Crossing Puzzle

V1.0

22-Apr-2016

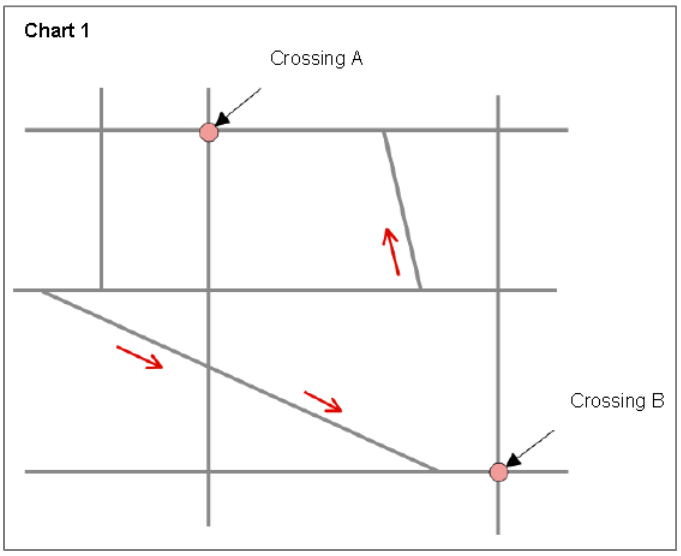
(Developed by Manish Agrawal – manish.agr.in@gmail.com)

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

- **Problem defined in terms of 'Graph Serach'**: This puzzle can be defined as a 'Graph Search' problem where each Crossing can be termed as a 'Node' and each directed connection between two Nodes can be termed as an 'Edge'.

  'Node' = Crossing.
  'Edge' = Directed connection between two Nodes.

  Objective of the puzzle can then be defined as *'Find the path between A and B with least number of Nodes'* which also means the path between A and B with least number of 'Edges':

  Path between A and B with least number of Crossings = Path between A and B with least number of 'Edges'.

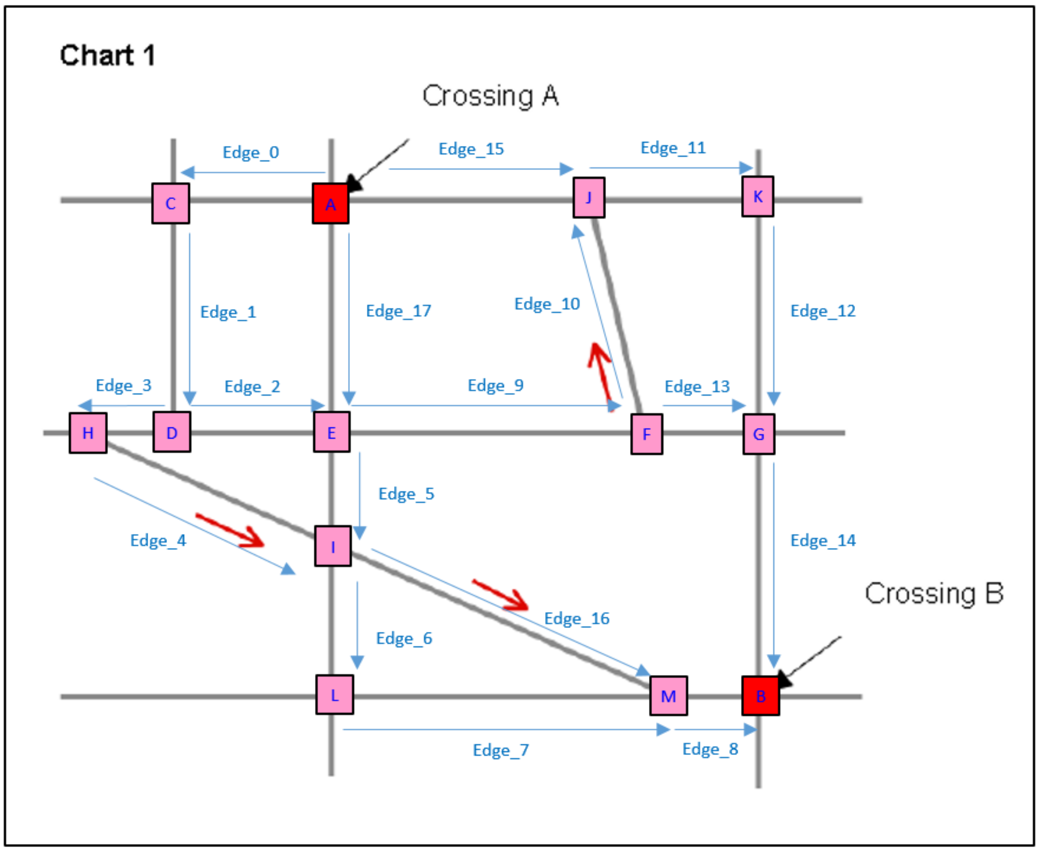
- **'Dijkstra's Algorithm' for Graph Searches:** 'Dijkstra's Algorithm'**,** which is a popular solution for Graph Searches is chosen for this problem because of its simplicity and efficiency.

- **Explanation:** Graph searches using this algorithm are accomplished by defining Nodes, Edges and weight of each Edge. The algorithm then finds the path between any two Nodes which has the least total weight of all Edges in the path.

  Typically the 'Dijkstra's Algorithm' algorithm is used to find the shortest path between any two Nodes when the weight of the Edges is expressed in terms of the proportional distance between Edges.

  Since the puzzle is about finding the path with least number of 'Crossings' between A and B *(and NOT the shortest path between them), equal weight is assigned to each Edge thus* the path with least 'weight' will automatically be the path with least number of 'Edges' (or crossings) between A and B.

- **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.

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 A & B. The path selected with the above solution might NOT be the shortest path between A and B.
- This algorithm can however be extended to find the shortest path amongst paths with least number of crossings between A and B.

- **Dijkstra's Algorithm description:**

  - 'Nodes', 'Edges' and 'Weights' of Edges are defined.
  - All nodes are partitioned into two distinct sets: **Unsettled and settled**.
  - Initially all nodes are in the unsettled sets, as they must be still evaluated.
  - A node is moved to the settled set if the path with least weight from the source to this node has been found.
  - Initially the weight of each node from 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 'weight' 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 total weight from the source to this node can be reduced if the selected edge is used.
  - If this can be done then the total weight 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 weight[node] = HIGH
SettledNodes = empty
UnSettledNodes = empty

Add sourceNode to UnSettledNodes
weight[sourceNode]= 0

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

getNodeWithLowestWeight(UnSettledNodes){
  find the node with the lowest weight in UnSettledNodes and return it
}

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

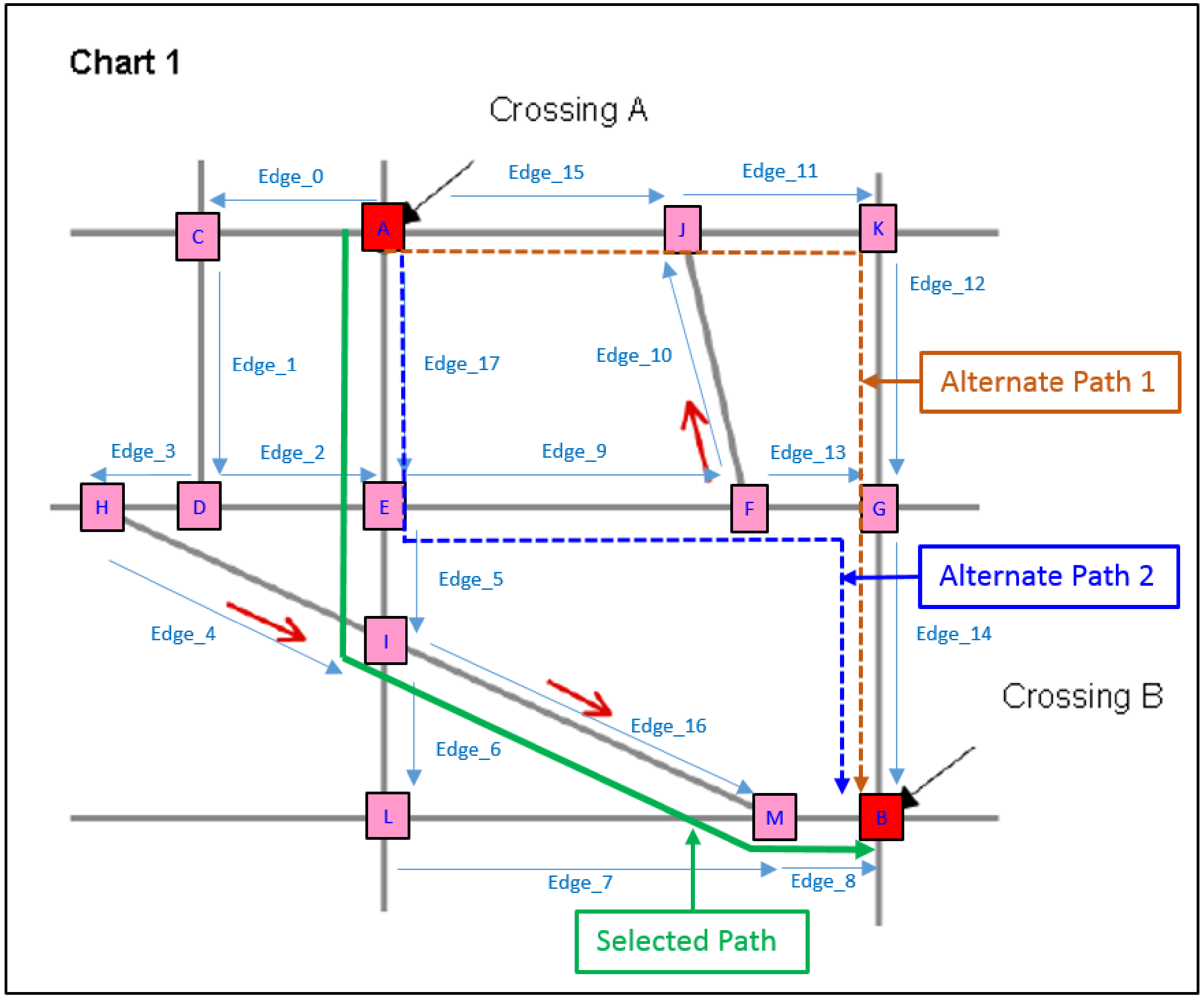
4. Java Implementation

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 – is an Eclipse project which can be unzipped and imported to Eclipse IDE).

The Java implementation, when executed, correctly finds the path with between A & B as:

Node_A > Node_E > Node_I > Node_M > Node_B



Alternate paths (e. g. ***Node_A > Node_J > Node_K > Node_G > Node_B*** OR ***Node_A > Node_E > Node_F > Node_G > Node_B***) could also have been selected by the algorithm as they also have the same number of edges (4) between them.

Algorithm would however NOT chose any path with edges higher than 4.