Instituto Superior Técnico – Universidade de Lisboa

2016 / 2017

Edge-connectivity in graphsNetwork Algorithms and Performance

Group nº 13:

* Dominik Gizynski, 85736
* João Caetano, 75218

# Introduction

In this mini-project we explore connectivity in graphs by applying the duality between graph connectivity and graph flows.

# Algorithms Explanation and Pseudo-codes

## 1st Script

In this script a file is inputted where each line has an edge represented by the nodes IDs. Each edge is translated as two links with opposite head and tail. Each link has a capacity of 1 and an initial flow of 0.

Then the program waits for the user to input a destination and source node for the computation of the minimum number of edges that separates them both.

This computation is done by calculating the max flow between these two nodes that corresponds to the duality of the problem described by Menger’s theorem.

For the computation of the max flow an algorithm based in the Edmonds Karp algorithm was used which pseudocode is presented next:

*edmondsKarp(graph, source, destination):*

*path = flowBFS(graph, source, destination) // BFS type algorithm that returns the shortest path available in the residual network from s to t*

*while path:*

*for edge in path: //Updates residual network*

*edge.flow += 1*

*path = flowBFS(graph, source, destination)*

*return sum(link.flow for link going from source)*

The difference between this Edmonds Karps algorithm and the traditional one is that here we do not look for the minimum difference between capacities and flow in the found path because for us it will always be 1. This improves the running time of the algorithm but not the asymptotic complexity being O(n\*m^2).

The *flowBfs* algorithm is a regular BFS with the addition that not only checks if a node has already been visited with the intention to explore it but also checks if the edge that leads to it has not filled it’s entire capacity with flow. It also saves the paths to the visited vertices and returns the path to the destination node if exists. This algorithm is presented next:

*flowBFS(graph, source, destination):*

*queue =FIFO queue initialized with the source node*

*seen(source node) = seen*

*seen(other nodes) = not seen*

*path(source) = [] # Initialize the path for the source node as nothing*

*while queue:*

*current = get first from queue*

*if current == destination:*

*return path[destination]*

*for edge in edges from current:*

*residual = edge.capacity - edge.flow*

*if residual > 0 and edge.sink not seen:*

*seen(edge.sink) = seen*

*path(edge.sink) = path(edge.source) + edge*

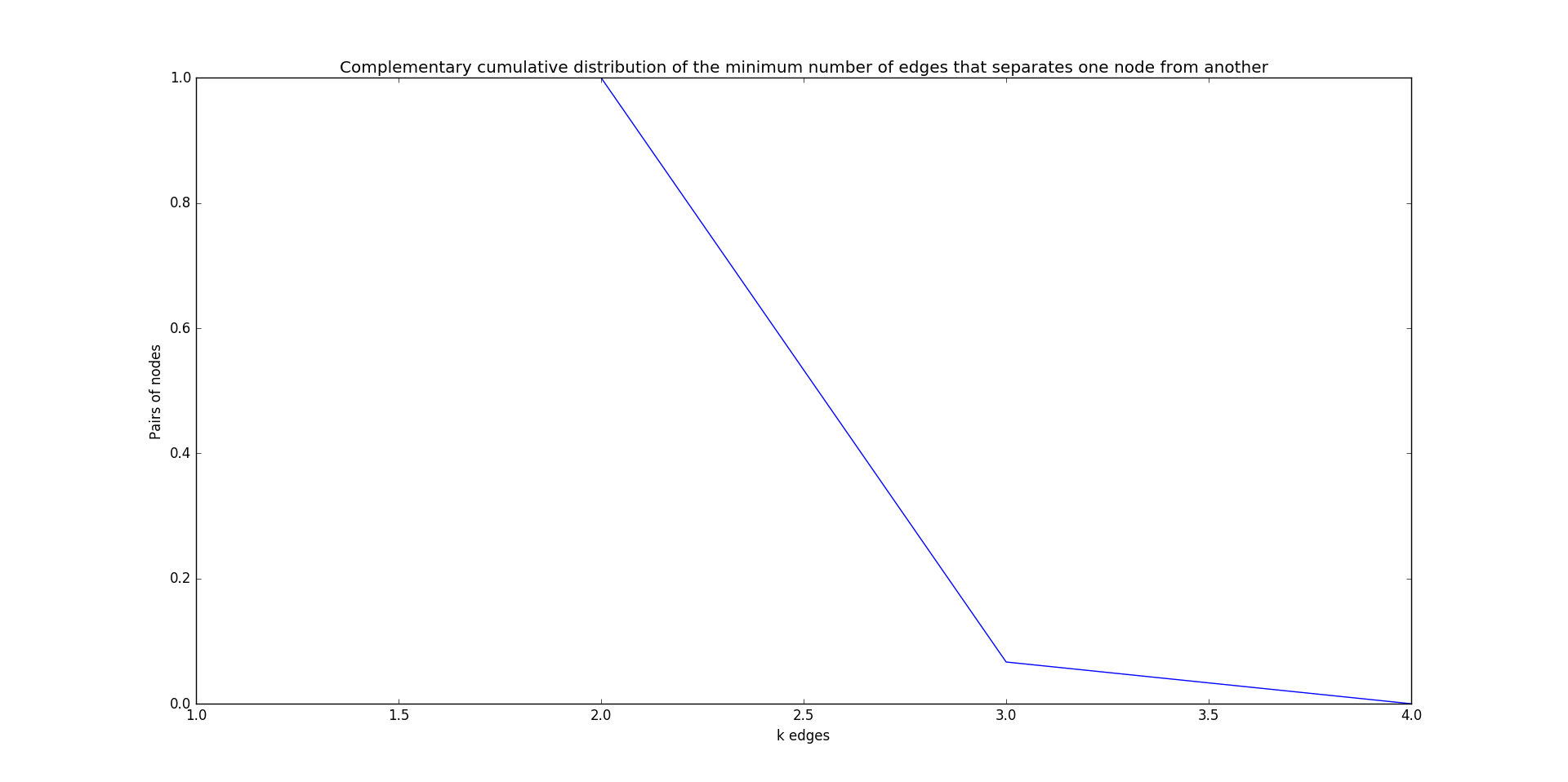
*add edge.sink to queue*

## 2nd Script

In this part, we run the above script from every node to every node (not considering when they are the same). Then we produce the complementary cumulative distribution of the minimum number of edges that separates one node from another.

As an example, for the following graph,

the next presented statistic was returned:



As we can see in this graph there is no pair of nodes that can be disconnected with the removal of one edge. They need at least two edges or maximum of three to get disconnected.

Because it runs for all pairs of nodes the complexity grows by n^2 resulting in O(n^3\*m^2)

## 3rd Script

In this part of the assignment the edge connectivity is asked, which is the minimum number of edges needed to be removed so the graph gets disconnected. So, we run again the 1st script for all pair of nodes and save the one with the minimum number of edges needed to be disconnected. For this pair, we also save the residual graph as well as the nodes visited by the BFS when running the residual graph in the last iteration where no path from s to t is available. This visited nodes are the ones that form one of the two parts of the graph resulted in a minimum cut.

Then to find one of the set of edges that would do the minimum cut we check for each of the visited nodes the saturated links that start in that node and end in a vertex not visited. This links are then printed for the user. The pseudocode is presented next:

*for vertex in visited:*

*for edge in edges from vertex:*

*if edge.flow > 0 and edge.sink not visited:*

*print(edge)*

This part of code that finds this edges takes at most n iterations when running the visited vertices and them m for each vertex edges, so O(n\*m).

# Short Discussion and Conclusion

Nothing much needs discussion.

All the tasks were concludes successfully.