Crossing Puzzle

V1.0

22-Apr-2016

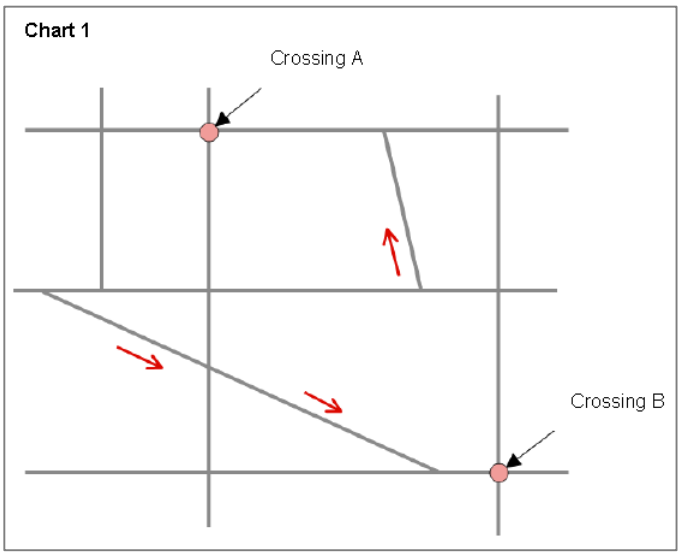
(Developed by Manish Agrawal – manish.agr.in@gmail.com)

Contents

1. Assignment

Please see the Chart1, a map. You are going to find a direction to move from Crossing A to Crossing B by car. Please explain an algorithm where it could **find efficiently a direction from A to B with less frequency passing the crossing**.

All points where the roads are meeting up with other roads are defined to be the crossing. You may answer this by itemizing, or pseudo-code.



2. Solution for Crossing Puzzle

Solution for the puzzle has been developed using **'Dijkstra's Algorithm'**, a popular algorithm for Graph Searches.

Explanation:

- **'Nodes', 'Edges' & 'Weight'**: Graph search algorithms work based on concepts of 'Nodes' & 'Edges'.

  To apply 'Dijkstra's Algorithm', every Crossing in Chart 1 has been defined as a 'Node' and the directed connection between two nodes has been defined as an 'Edge'.
  'Node' = Crossing.
  'Edge' = Directed connection between two Nodes.

- **Problem Statement Simplified**: Objective of the puzzle is to *'Find the path between A and B with least number of Crossings'*. This could also be interpreted as the path between A and B with least number of Nodes and in turn path with least number of 'Edges'.

  In other words:
  Path between A and B with least number of Crossings = Path between A and B with least number of 'Edges'.

  Below are the Nodes and Edges defined for Chart 1:

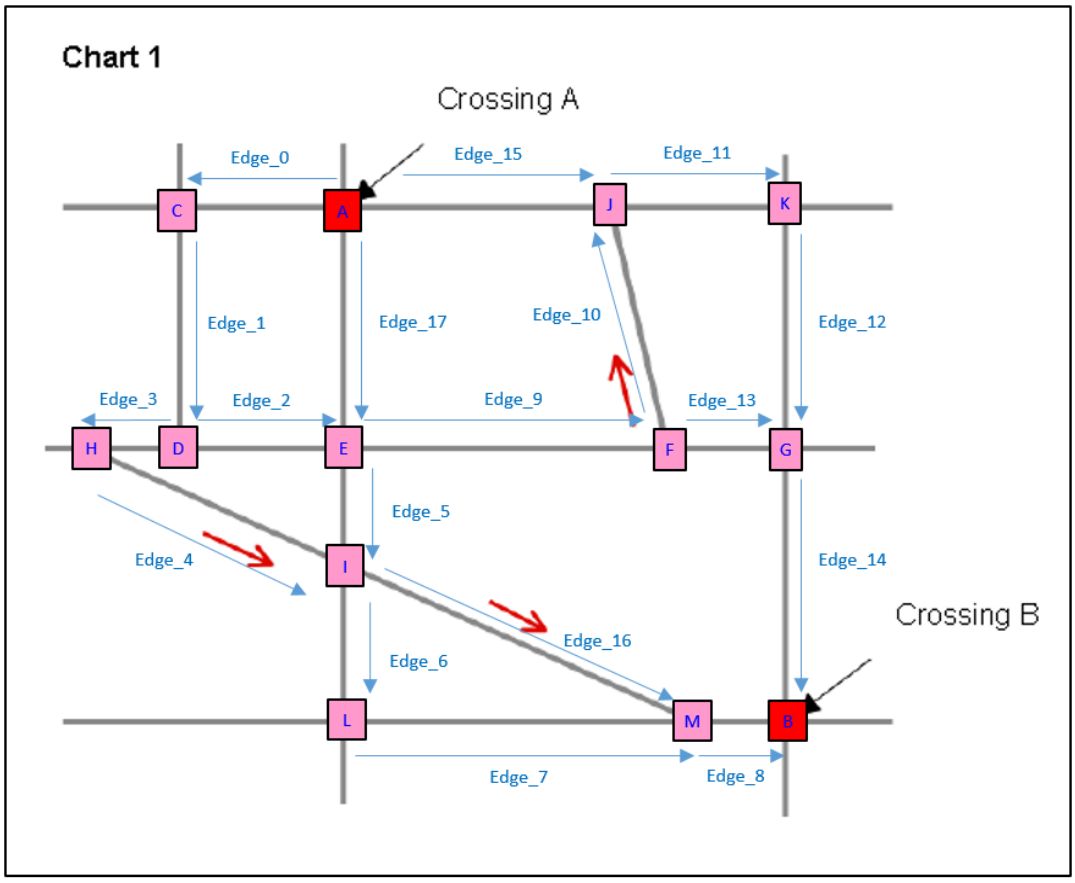
  Nodes:

13 Nodes from A to M as depicted in the below image. Source and destination nodes A & B are highlighted with different colors.

Edges:

18 Edges from Edge_0 to Edge_17 as below:

- Edge_0 - Node A to Node C, Weight 1
- Edge_1 - Node C to Node D, Weight 1
- Edge_2 - Node D to Node E, Weight 1
- Edge_3 - Node D to Node H, Weight 1
- Edge_4 - Node H to Node I, Weight 1
- Edge_5 - Node E to Node I, Weight 1
- Edge_6 - Node I to Node L, Weight 1
- Edge_7 - Node L to Node M, Weight 1
- Edge_8 - Node M to Node B, Weight 1
- Edge_9 - Node E to Node F, Weight 1
- Edge_10 - Node F to Node J, Weight 1
- Edge_11 - Node J to Node K, Weight 1
- Edge_12 - Node K to Node G, Weight 1
- Edge_13 - Node F to Node G, Weight 1
- Edge_14 - Node G to Node B, Weight 1
- Edge_15 - Node A to Node J, Weight 1
- Edge_16 - Node I to Node M, Weight 1
- Edge_17 - Node A to Node E, Weight 1



- **Equal 'Weight of all Edges:**
  As indicated above, each 'Edge' is defined with equal Weight: 1.

  This is the main premises of the solution as the Dijkstra's Algorithm is typically used to find the shortest path between two points in a Graph search based on the 'weight' defined for each Edge.

  Here since each Edge carries equal 'Weight', the algorithm will automatically find the path with least number of 'Edges', meaning path with least number of crossing between Point A & Point B.

  Important Disclaimer:

  - The problem statement only requires to find the path with least number of crossings between Point A & Point B and doesn't mention anything about the 'shortest' path between Point A & Point B with the least number of crossings.
  - Keeping the above in mind, the solution does NOT attempt to find the path between A & B which has the least number of crossings as well as is the shortest path between two points.
  - There is a possibility that if more than one paths have least number of crossings but one has a longer path than other, the path selected by the algorithm could be the one which has longer distance than the other path which has similar number of crossings but shorter path.
  - This algorithm can be extended to cover that scenario as well if required.

- **Dijkstra's Algorithm description:**

  - Dijkstra partitions all nodes into two distinct sets: Unsettled and settled.
  - Initially all nodes are in the unsettled sets, e.g. they must be still evaluated.
  - A node is moved to the settled set if a shortest path from the source to this node has been found.
  - Initially the distance of each node to the source is set to a very high value.
  - First only the source is in the set of unsettledNodes.
  - The algorithms runs until the unsettledNodes are empty.
  - In each iteration it selects the node with the lowest distance from the source out the unsettled nodes.
  - It reads all edges which are outgoing from the source and evaluates for each destination node in these edges which is not yet settled if the known distance from the source to this node can be reduced if the selected edge is used.
  - If this can be done then the distance is updated and the node is added to the nodes which need evaluation.
  - Algorithm also determines the pre-successor of each node on its way to the source.
  - Pseudocode of the algorithm in the below section.

3. Pseudocode

```
Foreach node set distance[node] = HIGH
SettledNodes = empty
UnSettledNodes = empty

Add sourceNode to UnSettledNodes
distance[sourceNode]= 0

while (UnSettledNodes is not empty) {
  evaluationNode = getNodeWithLowestDistance(UnSettledNodes)
  remove evaluationNode from UnSettledNodes
    add evaluationNode to SettledNodes
    evaluatedNeighbors(evaluationNode)
}

getNodeWithLowestDistance(UnSettledNodes){
  find the node with the lowest distance in UnSettledNodes and return it
}

evaluatedNeighbors(evaluationNode){
  Foreach destinationNode which can be reached via an edge from
evaluationNode AND which is not in SettledNodes {
    edgeDistance = getDistance(edge(evaluationNode, destinationNode))
    newDistance = distance[evaluationNode] + edgeDistance
    if (distance[destinationNode]  > newDistance) {
      distance[destinationNode]  = newDistance
      add destinationNode to UnSettledNodes
    }
  }
}
```

4. Java Implementation

A java application has also been developed to demonstrate the solution.

Available at: https://github.com/manish-agr-in/RSSFeedReader

(CrossingPuzzle_Java_v1.0.zip)