Computer Algorithm Assignment 3

Griffith university, Gold Coast, Queensland, Australia

Convener: Saiful Islam

Student: Sushant Karki

s5170503

contents:

Algorithm design:

overview

The problem is a K shortest path problem.

The algorithm uses Dijkstra's algorithm to find the shortest path. For the next K shortest path, a result from the Dijkstra's is used. For Dijkstra's, Priority queue is used to store the nodes/vertices.

How the data is stored:

Dictionaries are used to store the node/vertices and edge weights. The format of the dictionary is {source_vertix: [destination_vertix, edge weight from source to destination]}. The dictionary is named "Map" as it is a map of the graph. Another dictionary called "Reverse_Map" is used to store the graph in format {distanation_vertix: [source_vertix, edge weight from destination to source]}.

Priority Queue:

The priority queue used here is the same one used in the previous assignment with minor modifications. First's off, priority queue is adjusted to work with the new data format and if a duplicate node is inserted in the queue, it will update it instead of deletion and reading. Also, this is an min-priority queue.

The priority queue works very fast. The reason for this is that a dictionary is used to keep track of the position of the nodes in the priority queue. this is possible because there are no repeated nodes in the queue.

Dijkstra’s algorithm:

To start off, each node except the source node’s weight is assigned to infinity. The weight of the source node is assigned as 0. This weight is not the edge weight. It is the total distance of the node from the source node. With this, the nodes are added to the priority queue.

From the priority queue, the node with the smallest weight (current node) is taken. For the first run, that is the starting node. Then, all the nodes (neighbouring nodes) connected to current node is examined. If the weight leading up the neighbouring node from the current node is less that the existing weight of the neighbouring node then the weight of the neighbouring nod will be updated. Then the updated node’s neighbours are added to the priority queue. Because the priority queue has all the nodes from the beginning, the nodes are just updated and arranged in lowest weight first sequence.

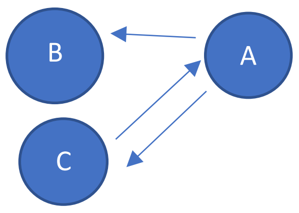
This goes on until the priority queue is empty or only has null nodes. We have not retrieved any important information. During this process, the updated weights of each node is stored in a dictionary called ‘weight’. Also, to trace back the nodes to create a path, previous node of the current node is stored in a dictionary called ‘prev’. There is also a dictionary called ‘next’ that stores the next node of the current node.

Using the ‘prev’ dictionary, the path can be traced. This Is not a requirement for the assignment but might help while debugging.

Reverse Dijkstra’s:

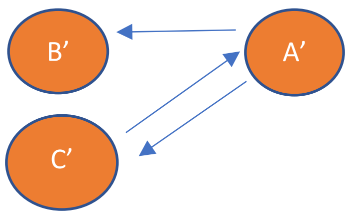
Reverse Dijkstra’s is when the starting node is the destination and destination is the starting node. the path is traced form the destination to the source. This does not mean the algorithm just runs Dijkstra’s from a different starting point. There is much more than that.

Dijkstra’s



B to A not possible

Reverse Dijkstra’s



B’ to A’ possible

Let us assume A is the source node and B is the destination node. in Dijkstra’s, the path is from A to B. if we had to apply Dijkstra’s with B as the source, then the path from B to A would not be possible because the path is directed. In Reverse Dijkstra’s however, it is possible to go from B’ to A’ Because it starts out from the destination and searches for a source.

Reverse Djikstra’s starts from the destination node and searches for the source node. Then the source node becomes the destination node and it looks for its source node.

INNOVATION, Idea behind the algorithm:

The idea is to find two different paths. From source to a node and from that node to the destination. We already have the distance/weight of each node form the source node from our Dijkstra’s algorithm. We also have the distance/weights from each node to the destination from our Reverse Dijkstra’s algorithm. All we must do now add the weights of a node from the Dijkstra’s to the same node from Reverse Dijkstra’s. This gives us the all the possible (shortest) full path from the source to the destination through each node.

Dijkstra’s			Reverse Dijkstra’s	
Source	Node A	+	Node A	Destination
Source	Node B	+	Node B	Destination
Source	Node C	+	Node C	Destination
…		+	…	…

All we have to do now is sort through the results and remove the duplicates. Duplicates more when two different path makes the same path for example: Source, A, B, C + D, Destination and Source, A + B, C, D, Destination. Theses duplicates may give slightly different results because of round off errors. That is to be filtered by making sure there is significant difference between the final distance/weight.

Because we are using Dijkstra’s, be it the regular kind or the reverse kind, it always gives the optimal distance (shortest). That means there cannot be multiple paths through the common node. for example if (Source to A + A to Destination) is the shortest path from source to destination through node A. then there cannot be any other (Source to A + A to Destination) path as there is only one shortest path. This means the algorithm works correctly.

This finally gives the K shortest paths from the destination to the source node.

Pseudo code:

```
no_of_nodes = 12
x = infinity
# To store the Graph ( source to destination )
Map = dictinary
# To store the Graph with ( destination to source )
Reverse_Map = dictinary

# To push empty node back in the priority queue
weight = {'null': infinity}

# To store the previous nodes for each node in the path
prev = Dictionary

# To store the next node for each node in the path
next = Dictionary

temp = Dictionary

# To store the combined weight of two incomplete paths
final_weight = Dictionary

# To easily find the position of the nodes in the priority queue
Map_pq = Dictionary

pq = Priority Queue

sortest_distance = 0

# To store the weights after the dijkstra's algorithm
directory_of_weights = []
shortest_path = []

# Reading the first line
f = input file
cases = information on the first line of the file
no_of_nodes  stores number of nodes
no_of_edges  stores number of edges

function reading
takes in no_of_edges
```

```
    read a line from input file and retrieve data
    store source, destination and edge weight in Map in the format {source: [[destination,weight], [destination,weight]…] }
    store source, destination and edge weight in Map in the format {destination: [[source,weight], [source,weight]…] }



reading(no_of_edges)

cases = read last line of input file
source stores the source node
final_destination stores the destination node
K = no of short distances

def initialization(source):
    # Setting the weight of each node to infinity except the source node, which is set to 0
    for key in Map:
        if key is source:
            weight of the key = 0
        else:
            weight of the key =  infinity
        prev[key] = 'null'
        # Each node will be added to the priority Queue
        Put the key in priority queue


def dijkstra(destination):

    # Getting the nodes from priority queue
    Until priority queue is empty:

        Get node from priority queue
        For each destination_node in Map[node]:

            # Updating the weight of the nodes if the new weight is less than the previous weight
            if weight[node] + edge weight from node to destination node < weight[destination node]:
                update weight of destination node
                # updating the previos node of the current node to track the path
                prev[destination node] = node
                put destination_node in priority queue


    temp = weight
    # storing the weight in a list
    Store temp in directory_of_weights

def Reverse_dijkstra(source):
    # Getting the nodes from priority queue
    Until priority queue is empty:

        Get destination_node from priority queue
        For each node in Map[destination_node]:

            # Updating the weight of the nodes if the new weight is less than the previous weight
            if weight[destination_node] + edge weight from node to destination node < weight[destination node]:
                update weight of destination node
                # updating the previos node of the current node to track the path
                prev[destination node] = node
                put destination_node in priority queue


    temp = weight
    # storing the weight in a list
    Store temp in directory_of_weights
```

```
# Quick sort
def sort(weight):
   less = []
   equal = []
   more = []

   if there are more than one weight:
      pivot = random weight
      # rearranging the Weights relative to pivot
      for current_weight in weight:
         if current_weight is less than pivot:
            add it to less
         else if current_weight equals to pivot:
            add it to equal
         else if  current_weight is greater than pivot:
            add it to more
      # returns sorted list each time by recursion
      return sort(less)+equal+sort(more)
   else:
      return current_weight

# initial run from source to destination
initialization(source)
dijkstra(final_destination)

# Stores K Shortest paths
final_array = []

# Storing the shortest path of the algorithm
node = final_destination
until node is source:
   add node to shortest_path
   node = prev[node]
add node to shortest_path
clear prev

# Second run from destination to source
initialization(final_destination)
dijkstra(source)

# After the second run, we add two incomplete paths to create a all possible paths's distance
for each node A in dijkister's :
    add (weight of node A from dijkister's + weight of node A from reverse dijkister's) to final weight


sorted_array = []
sorted_array = sort(final_weight)

for nodes in sorted_array
   choose unique wights and add them to final_array

output K shortest distances.
```

Result and algorithm analysis

```
K =  1   1035.6214009015748
K =  2   1036.71828763851
K =  3   1036.9471631289591
K =  4   1037.2310053388055
K =  5   1037.8310046138356
K =  6   1038.4562073671123
K =  7   1038.5488060515697
K =  8   1038.5679277758086
K =  9   1038.5768072636176
K =  10   1038.8226041991497
time taken is :  1.296875 (in seconds)
```

Analysis :

$O(n^2)$ for priority queue from previous assignment

O(n) for function 'reading(no_of_edges)' because of a for loop

O(n) for function 'initialization(source)' because of a for loop

$O(n^2)$ for dijkstra(destination) and Reverse_dijkstra(destination) because of the nested loop. (for loop inside while loop)

O(n log n) for quick sort algorithm

O(n) for finding the path, adding the weights of 2 complete path pairs, filtering and printing K distance

The final performance analysis gives us $O(n^2)$ as the complexity of the algorithm