Instituto Superior Técnico

Network Algorithms and Performance

Second part of the project

Authors :
João Maria Janeiro - 90105
Miguel Figueira - 90144
Pedro Guerreiro - 90161

Contents

1 BGP path type

1.1 High level overview

The designed algorithm defines the type of commercial path from one node to another. This algorithm mimics BGP.

The priority considered for the type of path is:

$$p < r < c < \cdot \quad \text{or} \quad 1 < 2 < 3 < 4 \tag{1}$$

The algorithm establishes the BGP path from every node to every other. This algorithm breaks down to two parts: a custom BFS from all nodes to a single destination, and a for loop, which iterates such BFS for every node.

1.1.1 Algorithm explanation

The destination node is picked as the first one to be explored. The path is built from the destination to the origin to allow that the paths formed by the closest nodes serve as building blocks to the ones formed by the nodes furthest away.

Iterating through the neighbours, the node currently being explored tries to improve its neighbour's path to the destination by passing through himself. If the proposed path is better than the neighbour's current path, the neighbour modifies its path and is added to a list of updated nodes. These updated nodes will be explored later, as they will open new possible paths.

When picking the next node to be explored, the list of updated nodes is ordered by path type. Exploring the ones that present the best type of path first will ensure fewer changes to path types.

The algorithm ends when the list of nodes to explore is empty.

1.2 Code implementation

The type of path from a certain node to the destination is kept in a array. This array will be consulted when checking if a path can be improved. The $typeOfPath[]$ array is initialized with infinity.

The list of updated nodes is organized in a *min heap* that prioritizes the nodes respecting the total order (1).

The condition to decide whether or not to change a neighbour's path is the following:

```
if(pathIsLegal(node, neighbour) && (pathTypeFrom(node, neighbour) < currentNeighbourPathType))
```

Let's break this down.

For a given node a, all of its neighbours will be offered the possibility of reaching *destination* through a. First, with b being one of a's neighbours, the extension of the path $(a \to destination)$ with the path $(b \to a)$ must be commercially legal. In other words, the path $(b \to a \to destination)$ must be a commercial path. The matrix $pathIsLegal$ is used to check this condition.

The second condition expresses that the new path must have a better type than the current one.

If both these conditions are met, the neighbour b will be added to the heap with the newly defined path type and the $typeOfPath[]$ array is updated as well.

1.3 Subtleties

Updating a path does not destroy other paths

In this algorithm, updating a node's path conserves all paths that passed through it. For every a, b for which $b \oplus a \to destination$ is legal, if the path $a \to destination$ is updated, the path $b \to destination$ remains legal.

Since if a's path can be improved, it is either of type 2/r or 3/c. Consequently, $b \to a$ has to be a connection of type 3/c in order to be commercially legal. As result, updating a's path won't hurt b's path.

All nodes are reached

As this algorithm allows checking a node multiple times (to a max of 3 times) and explores all possible paths, every pair of nodes establishes a path. This opposes an algorithm like Dijkstra's, where a node would be considered visited when reached through the best path. This could result in not discovering some nodes, if the path that leads to them is not the preferred one, and only the best option is explored in a Dijkstras.

1.4 Possibilities explored

As mentioned, a Dijkstra's algorithm with an array of *visited* nodes was considered but this would fail to find some nodes.

Alternatively, it could be verified if the given graph is commercially connected. In such case, it would be guaranteed that an unreachable node through a type 1 or 2 path establishes a path of type 3/c. This would work for the the first section, but would not be applicable for the following, so it was decided to put this algorithm aside and work on a more scalable one.

1.5 Complexity and Statistics

Regarding time complexity, the core algorithm that finds the path between all nodes and a destination has O((V + E)log n), since it is based in a BFS and the *min heap* is ordered with log n complexity. As the objective is to find the path between all pairs of nodes, the overall time complexity is O(V(V + E)log n).

The following table summarizes the statics for the $LargeNetwork$.

Table 1: Statistics for LargeNetwork

type 1	type 2	type 3	type 4
0.006117	0.112413	0.881470	0

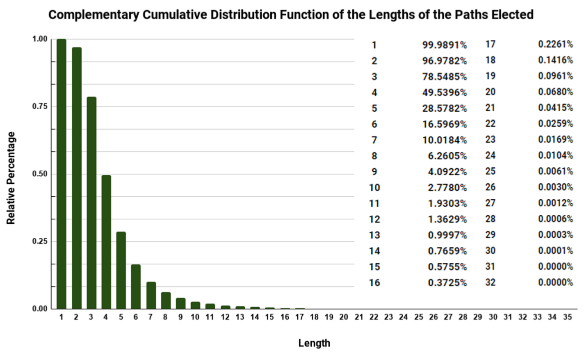
2 BGP path length

In this section, the algorithm finds the length of the BGP path, preferring the shortest one. It is used an algorithm very similar to the previous one, the only difference being in the highlighted *else if* statement that is now added to the previously existing *if* statement:

```
else if((pathIsLegal(node, neighbour)) && (pathTypeFrom(node, neighbour) ==
    currentNeighbourPathType)
&& (pathLengthFrom(node, neighbour) < currentNeighbourPathLength)) {}
```

There is another major difference: 3 heaps are used, one for each type of path. This reduces the computational time required to order the heap. Obviously, when a node is inserted in the heap, its path's length will be the one of the node that precedes him plus 1. As such, the heap is not only sorted by type, but also by length, in case of a tie.

Finally, the time complexity is the same as previous algorithm. The following figure presents the statistics obtained for the *LargeNetwork*.



1	99.9891%	17	0.2261%
2	96.9782%	18	0.1416%
3	78.5485%	19	0.0961%
4	49.5396%	20	0.0680%
5	28.5782%	21	0.0415%
6	16.5969%	22	0.0259%
7	10.0184%	23	0.0169%
8	6.2605%	24	0.0104%
9	4.0922%	25	0.0061%
10	2.7780%	26	0.0030%
11	1.9303%	27	0.0012%
12	1.3629%	28	0.0006%
13	0.9997%	29	0.0003%
14	0.7659%	30	0.0001%
15	0.5755%	31	0.0000%
16	0.3725%	32	0.0000%

3 Shortest BGP type path

Once more, the algorithm finds the path from every node to the destination, starting to build the path from the destination itself. This raises a problem: a path can't be proposed to nodes further away from the destination. In other words, if node b has a neighbour a which is closer to the destination, b can't assume that a's path will be the best for him.

As so, it is reasoned that a node has 2 paths: the one it elects, $privatePath$, and the one it presents to other nodes, $communityPath$.

To adapt to this new way of perceiving paths, the information that refers to the type of path and its length is stored in two different arrays, one for each kind of path.

The algorithm is, once again, similar to the one used in section 1. The node being currently explored tries to improve its neighbours' path, but the conditions to update a path change:

$$\text{If } \textit{proposed path's} \text{ length} < \textit{current path's} \text{ length} \longrightarrow \text{update the } communityPath$$

$$\text{If } proposedtype \text{ is better than the actual, or it is equal but its length is smaller} \longrightarrow \text{update } privatePath$$

Note that the $communityPath$ can be equal to $privatePath$, although that is normally not the case, since $communityPath$ prioritizes the smallest path, while $privatePath$ elects the shortest BGP type path.

3.1 Details

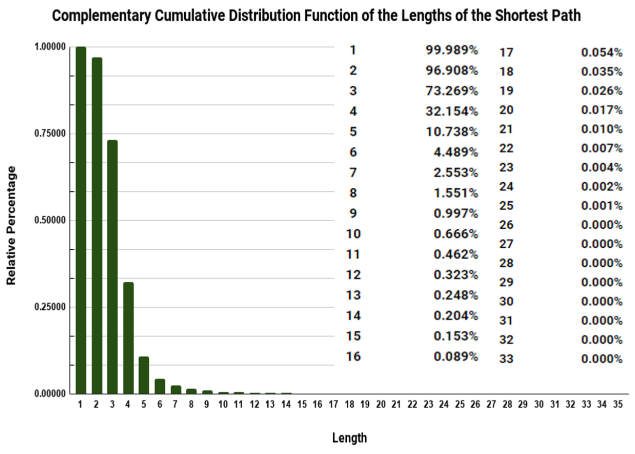
When one of the conditions mentioned above is met, the path is altered and the updated node is inserted into the heap. This happens not only for the $communityPath$ case, but also for the $privatePath$.

Such heap is ordered by path length, privileging nodes with the best type in case two nodes have the same length. Similarly to the algorithms in the previous sections, exploring the best options first will result in the fastest running time.

As result of exploring a node when its $privatePath$ is modified, it is ensured that all nodes are reached if the graph is commercially connected. In fact, let's imagine that the $communityPath$ of a certain node u is of type 3/c. This would not allow a's neighbours to go through a with a type 1/p link, for example. If these neighbours could not reach the destination through another path, or if the shortest BGP type path passed necessarily through a, the algorithm would be incorrect. Since a is inserted in the heap if its $privatePath$ is updated, all possible paths will be explored.

Finally, the time complexity is once more O(V(V + E)log n).

The results obtained for the $LargeNetwork$ are presented in the following figure. The paths go until size 35 but their percentage is 0 since there is not enough decimal cases to represent them.



1	99.989%	17	0.054%
2	96.908%	18	0.035%
3	73.269%	19	0.026%
4	32.154%	20	0.017%
5	10.738%	21	0.010%
6	4.489%	22	0.007%
7	2.553%	23	0.004%
8	1.551%	24	0.002%
9	0.997%	25	0.001%
10	0.666%	26	0.000%
11	0.462%	27	0.000%
12	0.323%	28	0.000%
13	0.248%	29	0.000%
14	0.204%	30	0.000%
15	0.153%	31	0.000%
16	0.089%	32	0.000%
		33	0.000%