Assignment #1
Networking for big data

Benedetto Jacopo Buratti, Michele Gentili,
Cristina Menghini, Paolo Tamagnini

May 2016

All the scripts are implemented using Python. Here follows brief descriptions of what we do in each part. The comments related to the different points are in the script.

1 Generate a random graph

Random Graph (n,p)

It's a Random Graph with n nodes. A generic edge between node i and node j it's added with probability p. So in the end we will have an expected value of links equal to $\frac{n(n-1)p}{2}$. The idea it's build a matrix with A $\subset \{0,1\}^{n\times n}$, do the pairwise product with the transpose in order to get a symmetric (the graph is undirected) adjacency matrix, i.e. $B = AA^t$, $b_{i,j} = a_{i,j} \times a_{j,i}$, but now edges are placed with p^2. So in bulding up the matrix A we have to use $p' = \sqrt{p}$.

Regular Random Graph (n,m)

n nodes each one connected with m random nodes. For each level j in $(1, ..., m)$ we do a random permutation of the list of nodes then split it into two groups, and match them as vertexes of new links. However we may add naturally some already existing edges, so we then do a check on the degree list and if we find a pair of nodes with *j-1* links we pick a source and destination of a random existing edge, remove this one connecting the vertex to the nodes with less links, in this way we generate two random links and delete one already existing, increasing the connectivity of the graph up to j.

2 Implement the Breadth-First Search algorithm

The BFS algorithm is considered as one of the easiest algorithms for searching a graph. When we say search a graph it means to explore it.

Given a graph G and a starting node s, also known as *root*, a breadth first search proceeds by exploring edges in the graph to find all the nodes in G that can be reached from s. The important thing to say about the BFS is that it finds the vertexes at distance *k+1* from s only when all the ones at distance k have been discovered. It is also interesting to notice that performing the algorithm

we can visualize the *search* procedure as a tree graph whose each level is defined once the previous has been completed.

The part of the script related to the BFS performs the algorithm on a single node or on all the nodes of a given graph. Moreover it is possible to visualize the result of the BFS performance starting from a *root*.

Here a description of each implemented function follows:

- `BFS(root, adjecency_list)`: Given as input the *root* for which the algoritm begins and the *adjacency list* related to each node of the graph, the BFS starts.

  The algorithm is initialized defining the following three variables: `level`(level of the tree), `parent`(parents of nodes), `frontier`(list of node to be explored in the next step). Thus we define a `while` loop that acts until the variable `frontier` is empty, it means that there are no more nodes to be explored.

  The loop does what follows: for each node in the `frontier`, for each neighbor in the adjacency list, whether it has not already been placed in the `level` variable(that has not already been explored) the algorithm fills it in `level` with the corresponding *search* level, defines parent's node and add the node to the `frontier` of the next level.

  Remark. It's important to stress the meaning of the `if` inside the algorithm, in fact it checks whether a node has already been explored, if it has, the algorithm does not consider the node in order to avoid an infinite loop.

- `create_all_nodes(list_node, ad_list)`: You can be interested in exploring the graph starting from each different node of the graph.

- `create_edges(node, adj_list, df)`: This function creates the edges of the tree graph obtained exploring the graph from a specific *root*

- `print_bfs(nods, rt, adj, data)`: Print the graph created using the previous function.

3 Check the connectivity of a graph

There are three ways to check connectivity of a given graph.

- The first is to use the adjacency matrix A. Indeed if A is irreducible then the Graph is connected. This notion derives from the fact that if A is reducible then we could split its indices in at least 2 groups.

  $$i \in I$$

  $$j \in J$$

  such that

  $$a_{i,j} = 0 \quad \forall i, j$$

If so then the graph should be made of different disconnected components.

An irreducible matrix instead shows that not even 2 groups like those can be found, that there is just 1 component, so the whole graph is connected.

- The second way is to use the Laplacian matrix of the graph. Such matrix is always symmetric and positive-semidefinite, then all its eigenvalues are greater or equal to 0. It can be proved that the number of null eigenvalues is equal to the number of disconnected components in the graph. Summing up if the second smallest eigenvalue is positive, then there is just one null eigenvalue and this means that the full graph has just one component, therefore it is strongly connected.

- The third way is empirical. By starting from a node we explore the graph with the breadth-first algorithm. If we visit in the end each and every existing node in the graph, then the graph is fully connected.

4 Generate a fat tree topology and explore it

A fat tree topology is a particular tree data structure in which each node is k-port switch and that can support up to $\frac{k^3}{4}$‘ leafs (hosts). This topology comprises k pods with two layers of $\frac{k}{2}$ switches each. In each pod, each aggregation switch is connected to all the $\frac{k}{2}$ edge switches and each edge switch is connected $\frac{k}{2}$ hosts. There are $(\frac{k}{2})^2$ core switches, each of them connected to one aggregation switch per pod.

This program generates a fat tree given in input the number of ports the user would like to have on each switch of the network. Than after generating the fat tree, the program is able to compute all the shortest paths between any pair of edge switches and also all the disjoint shortest path between any pair of edges switches. We report the description of the functionality that each declared function implements:

- `generatePod(k, numPod)`: The POD is the fundamental block of a fat tree. Each POD is composed by k number of switches each one with k ports. We have $\frac{k}{2}$ aggregation switches and $\frac{k}{2}$ edge switches. Between the two types of switches we have a bipartite interconection structure where each edge-switch is connected to each aggreate-switch. Then each edge-switch is connected to $\frac{k}{2}$ servers and each aggregate-switch is connected to $\frac{k}{2}$ core switch;

- `generateFatTree(k)`: A fat tree is composed by k PODs. Each POD is connected to all the core switches by means of the aggregate-switches: each one of those is connected to $\frac{k}{2}$ core switches;

- `dijkstra_all_shortest_paths(G, source, target, weight=None)`: This function is the networkX's implementation of the "all-shortest-paths-problem" algorithm and is used as ground truth for our implementation. It uses a modified version of the dijkstra algorithm that compute the shortest path length and predecessors on shortest paths;

- `disjointPaths(G, k, source, target)`: The disjoint shortest paths are the ones that do not have any common edge or vertex, with the exception

of the source and the target, here we compute those path in a naive way concatenate the neighbours of the source with the neighbours of the target by means of the common core-switches;

- `allShortestPaths(G, k, source, target)`: We compute the shortest paths between two elements by iterating over the neighbors at each level: edge, aggregate and core. We use the labeling used during the creation of the fat tree to understand the type of switch we encounter during the iteration.