Algorithm programming homework 3

Router

EE3 B05901172 黃禹靖

* Algorithm:

1. Dijkstra algorithm with weighting function:

W = if demand < capacity

3 otherwise

I don’t use the function in the hint . Because when demand is zero, the weights of all the edges are 0, which lead to Dijkstra’s algorithm does not rout the shortest path from source to target, but randomly routs depend on the order of edges in data structure. When demand is 0, it is like BFS in my weighting function with the weights of all the edges = 1, which can reach to optimal solution for the first routing.

And if demand >= capacity, the weights are set to 3. Because I don’t want overflow happens, so there is a discontinuity between demand < capacity and demand >= capacity. In addition, in worst case, when all the edges are overflow, I want it choose the shortest path since it would overflow in any way. So the algorithm would like BFS with uniform weight = 3.

1. Net order:

I route the pairs in the ascending order of Manhattan distance between source and target in each pair. I choose this order since it outperform all the other order I have tried in the given test cases.

* Data structure:

Class Node:

Node\* \_Predecessor:

The predecessor of the node for backtracing when the single source routing is complete.

Double \_d:

The distance from source.

Unsigned \_i, \_j:

Position of the node.

Bool \_isblack:

Record whether it have been explored in Dijstra’s algorithm.

Class Router

Node\*\* \_Grid:

A 2 dimensional array of Node, where it corresponding to the actual position.

Vector<Node\*>\*\* \_Edges:

Adjacent list recording the adjacent nodes for each node.

Vector<unsigned>\*\* \_Edges weights:

Same structure above recording the weight of the edges.

Priority queue \_pq:

Priority queue with key = distance from source to the node, and value = the node.

For extract min in Dijkstra’s algorithm.

Because it extracts min in constant time.

Vector<net> \_Nets:

The nets to rout. ( net id, (source, target ) )

Vector< pair< unsigned, vector<Node\*> > > \_Routing result:

Result of routing. ( net id, [path] )

* Discussion:

There is a difficult situation in the priority queue which is for extract min in Dijkstra’s algorihm. Because I want to access the node with min distance, so now the key is distance and value is the address of the node. But in relax, I want to modify the distance of some specific node, that is, modify the key of some value in priority queue. However, it is not good to do this, because there is not a good way to find a key according to value in priority queue.

I discuss the difficulty with 詹書愷 B0590127, and found a tricky solution.

When relaxing, if new distance is less than old distance, there is no need for modify the distance of the node in priority queue, just push a new node with the new distance. So the new node with less distance would be extracted before the old node with greater distance. Now I only need to know whether a node extracted from the priority queue have been extracted before. If yes, the node is old node with greater distance which I should discard it. And the \_isblack data member in Node can do the checking.

The drawback of this method is that the priority queue may be too big in the end, because I keep pushing items in it. But in this routing problem, the graph is actually a sparse graph, since there are at most four adjacent for each node. So a node would be relaxed at most four times. That is, the priority queue would not be greater than the four times of the amount of node. Hence, the method is acceptable in this problem.