Maximum Weighted Closure Of A Directed Graph

Pedro Marques, 92926

***Resumo* – Este relatório apresenta a solução do autor ao denominado *closure problem.* Este consiste em encontrar o fecho com o máximo ou o mínimo peso de fecho de um determinado grafo direcionado. Já existem alguns algoritmos que podem ser utilizados para responder a este problema como o algoritmo de Ford-Fulkerson, no entanto, todas as soluções apresentadas foram feitas pelo autor do relatório. Para além disso, é realizada uma análise à complexidade de cada algoritmo e posteriormente comparado com a realidade. Neste trabalho, o foco foi em encontrar o fecho com maior peso possível, como fora sugerido no projeto.**

***Abstract* -This report follows the author's solution to the closure problem, using Python3 scripts. This consist on finding the maximum or minimum weighted closure for a given graph G. As such, some algorithms have already been created like Ford-Fulkerson but the solutions that are presented were made by the author of this report. Furthermore, more data is available about the complexity of each implemented algorithm as well as a comparison between expectations and reality. For this work, the author focused on searching for the maximum weighted closure as it was suggested in the given project.**

I. INTRODUCTION

With this work, the author expects to find a good solution to the closure problem for random generated graphs. As such, many scripts were created both to simulate and solve the problem. The *graph_generation.py* scripts contains the main function that calls the search algorithms as well as generates the random graphs. Each graph is saved in a different file inside the *graphs* folder. The *search.py* contains the exhaustive search algorithm and the *greedy_search.py* contains the greedy algorithm. Finally the *search_node.py* contains the Node class while the *visualization.py* contains some functions in order to generate statistical and figures. Files brt_results.csv and gr_results.csv contains all statistics from 10 experiment of the exhaustive and greedy algorithms respectively, including number of iterations, total weight, closure size,execution time and number of edges.

II. CLOSURE PROBLEM

Regarding a certain directed graph G, the closure of each of vertex V corresponds to the set of all reachable vertices of G from V, either directly or through some other vertex. The closure problem consists on finding either the minimum-weight or maximum-weight closure for a given directed graph.

Not much is available online about this problem, however there are some algorithms that can be used to solve this. It has been mentioned on many publications that this problem can be solved by using either a condensation or a reduction to maximum-flow algorithm.

A. *Condensation*

This algorithm relies mostly on brute force/exhaustive search by changing the input graph through condensation. Condensing the graph means replacing vertices with the same closure by a simpler represetantion, thus reducing the overall graph size and preventing repeating closures for similar vertices. Considering the graph G with vertices A,B and C represented in the following figure.

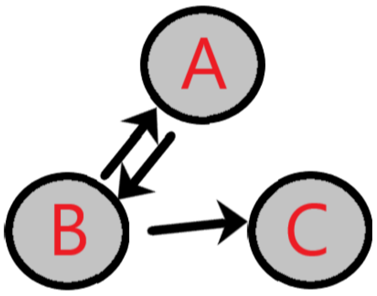


Figure 1 - Graph G with 3 Vertex

The vertices A and B create a cycle because there is a direct path that connects both to each other, as such the condensed graph would replace A and B by a third vertice AB and reduce the size of the first graph. The closure of the vertex AB would represent the condensation of both vertex A and B.

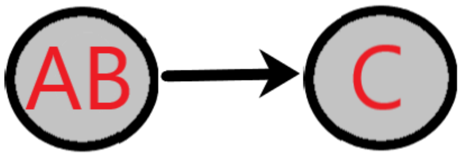


Figure 2 - Graph H obtained by condensing G

This reduces repetition in closure calculation and fewer iterations inside the graph, further reducing the execution time.

B. *Maximum Flow Algorithms*

A flow or transportation network is a directed graph on which each edge has a fixed maximum capacity capacity. The flow on each edge cannot surpass this capacity. Each flow network has a source (usually represented by S) from which comes all the flow, and the final destination of the network is called the sink (usually represented by a T).

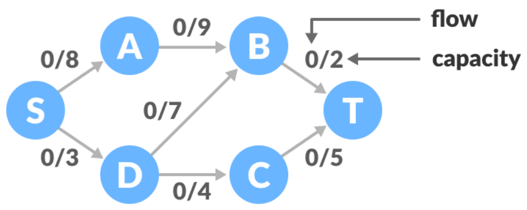


Figure 3 - Representation of a Flow Network

With this, we want to calculate the maximum amount of flow that can be sent from the source to the sink in an instance. The Ford-Fulkerson algorithm is a greedy algorithm used to solve this problem. It constantly adds flow to a network until there is not possible path between source and sink (also known as an augmenting path). The maximum flow will be the sum of all flows that where added progressively in this loop.

This algorithm can be applied to the closure problem if we add a source and a sink to the initial graph and set a rule that relates the vertex weight with the edge weight.

III. SOLUTION

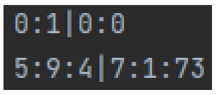
A. *Graph Generation*

The script graph_generation.py is responsible for generating random graphs. These graphs are directed, represented by the incidence matrix and each vertex has a random weight between 1 and 100. No vertex has an arc onto itself but the graph can be cyclic. Each vertex is a 2D point on the xOy plane, with x and y between 1 and 9, randomly given.

The incidence matrix is simply determined by generating an array of random arrays with numbers 0 or 1

```
FOR _ in range(no_vertex):
        Row=[randomInt(0,1) for _ in range(no_vertex)]
        Matrix.Append(Row)
```

Graphs generated have between 2 and 81 vertex and are saved in different files inside the graph folder. Each file name is different based on said number of vertex, so that files can each graph can be easily accessed. Inside each file, 2 lines represent the graph. The first line represents the adjacency matrix and the second one represents the vertices along with their weights.



```
0:1|0:0
5:9:4|7:1:73
```

Figure 4 - File of Graph With 2 Vertex

Each row of the adjacency matrix is written in the first line and separated by the '|' char. Furthermore, each column of each row is simply separated by the ':' char. This allows for a simple understanding of the file as well as fast loading time for each graph. Finally, on the second line, each vertex is written along with their weight.. For each element separated again by the '|' char, we can split the string into 3 different numbers separated by ':'. The first 2 represent the vertex x and y coordinates, the last one represents the given weight.

B. *Exhaustive Search*

The script search.py is responsible for doing all the brute force searching for a solution for the given problem. The class defined inside takes into consideration a given graph G and starts by transforming the adjancency matrix into a dictionary of available actions for each vertex.

REACH = {V,Actions(V) For Each Vertex}

This dictionary represents the immediate connections for each vertex and will be the basis for the search algorithm.

Afterwards, foreach vertex we will consecutively find connections and add those connected vertex to the REACH dictionary.

Below is represented the idea behind the developed algorithm

```
REACH = DictWithActionsFromEachVertex
FOREACH vertex Vi:
        V_NEXT = Actions(Vi)
        While V_NEXT is Not Empty:
                act = V_NEXT.pop(0)
                new_actions=Actions(act) if not in
                                REACH[Vi]
                REACH[Vertex] += new_actions
                Vi = act
                V_NEXT.AddToFront(new_actions)
```

Each vertex is represented by a node in the search tree. We start by finding the immediate reachable vertex from the initial node. Afterwards we start by also exploring the immediate actions for each of the ones found before and consecutively. As such we are using the depth first search technique in order to achieve our results. Seen that we are saving the results in a global dictionary, some explorations will not be made again.

For the worst case possible, each vertex will have a connection to all other vertex. As such, the complexity of the above mentioned algorithm will be O(n^2)

C. *Greedy Search*

In order to achieve a fast greedy search for the maximum weighted closure, the Node class was created for better representing each vertex in the search tree.

```
CLASS Node:
    STATE - Current State or Vertex Coordinates
    PARENT - Parent Node during search
    INITIAL – Root State
```

While searching, a PriorityQueue structure is utilized in order to sort which nodes should be explored. This priority is the heuristic for each node. For this algorithm, preferable nodes to be explored are the ones with higher number of outgoing edges (which will probably result on even more nodes to be explored), as well as the biggest weight of those edges. The heuristic is the addition of the number of outgoing edges from the vertex A and the weight of the vertex which are directly reachable from A

HEURISTIC = LEN(REACH(V)) + SUM(Weight(Reachable_Vertex))

During the greedy algorithm, the REACH dictionary (dictionary vertex: actions from vertex) is constantly being updated, as such, the heuristic for each vertex is also constantly changing. Below is represented in a simple way how the greedy search algorithm works.

```
V_NEXT = PriorityQueue()
N = Node(state: VertBigHeuristic, parent : None, initial: VertBigHeuristic)
V_NEXT.Put(N)
REACH = DictWithActionsFromEachVertex
TOEXPLORE = DictWithActionsFromEachVertex
WHILE V_NEXT.NotEmpty():
    NV = V_NEXT.Pop()
    #Get reachable vertex from NV not yet explored
    NOTEXPLORED = [TOEXPLORE(NV.State) If not in TOEXPLORE(NV.Initial)]
    FOR ALL Previous Nodes:
        REACH[PrevNode]+=NotExplored
        ToExplore .Remove (NV.State)
    IF TOEXPLORE.isEmpty():
        return NV.Initial, REACH[NV.Initial], TotalWeightOf(REACH[NV.Initial])
```

As demonstrated, we started by adding to the PriorityQueue the vertex with the highest heuristic given the initial data. The global dictionary REACH will contain all reachable vertex for each key in said dictionary. TOEXPLORE is initialized with the same data. On each iteration, new nodes that have not been explored yet are added to the TOEXPLORE dictionary until there’s no new nodes to expand. Furthermore, the REACH collection will also be updated on each iteration, with new vertex that have been found to be connected to the initial node or root of the search tree.

Because we are saving the nodes that already have been explored, the number of iterations has a logarithmic increment. Has such, for each root vertex, the number of iterations will decrease while the graph is being explored. As such, we can estimate the complexity as being O(n*log(n)).

IV. RESULTS

The following results were obtained through sets of 10 experiments. All values are the average from the mentioned experiments. The data presented below will evaluate how the algorithms behave for each graph. For these tests, graphs were created with a number of vertex ranging from 2 to 81. For generating graphs, it was given a seed equal to the student id upon random number generation. Each graph was generated 10 times and the algorithms were executed for each graph. Finally, all the data below is available in the files *gr_results.csv* and *brt_results.csv*.

Considering the brute force algorithm, the evolution of iterations was exponencial. However, the implemented greedy algorithm’s results are almost exactly the same as the Depth First Search implementation.

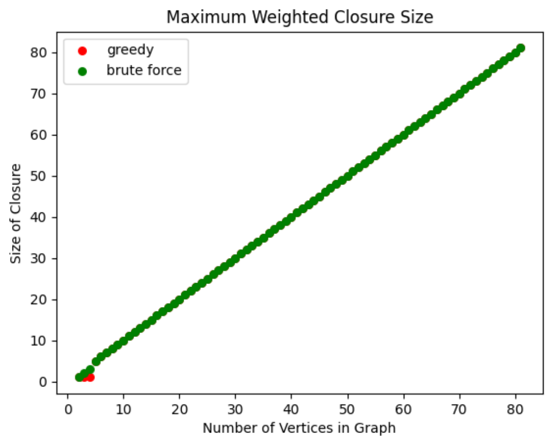


Figure 5 - Number of Vertex in Closure Solution

The figure above represents the size of the closure that is returned by both algorithms. The only disparity between both occurs on some of the first graphs. This difference is probably given to the heuristic. Considering that the greedy search follows first the vertex with the biggest number of outgoing edges and summation of weights, it probably finds a terminal vertex. As such, no new vertex are found and the search ends.

As shown in the figure below, the weight of each closure solution is almost coincidental between both algorithms. Only one graph gives a different result for each algorithm.

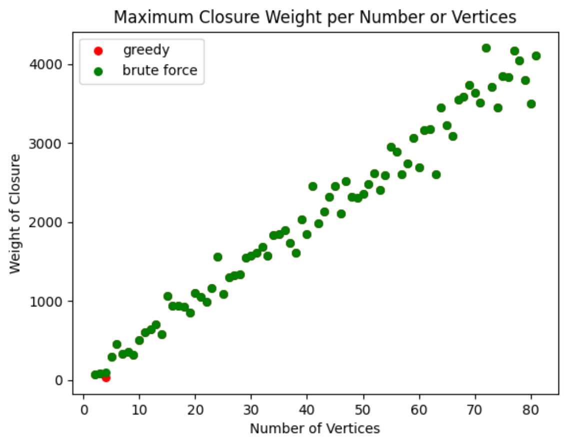


Figure 6 - Maximum weight for each solution

Even though the solutions found are practically the same, the computational complexity of the algorithms is very much different. The figure below represents the iterations made within each algorithm for each graph. As we can see, the iterations using the brute force algorithm increase exponencially, allowing us to confirm the complexity that we estimated of O(n^2). However, the greedy algorithm only reaches a maximum of 454 iterations, where as the first one reaches 6561.

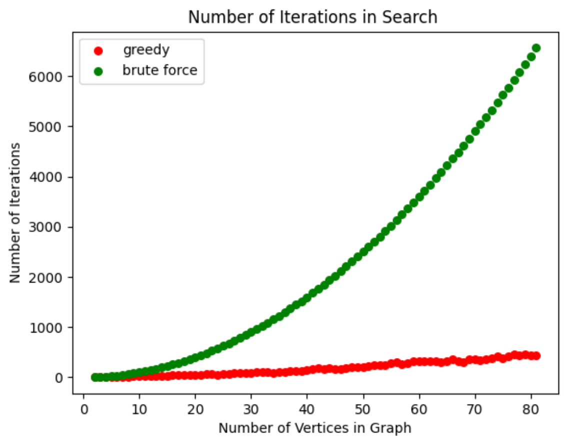


Figure 7 - Number of iterations during search

As expected, the number of iterations using the greedy algorithm is far lower. The maximum number of iterations reaches 454 where as using the exhaustive search, reaches a maximum of 6561. We can also confirm the complexity of the developed greedy algorithm as being O(n*log(n)).

The difference between both algorithms is also reflected on the comparison between execution times.

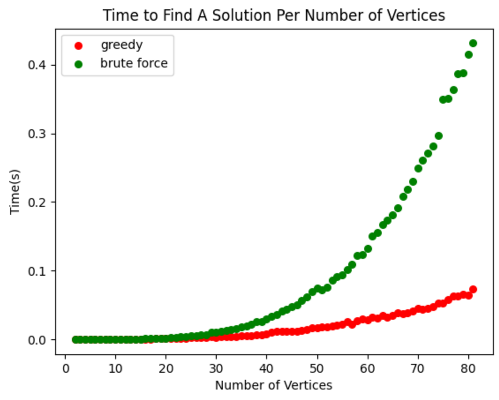


Figure 8-Execution time for each algorithm

As seen in the figure above, the increase in execution times is far bigger using exhaustive search when compared to the greedy algorithm. This was also expected, considering the difference in the number of iterations as seen previously.

It is estimated that for samples with graphs with many more vertex, the execution time would be far too great, where as the greedy algoritm would produce fast results.

N=100:

Brute Force Iterations=10,000

Greedy Iterations = 664

N=10,000

Brute Force Iterations = 10,000,000,000

Greedy Iterations = 132,877

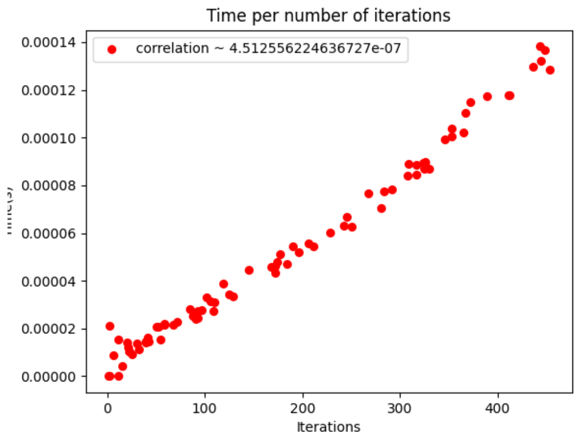


Figure 9- Correlation between execution time and number of iterations for greedy search

From the figure above we can guarante that the time per iteration increases in a somewhat linear form for the greedy algorithm. After some calculations, for this solution, it was determined that it obeyed approximatly the y=4.51*10^{-7}x equation. As such we can try and predict the execution time for the greedy algorithm. Assume TPI as time per iteration

N=100:

N_Iter = 664

TPI = $4.51*10^{-7}$* N_Iter = $2.99*10^{-4}$

Total Time = TPI* N_Iter = 0.198s

N=10,000:

N_Iter = 132,877

TPI = $4.51*10^{-7}$* N_Iter = $5.98*10^{-2}$

TotalTime = TPI* N_Iter =7,945s

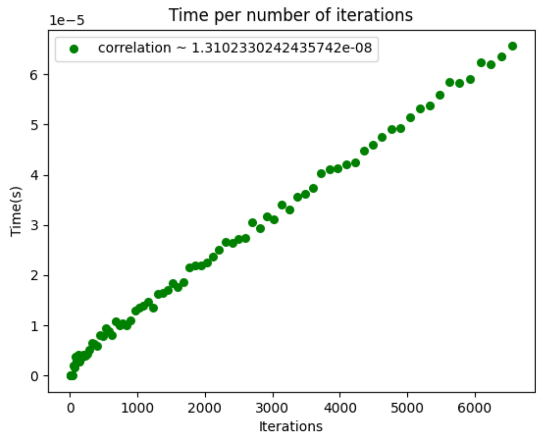


Figure 10 - Correlation between execution time and number of iterations for exhaustive search

The previous graph follows the same principal as the first one but it applies for the exhaustive search algorithm. With this data we can predict the following execution times:

N=100:

N_Iter = 10,000

TPI = $1.31*10^{-8}$*N_Iter = $1.31*10^{-4}$

Total Time = TPI*N_Iter = 1.31s

N=10,000:

N_Iter=10,000,000,000

TPI = $1.31*10^{-8}$*N_Iter = 131

TotalTime = TPI*N_Iter=$131*10^{10}$s

IV. CONCLUSION

With this work, it was obvious the difference between both algorithms. Considering the results that have been presented, it is obvious that the greedy algorithm is far better than the exhaustive search one. However, the greedy algorithm cannot guarantee the final solution to be optimal. Regardless, the compromise is acceptable, simply because of the difference on execution times. Furthermore, after predicting execution times for far larger graphs, it is obvious how much better in efficiency the greedy algorithm is.

REFERENCES

[1] Wikipedia, "Closure Problem", (https://en.wikipedia.org/wiki/Closure_problem).

[2] Wikipedia, "Flow Netword", (https://en.wikipedia.org/wiki/Flow_network).

[3] SetScholars, "Python data structure and algorithm tutorial Ford Fulkerson algorithm", (https://setscholars.net/python-data-structure-and-algorithm-tutorial-ford-fulkerson-algorithm/).