this reason, we will examine all the line graphs together and make a final conclusion.

If we look at the generated nodes, we can say that advanced branching and adding minimum bound improvements have done most of the work and pruned unnecessary nodes. And that saved lots of time for us. After the these improvements, there is no big gain obtained by ordering vertices for advanced branching and elimination of degree one vertices but they are still improvements and they have a contribution on the performance gain even it is small comparing to first two improvements. As summary, we have reduced the generated nodes from around 13 millions nodes to 5000 nodes for a graph with 30 vertices with all these improvements which is great.

4. Final Conclusion

As a final conclusion, even if we have done a lot of improvements for Branch-And-Bound algorithm, It is still slow comparing to approximate algorithms. But no one talked about the obligation of picking only one algorithm for every possible case, and there is no super algorithm fits to it. So instead of go along with a final algorithm, we can switch between these algorithms according to our needs by looking following schema ;