Instituto Superior Técnico

Network Algorithms and Performance

First part of the project

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

Contents

1 Data Structures

1.1 Storing the graph

1.1.1 Defining a node

A node is defined as:

```
// A structure to represent an adjacency list node
typedef struct AdjListNode {
    int node; // The node
    int neighbour; // The node's neighbour
    int hierarchy; // The relationship between the node and neighbour, can either be 1, 2 or 3
    struct AdjListNode* next; // Pointer to next adjacent
} AdjListNode;
```

1.1.2 Defining the adjacency list

Using this node definition, the graph is implemented as an array of adjacency lists. At first, with the objective of keeping this array with size V, a hash function was used. In fact, the implementation included double hashing to avoid collisions, but the number of collisions was still rather big or, if an iterative approach was used, the search for an index could degenerate into O(V). As there were no memory restrictions (and the professor said to allocate an array with size 2^{16}) it was decided to simply allocate an array with the size of the **biggest index which will be referred to as BI in the future** as to have an access of O(1). Memory wise, it is used O(BI).

2 Algorithms

2.1 Determining if an input internet is connected

In order to determine if a given graph is connected it is done, first, a simple check `if (graph->edges < graph->vertices - 1) return false`. If so, the graph is not connected. On the other hand, if the number of edges is bigger than V-1 a BFS is executed, where the visited nodes are signalled. Finally, If there's a node that was not visited, a node can not reach all other nodes. Consequently, the graph is not connected. The time complexity of the BFS is O(V + E), then checking the visited array is O(BI) so the overall complexity is O(V + E + BI).

2.2 Determining if an input internet is link-biconnected

By definition, a network is link-biconnected if it has no bridges. Therefore, the function *bridges*() will search for a bridge and depending on the result, defines if the network is link-biconnected.

Firstly, it was thought that between two neighbour articulation points there would always be a bridge. In fact, this affirmation is wrong, or rather incomplete, as the two articulation points could be linked by more than one edge or involved in a cycle.

For a node u and its adjacent node v, if $discovery[u]$ is equal to $low[v]$ there is a path from v that goes back to u or, in other words, there is a cycle rooted in u. To assure that there is a bridge the condition should be:

```
discovery[u] < low[v] /*instead of*/ discovery[u] <= low[v]
```

The implementation used is based on the Articulation Points algorithm from the slides, which is itself based on a DFS. The algorithm searches the graph in depth recursively and, while traversing back to the root u, verifies if the subtree rooted in v can not climb as high as u. If that is the case, the function returns the edge (u, v).

The worst case performance would represent a time complexity of O(V + E), but as the the function *bridges*() ceases its execution as soon as it finds a bridge, it is normally much faster.

2.3 Determining if an input internet is commercially acyclic

In order to determine if a graph is commercially acyclic we simply search for a provider-customer cycle. If one is found, the graph is not commercially acyclic. As such, the implementation executes a global DFS (so it works on unconnected graphs) with a custom DFS. This custom DFS stops the adjacency iteration if a cycle was found or all the adjacencies of a node were visited.

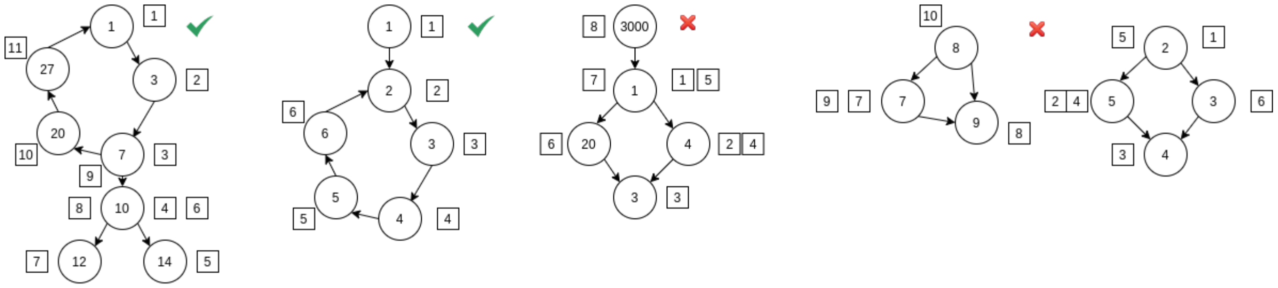
As the cycle has to be a commercial path, only the adjacents that have a connection of type '1' are analysed, the others will simply skip to the next adjacent. If the connection of the adjacent is of type '1' and has not been visited, a recursion is called on it.

Finally, if such adjacent has been visited, if it's not the previous node and has not visited all of its adjacents, a customer-provider cycle is detected. This verification is done as shown in:

```
if(!visited[adjacent]) DFS();
else if((adjacent != prevNode) && (visitedAllAdjacents[adjacent] == false))
```

It's set `visitedAllAdjacents[currentNode] = true` everytime a node leaves the adjacency iteration, to mark all of its adjacents have been visited. This array is used to prevent a case like the third one on the following image where the node 20, in iteration 6, would say that a provider-customer cycle was found. This is not true because the node 3 had already visited all of its adjacents and this can only happen in a cycle that's not provider-customer, where there is at least a node that does not have a connection of type '1' to another node. Every node should have one connection of type '1' and one of type '3' in a commercial cycle.

The following examples demonstrate the execution of the algorithm. Each square indicates the number of the iteration. The left one have a customer-provider cycle while the ones on the right do not. The last one has two disconnected graphs but with cycles that are not customer-provider.



Since a global DFS is executed, the worst case complexity is O(BI × (V+E) + BI) where it iterates through all the vertices and a DFS is called for every one of them (the adjacency list has size BI but the DFS is called only on the vertices). The additional logic did not change the complexity of the DFS. Printing the cycle adds an additional iteration over BI.

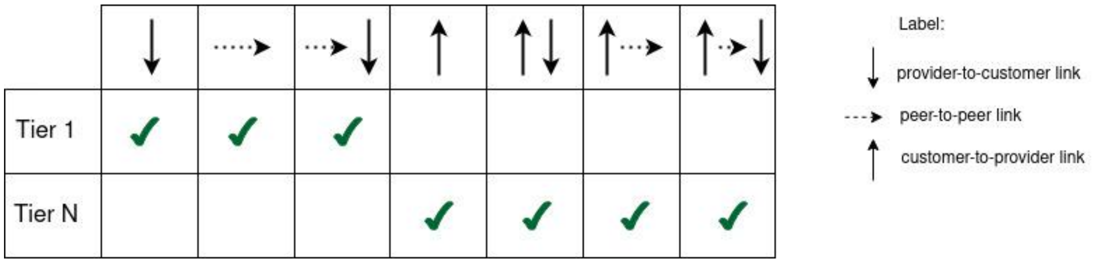
2.3.1 Output a provider-customer cycle if the internet is not commercially acyclic

We know how to detect a customer-provider cycle but how do we save it? In order to do so an array is used. Once a cycle is detected, the current node and the adjacent are added to the cycle array. Also, the $cycleFound$ flag is set to 1 and `firstNode=currentNode->adjacent`. The variable $firstNode$ indicates the origin of the cycle.

Afterwards,the algorithm proceeds to backtrack the cycle until the origin is found, `currentNode==firstNode`. When this condition is satisfied, the cycle is finished and all its elements have been added. If not, the node is added to the cycle array and the search for the origin continues.

2.4 Determining whether an input internet is commercially connected

The following table presents all possible paths for a Tier 1 node (T1) and the minimum possible paths for all types of Tier N nodes (TN). A Tier 1 node is defined as having no providers. A Tier N node has at least one provider.



	↓	⇢	⇢↓	↑	↑↓	↑⇢	↑⇢↓
Tier 1	✓	✓	✓				
Tier N				✓	✓	✓	✓

Label:

↓	provider-to-customer link
⇢	peer-to-peer link
↑	customer-to-provider link

The necessary conditions to establish that a given graph is commercially connected are:

i) All Tier 1 nodes shall be connected between them by peer-to-peer links.

ii) A Tier 1 node shall be commercially connected to every node.

The first condition expresses the impossibility of a commercial path being established between two T1 if not by a parity relationship. Given that, other than by a peer-to-peer link, a T1 node can only be accessed from its customers, and that the path comprised of a provider-to-customer link followed by a customer-to-provider is not legal, the only way to relate both T1 nodes is by a peer-to-peer link. Furthermore, only one peer-to-peer link is accepted, hence the conclusion that all T1 nodes have to be pairs.

The second condition reflects the fact that the paths possible to a T1 node are a subset of those allowed to other nodes, as shown in the following expression.

$$\{\downarrow, \rightarrow, \rightarrow\downarrow\} \subset \{\uparrow\rightarrow\downarrow\} \tag{1}$$

In other words, if a T1 node can establish a commercial path to a certain node A and to another node B, such node A can also contact B through a commercial path.

Summarising, if all T1 are connected and one of them is commercially connected to all nodes, the graph is commercially connected.

In this implementation, firstly, T1 nodes are found by the function $findTier1()$. This function checks all nodes' adjacency lists looking for one with no providers. Afterwards, the previously mentioned conditions are tested with the functions $tier1AllCon()$ and $bfsTier1()$. The first one, as the name suggests, verifies the adjacency list of each T1 node, checking if all T1 nodes are present. The function $bfsTier1()$ traverses the graph only adding to a queue of visited nodes the ones that can be accessed with legal moves to T1. As a result, only nodes that are customers of the one in front of the visited nodes queue or the T1's peers are added and marked as visited. Finally, the $visited[]$ array returned by $bfsTier1()$ is checked to verify if the selected T1 can reach all nodes.

Regarding time complexity, the function $findTier1()$ has O(BI + E) in the worst case. In other hand, $tier1AllCon()$ could have complexity of O(V^2 × E) if the graph is composed only by T1 all connected. Finally, $bfsTier1()$ has a time complexity of O(V + E).

In case there are no T1, a search for strongly connected components (SCC) will be executed. These SCC contain several nodes that can establish commercial paths between them. Therefore, if one of the nodes, a, contained in the SCC can be contacted from the exterior, all nodes can also be contacted through the node a. This SCC is considered a supernode. After finding these SCC, the graph is altered: every component of the SCC is deleted and the supernode SCC is added. All edges that existed previously connecting the exterior with elements of the SCC are now linked to the SCC. In other words, there is now a supernode in a graph with all the edges that its elements had.

This supernode has the same properties of a Tier 1. As such, the same necessary conditions previously mentioned also apply in this case. In conclusion, after finding all supernodes and changing the graph accordingly, the functions $tier1AllCon()$ and $bfsTier1()$ are invoked and will determine if the graph is commercially connected.