K-Shortest Paths Algorithm

Tomas Sheiles-Hutton s5132012

# Overview

Finding the shortest route between two locations through a particular set of nodes is a common computer science problem. While finding an arbitrary route between two nodes is a relatively simple problem; without using a heuristic to determine an efficient route, routes between nodes in a large graph may become needlessly long. In order to plot an efficient route between two nodes, some heuristic is required. For this project Dijkstra’s algorithm was used to find the shortest route between two nodes of a loop-less, directional graph. This algorithm calculates the distance from beginning of the graph to all adjacent neighbours, if the distance from one node to its neighbour is lower than the distance to the same node from another route, the distance is updated. Each node with the next shortest distance is then taken as the next node in the route. This ensures that the final route to each node in the graph is the shortest possible.

Finding the next shortest paths from the source to the goal node is similarly challenging. Using a brute force approach to determine these paths would be impractical for larger graphs. Because of this, a method must be determined to efficiently calculate similar divergent paths from the original shortest route. For this project, Yen’s algorithm was used to calculate the k shortest paths where k is a user defined number. This algorithm uses the previously determined routes in order to calculate the potential next shortest routes. For each node in the graph, all previously calculated next routes won’t be revisited; once all routes from the source to the goal are found, the k shortest routes are chosen.

# Algorithm Description

This program uses an object oriented solution in order to represent the data of the graph. The Graph Nodes class defines an object that contains the required attributes to represent a single node and its edges. A distance variable is used to represent the distance from the source of the graph to the object; this number will be updated by the nodes neighbours as the solution is calculated. A vector containing a list of pointers to other graph nodes objects is used to represent the edges of the node and weight between those edges. The add edge function is used to append to this vector an edge with a defined edge weight.

For file input, a unique object containing each node is needed. The program checks each individual line of the input file and if no copy of the current node is found, a new node object is added to the graph. Once all unique nodes are found, the edges between each node are determined and their weights added to each nodes object.

Once the graph is initialized, the K shortest paths can be calculated. Yen’s algorithm works by creating variations on the shortest path from the source to the goal node; because of this, the shortest path must first be calculated using Dijkstra’s algorithm.

The initial distances between all the nodes are sets to infinite and the origin node is initialized to 0. All the nodes are added to a queue that tracks which nodes are yet to be processed. The helper function find lowest distance is then used to find the node in the queue with the lowest distance from the graph origin. Once the lowest distance node is found, each edge of that node is checked; if the distance from the current node to its neighbour plus the current total distance is lower than its distance to that node from its previous lowest distance neighbour, the distance and previous node is updated. The lowest cost node is then removed from the queue and the process is repeated until all nodes have been processed. By then following the path from the previous node of the goal until the source, the shortest path through the graph has been found.

Once the fastest route has been found, the next fastest routes can be calculated using Yen’s algorithm. Each node in the fastest route in turn is used as the ‘spur node’ or the point at which the graph diverges from the original fastest route. Each node before the spur is kept the same as the original solution; this is done as the next fastest routes are likely to share the much of same path and to avoid calculating paths that diverge too far from the original. Once the divergent point is found, all neighbour nodes that have previously been taken edges are set to infinite and are not taken. Once each divergent route is calculated the K shortest paths are determined and added to a vector contain all other fastest routes.

# Pseudo Code ->:

# 



# Results and analysis

|  |  |  |  |
| --- | --- | --- | --- |
| Input nodes | K size | answer | time |
| C, D, E, F, G, H | 3 | P1: C E F H cost = 5  P2: C E G H cost = 7  P3: C D F H cost = 8 | 0.0063 seconds |
| 0, 1, 2, … , 11824 (route 7685, 8714) | 3 | n/a | n/a |

While theoretically, the algorithm can solve for the larger input file, due to an issue with the initialization, only single digit inputs could be accepted. Using an integer to store the nodes instead of a char could have solved this issue.

The time complexity of the program needs to be analysed in stages, as each section has a separate complexity from the previous. First, the fastest route must be calculated using Dijkstra’s algorithm; as this implementation uses a regular vector for storing the values of the nodes, each node in the queue must be searched linearly. This produces a run time of where is the number of vertices in the original queue. This performance can be improved with the addition of a min heap priority queue solution for storage of the nodes; this would improve the performance to that of in the worst case, as the entire queue does not need to be searched for each next lowest value.

Once the fastest route is calculated, the K next fastest routes must be determined using Yen’s algorithm. For this, the program must make N calls to the Dijkstra function where N is the number of nodes in the previous route. The size of each call is reduced however as the root path is not recalculated for every new path. Because of this, the worst case performance for the Yen’s algorithm function is where K is the input K value; N is the number of nodes in the route and is the complexity of each call to the Dijkstra function. In order to improve this performance, a number of improvements could be made. Storing the routes in priority queue, similar to in the Dijkstra algorithm would reduce the time to retrieve the next fastest route to . Dynamic programming with memorization could be used to ensure no routes are ever recalculated. In order to avoid recalculating the full route multiple times, a lookup table could be implemented that returns the distance for each nodes neighbour, reducing the complexity of subsequent shortest path calculations to below even.