Randomized Algorithms for Combinatorial Problems

Eduardo Santos, nºmec 93107, eduardosantoshf@ua.pt

***Abstract* – The objective of this assignment was to design and test a randomized algorithm to solve the combinatorial problem from the first assignment, which was to find a minimum weighted closure for a given vertex-weighted directed graph *G(V, E)*, with *n* vertices and *m* edges. This randomized algorithm was based on the Monte Carlo algorithms, and all the computations were made using a variety of parameters, which will be referred on the following sections.**

I. Introduction

A randomized algorithm is an algorithm that makes use of randomly generated numbers to make decisions. This randomness is used to reduce that algorithm's time or space complexity. Monte Carlo algorithms are randomized algorithms that have, specifically in this case, a chance of not producing a solution. Let's look at the following example:

> I want to find a '0' on a binary string but only compute 5 iterations, at most.

An example of a Monte Carlo algorithm to compute this problem, in pseudo-code, is:

Algorithm 1 Monte Carlo Algorithm Example

```
i = 0
do
    randomly select an array element
    i = i + 1
while i == 5 or '0' is found
```

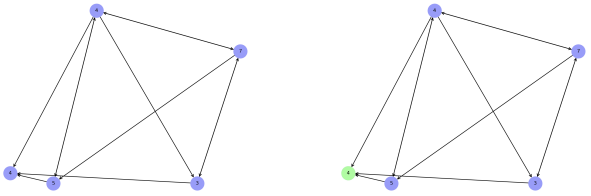
If an '0' is found, the algorithm succeeds, if not, it fails. This algorithm will be adapted to the context of the assignment's problem and will be explained in depth later.

II. Problem Description

As this assignment takes the previous one as a base, there is no need to explain again the context of finding a minimum weighted closure of a given directed graph, as it was explained previously. Given a randomly generated graph, or one read by a file, the computation of the minimum weighted closure can be very time-consuming, as the time complexity is given by:

$$O(2^N * N)$$

Thus, there was a need to reduce time complexity, this can be achieved with the implementation of a randomized algorithm. Let's consider a random graph with 5 nodes and 8 edges.



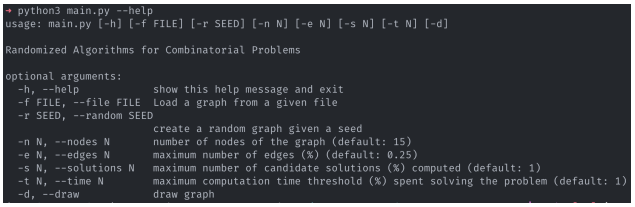
(a) Without Solution (b) With Solution

Fig. 1: Random Graph and its computed solution

Given this example, and using the **Exhaustive Search Algorithm**, we first need to compute all the subsets (closures) of the graph, which in this case are 22, and then find the minimum weighted closure, from all candidate solutions, and that can be, as previously mentioned, very inefficient. This can be optimized by passing some parameters that control the algorithm's output. These parameters as well as the algorithm itself will be explained in the following sections.

III. Implementation Description

Running the main program, there are a few flags that can be used.



```
→ python3 main.py --help
usage: main.py [-h] [-f FILE] [-r SEED] [-n N] [-e N] [-s N] [-t N] [-d]

Randomized Algorithms for Combinatorial Problems

optional arguments:
  -h, --help            show this help message and exit
  -f FILE, --file FILE  Load a graph from a given file
  -r SEED, --random SEED
                        create a random graph given a seed
  -n N, --nodes N       number of nodes of the graph (default: 15)
  -e N, --edges N       maximum number of edges (%) (default: 0.25)
  -s N, --solutions N   maximum number of candidate solutions (%) computed (default: 1)
  -t N, --time N        maximum computation time threshold (%) spent solving the problem (default: 1)
  -d, --draw            draw graph
```

Fig. 2: Help Menu of the Main Program

Most of the parameters are the same as in the previous assignment, the ones that are used as part of the randomized algorithm are:

- **-s** - this flag allows the user to define the maximum number of candidate solutions to test, its value belongs in [0, 1], with 1 representing all the possible candidate solutions (100%).
- **-i** - this flag allows the user to decide when to stop testing candidate solutions after spending a certain amount of computation time (in %).

A. Bug Fix

It is important to notice that there was a bug in the previous assignment's exhaustive algorithm. Instead

of only testing the subsets with edges that, in fact, existed on the graph. The algorithm computed all the combinations given the graphs' nodes and then found all the possible closures, for each combination, even if it was not present on the graph. This made the algorithm very inefficient.

On this assignment, the problem was fixed. Now, the algorithm computes only the subsets that truly exist on the given/generated graph, reducing significantly the time execution time, as well as the number of computations.

B. Randomized Algorithm

As previously mentioned, the randomized algorithm was based on the exhaustive search and on the Monte Carlo algorithms. Therefore, the algorithm decides when to stop based on a set of user-given parameters (flags), this allows for an increase in efficiency, making the solution less time and resources consuming.

As a non-deterministic algorithm, for the same input, there can be different results on each run, this gives an approximate solution, i.e., a solution that may not be the optimal one. This means that the solution computed may not be the minimum overall weighted closure, but instead, the relative minimum weighted closure of the computed candidate solutions.

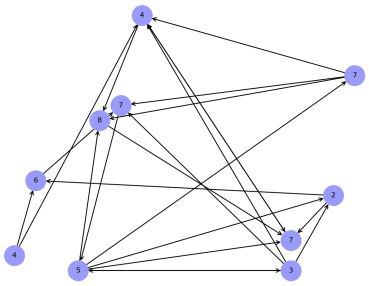
Firstly, the algorithm computes all the subsets, S, of the given graph. On the exhaustive search, every subset was considered a possible closure, but in this case, the *sample()* [2] function from the random module in Python was used to choose random subsets with the number of subsets being the maximum number of candidate solutions given by the **-s** flag. Then, for every subset, C, it is necessary to check if there are any edges for other nodes that are not in C. If this happens, then C is not a closure, if not, then the subset is added to the closures list. After having all closures, every subset node's weight is added, and the closure with the smallest sum is considered the minimum weighted closure.

Another strategy of the randomized algorithm is by using the **-t** flag. In this case, the algorithm stops computing possible closures when a certain execution time is reached, then computes the minimum weighted closure from the ones found.

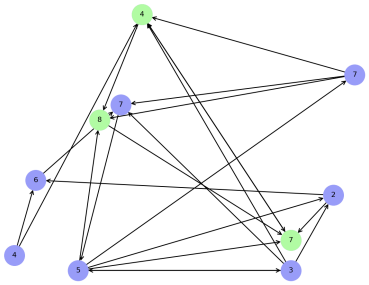
Both strategies can be used simultaneously, i.e., the algorithm can stop to choose random subsets with the number of subsets being the maximum number of candidate solutions and stopping when an also given execution time is reached.

B.1 Formal Computational Complexity Analysis

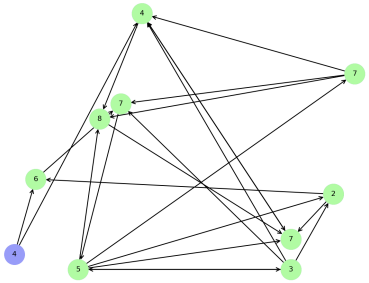
Performing a formal computational analysis of the exhaustive search algorithm, regarding the time complexity, the complexity would be $O(S * N * M)$, where S is the number of subsets, N is the number of nodes, and M is the maximum number of edges a node can have. This is because the first *for* loop iterates through the randomly chosen subsets, S, which is $O(s)$, the second *for* loop iterates through all the elements in each subset, which is $O(n)$, and the third *for* loop iterates through all the edges of each element in the subset, which is $O(m)$. Multiplying the time complexities of the three *for* loops gives the overall time complexity of the algorithm.



(a) Random Graph



(b) Minimum Weighted Closure Given By Exhaustive Search Algorithm



(c) Minimum Weighted Closure Given By Randomized Algorithm with **-s** = 0.5

Fig. 3: Comparison Between Exhaustive and Randomized Algorithms

In the images above there is a randomly generated graph with 11 nodes and 20% of the maximum number of possible edges between those nodes [3a]. For this graph, both exhaustive search [3b] and randomized [3c] algorithms' solutions were computed. Regarding the randomized method, the maximum number of candidate solutions given was 50% of the maximum number of possible solutions/subsets.

The solution given by the exhaustive search is the optimal one, which means that the closure returned by the algorithm is the minimum weighted one of the whole graph subsets. As the randomized algorithm only computes half of all the possible candidate solutions, the minimum weighted closure it retrieves as a

solution is, in this case, the entire graph. It is still a solution, but not the optimal one.

IV. RESULTS AND DISCUSSION

Several plots were drawn to better understand the difference between the randomized and the exhaustive algorithms and compare each other. Both algorithms were compared using three parameters: execution time, the number of solutions found, and the number of iterations. For each algorithm, the results were computed using successively larger random graphs with 1 to 25 nodes and 25% of the maximum number of possible edges between nodes. All of the results can be found in the **results/** folder.

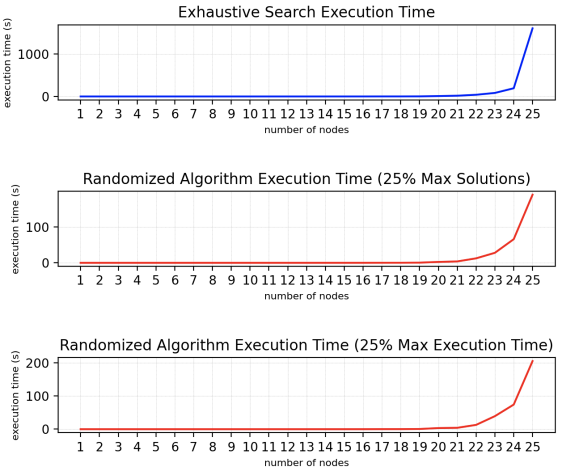


Fig. 4: Execution Time on the Exhaustive and Randomized Algorithms

Comparing the two algorithms, as seen in the plot [4], passing 25% of the maximum number of candidate solutions, as well as 25% of the maximum execution time, we get similar results between both variations of the randomized algorithm, both representing approximately 20-25% of the previous one's execution time, when comparing to the exhaustive search. This variation is directly related to the randomly chosen subsets for the computation.

As well as the execution time, there is also a reduction to 25% of the solutions number [5] of the exhaustive algorithm, both using the **-s** and **-i** parameters. It makes sense that when defining, f.e., 25% of the maximum number of candidate solutions, the number of solutions found is approximately 25% of the brute force solution's ones. But when passing the maximum computation time parameter, the similarity, although present, it's not as noticeable, in comparison to the **-s** flag.

Finally, comparing both algorithms regarding the iterations number [6], we can see that, using both the **-s** and **-t** parameters, the number of iterations was approximately half of the iterations of the exhaustive search. This can be the supposed results, but can also be a bug when computing the iterations of the algorithm.

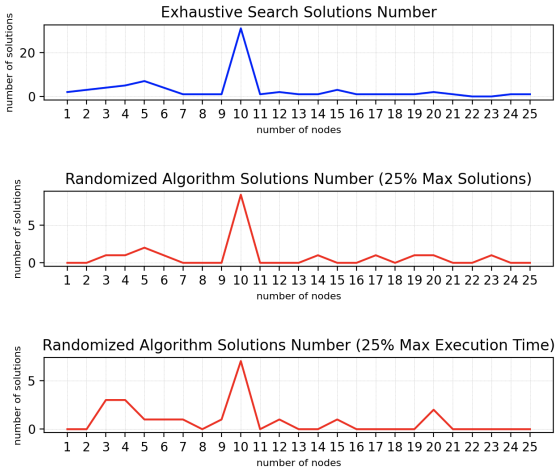


Fig. 5: Solutions Number on the Exhaustive and Randomized Algorithms

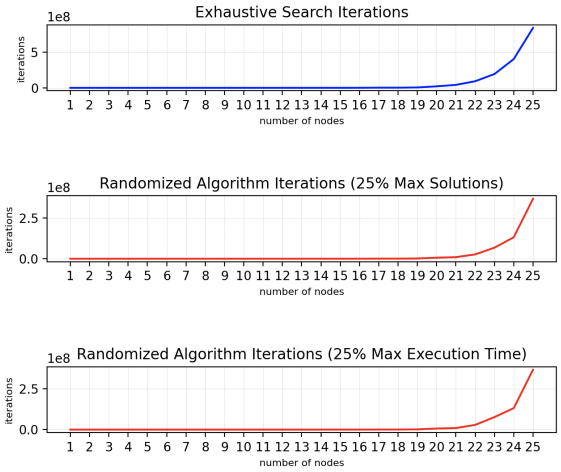


Fig. 6: Iterations Number on the Exhaustive and Randomized Algorithms

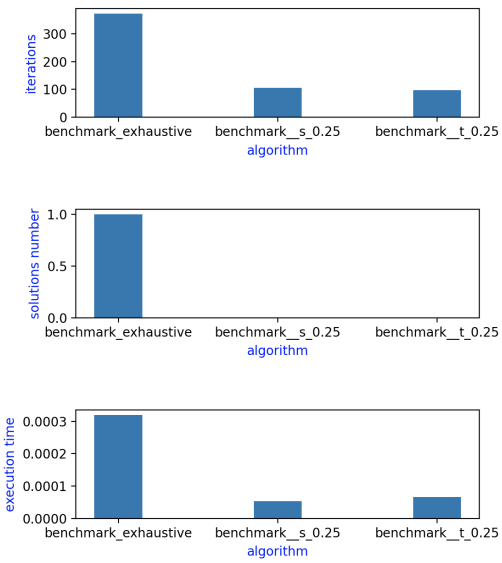


Fig. 7: Comparison between Exhaustive Search and Randomized Algorithms using the *SWtinyEWD.txt* file

Using the benchmark file to compute the above-mentioned parameters [7], the results are as expected. Due to the graphs' size being small, regarding the number of solutions found, we can see that, as probably the subset that contained all nodes was not computed, the number of solutions the randomized algorithm returned was zero.

V. Conclusion

This assignment allowed for a better understanding of a randomized algorithm and its variations. Although the randomized algorithm performed better than the exhaustive one, regarding the iterations numbers, as well as execution time, the algorithm does not always return a solution, and when it does, most of the time it is not the optimal one. The randomized algorithm may be seen as a mid-term between the brute force method and the greedy one, both compared to the previous assignment.

VI. Future Work

A feature that could be implemented is the ability of the algorithm to decide when to stop testing candidate solutions of a certain size and start testing larger or smaller solutions. This feature, despite not being implemented in this solution, can be done as a future work of the project.

References

[1] GeeksforGeeks. (2022, October 22). Randomized Algorithms. GeeksforGeeks. https://www.geeksforgeeks.org/randomized-algorithms/

[2] Python Software Foundation. (2001-2022). random — Generate pseudo-random numbers. https://docs.python.org/3/library/random.html