Advanced Algorithms
1st Project - Exhaustive Search

Simão Teles Arrais - 85132
Departamento de Eletronica, Telecomunicações e Informática
Universidade de Aveiro

***Abstract* - The main objective of this article is to analyze the Independent Set of Size k Problem with different algorithms and for different sizes of the problem. Formal analysis of the computational complexity of two algorithms are made: an exhaustive approach and a heuristic based approximation. All is complemented with an experimental analysis based on their execution times and number of basic operations. There is also a helpful visualization of the given solution of the graphs under analysis.**

I. Introduction - Independendt Set of Size k

In graph theory, an independent set is a subset of vertices where no two of which are adjacent.

This paper aims to analyze and compare results of two different algorithmic approaches to solve the independent set of size k problem:

For a given undirected graph G(V, E), with n vertices and m edges, does G have an independent set with k vertices?

The independent set of size k is a graph optimization problem similar to finding the maximum independent set problem. Many graph optimization problems like these ones have significance for Computer Science and Software Engineers since it's a problem that has correspondence in real life.

II. Description of the Algorithms

A. Exhaustive Algorim

Exhaustive search is a very generic problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. Brute-force search is simple to implement and will always find a solution if it exists, implementation costs are proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases [1].

Algorithm 1: Exhaustive Independent Set of Size K

```
function Brute Force (adjacency_matrix):
  Input  : Dictionary representing a graph adjacency matrix
           adjacency_matrix
  Output: List of independent sets
if length_adjacency_matrix = 0 then
    return [];
if length_adjacency_matrix = 1 then
    return [single_vertice];
else
    final_result ← []
    for i ← 2 to total_number_of_vertices do
        all_combinations ← itertools.combinations
        temp_result ← all_combinations
        for every combination in all_combinations do
            for node1 in combination do
                for node2 in combination do
                    if node1 is connected to node2 then
                        temp_result.remove(combination)
                end
            end
        end
        final_result ← final_result + temp_result
    end
    return final_result
end
```

Figure 1: Pseudo Code of Exchaustive Search Algorithm

Firstly, the exhaustive search algorithm has two base cases, first one is when a graph is composed of 0 vertices , and it will return an empty list as a solution. The second base case is when a graph only has 1 vertice which means that will be the only existing set. When a graph has 2 or more vertices, it will calculate every combination of vertices possible using itertools library, loop through those combinations and check if all the nodes of a certain combination are valid for it to be an independent set. Upon finding every independent set that exists in the graph that was generated, it will return a list with all the solutions it got.

B. Greedy Algorim

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage. In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time [2].

Algorithm 1: Greedy Independent Set of Size K

```
function Greedy (adj_matrix, initial_vertice):
  Input  : Dictionary representing a graph adjacency matrix adj_matrix
           Initial vertice initial_vertice
  Output: Set containing the vertices that makes the independent set
visited ← []
 queue ← []
 solution ← {}
 add initial_vertice to visited
 add initial_vertice to queue
 while queue do
3  node ← queue.pop()
    if node has maximum number of edges possible then
4    node ← sorted_queue.pop
5  neighbours ← adjacency_matrix(node)
    not_neighbours ← difference(adj_matrix.keys, neighbours)
    not_neighbours.sorted()
    for vertice in not_neighbours_sorted do
6     if Vertice not Visited then
7       common_neighbours ←
          similarity(adjacency_matrix(node), not_neighbours_sorted)
        if common_neighbours = 0 then
8         solution.add(vertice)
          solution.add(node)
9       if common_neighbours = 1 and vertice not in solution then
10        solution.add(common_neighbours.pop())
11 end
12 end
```

Figure 2: Pseudo Code of Greedy Algorithm

For the Greedy algorithm, the idea is to consider the not adjacent vertices and pick the vertices that have the minimum number of edges, since they're better candidates for a possible solution. The search algorithm is similar to a *Breadth-First Search.*

Firstly, everything is initialized, then while there are candidate nodes to be visited, get all the vertices that are not adjacent to that node in a list, apply a heuristic that sorts them by the number of edges they have. For every not adjacent vertices, check if they have any edge in common with the solution set, if not they are part of the solution independent set.

III. Graph Generator

For this Graph Independent Set Problem, a custom graph generator was built in order to cover every topic it was asked. The generator takes into account the chosen number of vertices, maximum number of edges which is a percentage between 4 values (12,5%; 25%; 50% and 75%) for the number of vertices and a seed which in this case is the same as the Student ID value "85132".

All vertices have coordinates randomly calculated between 1 and 100, there are no vertices that coincide nor are too close to each other (*Euclidean Distance* needs to be above a certain threshold) and the number of edges that share a vertice is completely randomly determined. Everything that needs to be calculated for a given choosing is completely random down to the core

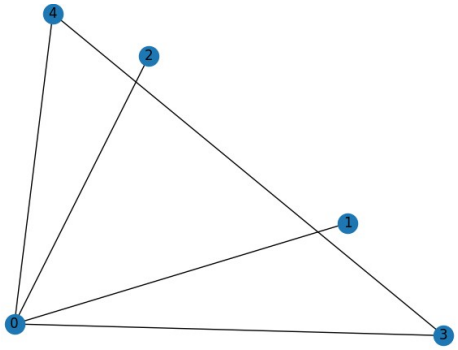

Figure 3: Graph with 5 vertices and 50% of maximum edge density

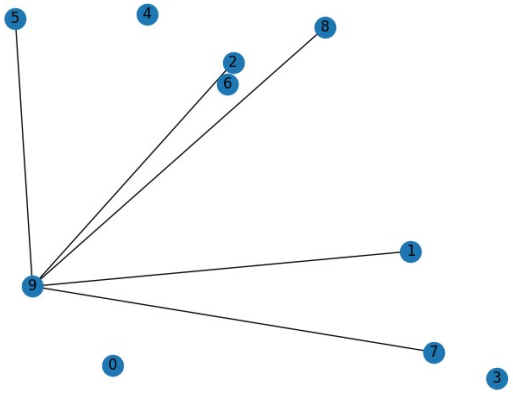

Figure 4: Graph with 10 vertices and 12.5% of maximum edge density

IV. Formal Computational Analysis

A. Exhaustive Algorithm

The computational complexity of iterating all the vertice combinations is the number of k combinations for all k, where k is the size of the independent set O(nCk), iterating through each combination of size k takes linear time O(k) and iterating every node of the combination is also linear time. Therefore, the computational complexity of this algorithm is $O(nCk * k^2)$.

B. Greedy Algorithm

The greedy search algorithm has a computational complexity of O(n) to find the vertices that are not adjacent to the current chosen node, O(k) for sorting the best vertice candidates and O(k) to iterate through every vertice that is a candidate. Therefore, the computational complexity of this algorithm is O(n + 2k).

V. Experiments

The Exhaustive Search with 18 vertices and 75% edge density already takes a total of 10 minutes approximately so for that reason, there will be no tests done to graphs more complex than that since it takes a lot of time to compute.

A. Execution Time

For the execution times, the exhaustive algorithm has a massive disadvantage compared to the greedy heuristic, specially in graphs where the edge density is bigger. The normal advantage of greedy is due to the fact that it doesn't need to generate and iterate over all vertice combinations and test every solution available.

The even bigger advantage on graphs with bigger edge density is due to the greedy strategy chosen which tries to pick the best not adjacent neighbours, since a lot more vertices will have a lot more edges, the iteration on not adjacent neighbours will be much faster.

n	Exhaustive Search				Exhaustive Search			
	12,5%	25%	50%	75%	12,5%	25%	50%	75%
2	1.2e-5	1.2e-5	1.2e-5	1.2e-5	1.8e-5	1.8e-5	1.8e-5	1.8e-5
3	1.33e-5	1.26e-5	1.33e-5	1.45e-5	2.19e-5	2.14e-5	2.07e-5	2.12e-5
4	1.81e-5	2.03e-5	2.12e-5	2.01e-5	2.83e-5	2.65e-5	2.31e-5	2.80e-5
5	3.45e-5	3.79e-5	3.88e-5	3.93e-5	3.24e-5	3.81e-5	3.88e-5	3.36e-5
6	7.03e-5	7.55e-5	7.36e-5	7.41e-5	3.71e-5	3.69e-5	3.52e-5	2.78e-5
7	1.68e-4	1.62e-4	1.76e-4	1.84e-4	4.26e-5	4.31e-5	4.17e-5	3.64e-5
8	4.33e-4	4.85e-4	5.41e-4	5.14e-4	6.52e-5	4.53e-5	3.92e-5	3.48e-5
9	1.83e-3	1.79e-3	2.17e-3	2.21e-3	5.62e-5	5.76e-5	5.81e-5	4.31e-5
10	5.06e-3	5.11e-3	5.42e-3	6.54e-3	6.03e-5	6.60e-5	5.65e-5	5.12e-5
11	1.10e-2	2.21e-2	2.16e-2	2.47e-2	7.39e-5	6.65e-5	6.74e-5	5.91e-5
12	5.55e-2	6.13e-2	7.04e-2	8.50e-2	7.92e-5	7.93e-5	7.60e-5	6.34e-5
13	0.21	0.23	0.30	0.35	8.56e-5	8.48e-5	8.27e-5	6.5e-5
14	0.58	0.83	1.12	1.58	9.01e-5	9.09e-5	8.61e-5	7.3e-5
15	2.22	3.33	4.33	6.42	9.25e-5	9.52e-5	8.65e-5	7.43e-5
16	17.54	19.92	24.32	27.23	1.02e-4	1.01e-4	9.20e-5	8.27e-5
17	53.12	75.48	102.73	116.18	1.21e-4	1.14e-4	1.00e-4	7.46e-5
18	137.58	315.58	492.34	641.11	1.23e-4	1.23e-4	1.17e-4	9.01e-5

Table 1: Comparison of Search Algorithms Execution Times in Seconds

B. Number of Solutions

For the number of solutions Exhaustive Search algorithm outputs, it's clear why it has such a big difference of total solutions between the numbers of vertices since it computes all possible combinations of vertices.

Even though the greedy algorithm only returns one solution, technically, each time a node is added to the solution, it counts as a new one. Since it's built to always choose the local optimum solution without backtracking, it will always have a big difference towards the Exhaustive Search, even in the final result.

It's clear that less edge density creates more solutions, since a low density helps in the generation of independent sets. We can compare both algorithms for n vertices and edge density by looking at Table 2.

n	Exhaustive Search				Exhaustive Search			
	12,5%	25%	50%	75%	12,5%	25%	50%	75%
2	1.2e-5	1.2e-5	1.2e-5	1.2e-5	1.8e-5	1.8e-5	1.8e-5	1.8e-5
3	1.33e-5	1.26e-5	1.33e-5	1.45e-5	2.19e-5	2.14e-5	2.07e-5	2.12e-5
4	1.81e-5	2.03e-5	2.12e-5	2.01e-5	2.83e-5	2.65e-5	2.31e-5	2.80e-5
5	3.45e-5	3.79e-5	3.88e-5	3.93e-5	3.24e-5	3.81e-5	3.88e-5	3.36e-5
6	7.03e-5	7.55e-5	7.36e-5	7.41e-5	3.71e-5	3.69e-5	3.52e-5	2.78e-5
7	1.68e-4	1.62e-4	1.76e-4	1.84e-4	4.26e-5	4.31e-5	4.17e-5	3.64e-5
8	4.33e-4	4.85e-4	5.41e-4	5.14e-4	6.52e-5	4.53e-5	3.92e-5	3.48e-5
9	1.83e-3	1.79e-3	2.17e-3	2.21e-3	5.62e-5	5.76e-5	5.81e-5	4.31e-5
10	5.06e-3	5.11e-3	5.42e-3	6.54e-3	6.03e-5	6.60e-5	5.65e-5	5.12e-5
11	1.10e-2	2.21e-2	2.16e-2	2.47e-2	7.39e-5	6.65e-5	6.74e-5	5.91e-5
12	5.55e-2	6.13e-2	7.04e-2	8.50e-2	7.92e-5	7.93e-5	7.60e-5	6.34e-5
13	0.21	0.23	0.30	0.35	8.56e-5	8.48e-5	8.27e-5	6.5e-5
14	0.58	0.83	1.12	1.58	9.01e-5	9.09e-5	8.61e-5	7.3e-5
15	2.22	3.33	4.33	6.42	9.25e-5	9.52e-5	8.65e-5	7.43e-5
16	17.54	19.92	24.32	27.23	1.02e-4	1.01e-4	9.20e-5	8.27e-5
17	53.12	75.48	102.73	116.18	1.21e-4	1.14e-4	1.00e-4	7.46e-5
18	137.58	315.58	492.34	641.11	1.23e-4	1.23e-4	1.17e-4	9.01e-5

Table 2: Comparison of Search Algorithms Tested Solutions

C. Basic Operations

Basic operations on both algorithms starts increasing as the number of vertices increases. Naturally, the increase is much more noticeable in the exhaustive search, since it does more iterations and more operations.

VI. Analysis and Time Comparison

The next figures 5, 6 and 7 are a plot of execution time of 50% edge density condition

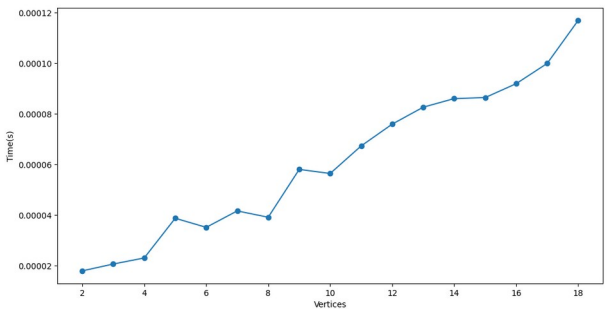

Figure 5: Greedy Search Time Plot

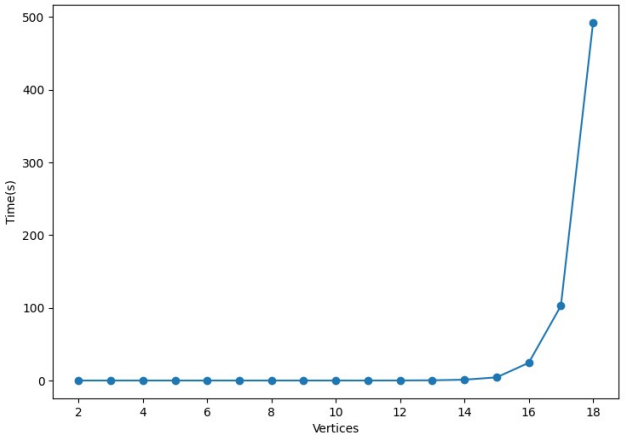

Figure 6: Exhaustive Search Time Plot

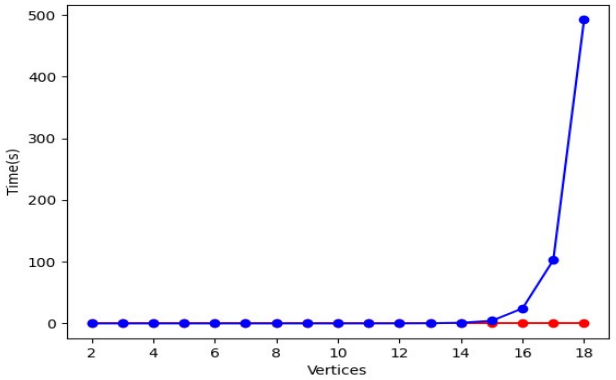

Figure 7: Comparison of both algorithms

By analyzing all figures 5, 6 and 7 we can really see the difference in execution time between both algorithms. At 12 vertices in a graph we can see that the algorithms start to grow apart from each other and at 16 or more vertices, the difference starts to be unfathomable.

VII. Conclusion

With this analysis, it can be concluded that both implementations have their pros and cons. The exhaustive search is a simple approach to find an optimal solution but only feasible for smaller computational problems because of it's exponential increase in execution time. The greedy solution is more suitable for larger problems or when an approximation for the optimal solution has more value than the optimal solution itself. For smaller computational problems, the difference is negligible.

Links

https://github.com/simaoarrais/Advanced-Algorithms

References

[1] "Brute-Force Search" https://en.wikipedia.org/wiki/Vertex_cover

[2] "Greedy-Algorithm" https://en.wikipedia.org/wiki/Maximal_independent_setz