Graph Travesal Algorithms

Graph Algorithms

The three basic algorithms that search to find a path in a graph from the given origin to the given destination are the *Breadth First Search* (BFS), the*Depth First Search* (DFS) and the *Shortest Path* (SPS) algorithm. All three use the same basic approach and differ only in the manner in which they keep the *To Do* information of nodes to visit next.

The BFS implements the *To Do* collection of edges as a queue.

The DFS implements the *To Do* collection of edges as a stack.

The (SPS) implements the *To Do* collection of edges as a priority queue, selecting at each step to remove the edge with the shortest distance to the origin.

For this algorithm to work, we need to represent the graph as a collection of nodes (we are using a `HashMap`) where each node can look up its list of neighbors and it also can determine the distance to each neighbor.

When we visit a node *N*, we add edges leading to all of its neighbors to the *To Do* collection: the information that we came from *N* and, in the case of the SP algorithm, also the distance to each neighbor if we reached it through the node *N*.

In addition, as we go on, we keep track for every visited node how did we get there (the edge that we took to reach this node). This is our *path* represented as an instance of the `Path` class.

Each algorithm consumes the name of the starting and ending nodes and the data that represents the graph and produces a list of edges that represent a route from the starting to the ending node.

Here is a brief description of all three algorithms:

Search Algorithms

1. Start with an empty *To Do* collection and initalize it by adding to the *To Do* the first *edge* with an empty `String` as `origin`, our start node as `destination` and `distance` equal to zero.

2. Repeat the steps 3. though 4. until one of the conditions in the next step is satisfied.

   - The *To Do* collection is empty, in which case no path has been found.
   - Remove an edge from the the *To Do* collection. Add the removed edge *(from X to N distance d)* to the *path*.

     If the the `destination` of the edge we have removed from the *To Do* collection matches our desired *destination* finish the work with the *Backtracking algorithm*

3. Add edges leading to all neighbors *M* of the node *N* to the *To Do* information as follows:

   - Do not add an edge leading to *M* to the *To Do* collection if *M* has been already visited (it appears in as a `destination` of one of the entries in the *path*).
   - When adding an edge leading to *M* to the *To Do* collection do the following:
     - For the DFS and BFS do not add, if the *To Do* collection already contains the node *M* as a `destination` of one of its entries.
     - For the SPS

       If the *To Do* collection does not contain a `destination` equal to the node *M*, add the edge from *N* to *M* with the distance that is the sum of the distance to the node *N* and the distance between the nodes *M* and *N*.

       If it already contains the node *M* as a `destination` of one of its entries check if the new distance (computed as above) is shorter that the one already recorded in the list. If the new distance is shorter, replace the previous entry for the destination to *M* in the *To Do* collection with the new one.

Backtracking algorithm

Initialize a list of edges that will represent the *route* from the given *origin* to the given *destination* in our *graph*.

1. Remove the edge leading to the destination from the *path* and add it to the *route*.
2. 2. Repeat: In the *path* find the edge that has as its `destination` to the `origin` node of the edge that has just been added to the *route*.
3. Add it to the *route*.
4. If the `origin` node of the edge that has just been added to the *route* is the empty string, stop and return the computed *route*, otherwise return to the step 2.