Dijkstra's Algorithm

Aditya Gupta

October 2022

1 Introduction

In the previous assignment, we looked at a way to find the shortest distance between any two nodes in a graph. Based on our benchmarks and analysing that solution we came to the conclusion that it wasn't a very efficient solution to this problem. In this report, we try to improve our solution by making use of priority queues. This implementation is known as Dijkstra's algorithm.

2 Shortest Path

This implementation is analogous to a breadth-first search inside of a graph structure if we made use of a priority queue. In this implementation whenever we come across a node in the graph we add its neighbour nodes to the queue.

However, unlike a data structure such as a binary tree or other linked data structures, in graphs, there is a strong likelihood that we may encounter loops. To avoid loops, our algorithm checks if we have already determined the shortest path to the neighbour node. In case we have done that we don't add the neighbour node to the queue. Otherwise, the algorithm is made to check if that node is present in the queue, If we find the node inside the queue then we update its path and if it's not part of the queue then it adds the path to the queue.

In this algorithm, our implementation is meant to work in a way such that the root of the priority queue is supposed to contain the shortest path to the "destination" node from the start node. Given our implementation, this makes sense as we are storing the distances to each node inside a priority queue which preserves the order so that the value at the root node is always smaller than all other entries inside of the heap.

In case any of the entries is smaller then it would be readjusted so that the distances are in sorted order by updating the route or path at the root. To get this algorithm we need 3 main components: the graph (the same one that we used in the last assignment), a priority queue, and a way to store all the cities using an indexed fashion (for this we use a Hash map).

3 Identity Sequence (Hash Map)

In the previous implementation, we constructed a hash map where the keys were represented by the city names and the values associated with all keys of this sort were the city nodes associated with that name. We create a new hash map in which every city has a key such as 0, 1, 2, 3...(This results in a one-one map). Now, this makes it a lot easier to look up information on any of the cities. Whenever we add cities to the old hash map, now we also add them to the ”indexed” hash map. Now each city has a unique key.

```
 public void addNew(City city){
    if(k<array.length){
        city.setHash(k);
        array[k++] = new Node(city);
    }
}
```

4 Priority Queue

We make use of the array implementation of the heap and the array contains a node made out of the following attributes: The current city (to keep track of the cities the path has travelled through), the distance (This is used as the priority in this heap), and the previous city.

Our heap makes use of the distance as a priority, where a smaller distance is treated as a higher priority. We have the `bubble()` function take a position where it starts the bubbling process. This is used to update the queue and readjust values if we find a path with a smaller priority than a path inside of the queue. This is done so that the order of the priority queue is maintained such that all entries are sorted based on the distance attribute of heap nodes.

4.1 Updating entries inside of the queue:

Whenever we add a city to the queue, we also add it to the hash map with city names as keys. Whenever we want to compare the path of a city to its path inside of the queue, we can use the ”city names” hash map. We can fetch the node in the queue from this hash map, which can be used to compare the paths. We need the position of each node in the queue as well so that when we call the `bubble()` function to readjust nodes in the heap, we can give it the position of the node as an argument. Thus the position attribute is also added to the heap nodes. Now we can keep track of the position of each node inside of the heap and we can update it whenever we call bubble() function. We also have the pull function that is used for updating the queue.

```
public void pull(Node node, int distance, City prev){
    node.distance = distance;
    node.prev = prev;
    bubble(node.pos);
}
```

5 Dijkstra's Algorithm Implementation

Finally, we look at the implementation of the "better" shortest path algorithm. Here we end, as soon as the root contains the path to our destination city or whenever the queue has no elements, which is based on if I want to find the shortest path to just one city or all cities contained inside of our map. We remove the root node from the queue and update the "city names" hash map to contain the shortest path and the previous node.

We do the following for all the neighbouring nodes:

- If we have already determined the shortest distance to the neighbour node, then we go to the next node.
- If a path to the neighbour node exists inside of the queue, we check if the current node's path is shorter, and then we update its value.
- If the path to the neighbour node is not present inside the heap, we add it to the heap.
- Go to the next neighbour.

In the end, we return to the shortest path to the destination city. Here is my implementation of Dijkstra's Algorithm:

```
private static Hash shortest(City from, City to, Hash hashmap) {
    Heap q = new Heap(55);
    q.add(from, 0, null);
    City current = from;
    while (true) {
        Heap.Node currentNode = q.remove();
        if (currentNode == null || currentNode.city.equals(to)) {
            current = currentNode.city;
            hashmap.setDis(current.hash, currentNode.distance);
            hashmap.setPrev(current.hash, currentNode.prev);
            break;
        }
        current = currentNode.city;
        hashmap.setDis(current.hash, currentNode.distance);
        hashmap.setPrev(current.hash, currentNode.prev);
        int currentDistance = currentNode.distance;
        for (int i = 0; i < current.neighbors.length; i++) {
            Connection neighbor = current.neighbors[i];
            if (neighbor == null)
                continue;
            if (hashmap.array[neighbor.city.hash].distance != null) {
                continue;
            }
            Heap.Node neighborQNode = hashmap.array[neighbor.city.hash].queueNode;
            if (neighborQNode != null) {
                if (neighborQNode.distance > currentDistance + neighbor.distance) {
                    //update the element in the queue
                    q.pull(neighborQNode, currentDistance + neighbor.distance, current);
                }
            }
            else {
                hashmap.array[neighbor.city.hash].queueNode =
                q.add(neighbor.city, currentDistance + neighbor.distance, current);
            }
        }

    }
    return hashmap;
}
```

6 Benchmarks

If we compare our previous algorithm (the depth-first search) with Dijkstra's algorithm, we find that our new implementation is a lot better. We also notice that for longer paths, Dijkstra's algorithm is a lot more efficient.

Start City	Destination City	Depth-First Search	Dijkstra's Algorithm(μs)
Malmö	Stockholm	67 μs	88
Malmö	Sundsvall	6 ms	119
Malmö	Umeå	33 ms	111
Malmö	Luleå	55 ms	72
Malmö	Kiruna	156 ms	57

It took 156ms to find the shortest path between Malmö to Kiruna while the time taken to do the same thing using Dijkstra's algorithm is just 57 μs. Thus there is no contest that the new implementation is way better than the previous one. Our new implementation is 2700 times faster.

The run time of this algorithm depends on how often the entries inside the priority queue need to be rearranged. The time complexity of rearranging a heap is given by $O(log(v))$ (where v represents the number of vertices). For every iteration, we rearrange the heap every time a vertex is removed (dependent on the number of vertices) and if a shorter path to a vertex(or in our case a city) is found (which depends on the number of connections or edges).

Thus the time complexity of the Dijkstra Algorithm can be defined based on the above criteria using the following notation: $O((e+v).log(v))$ where e is the number of edges and v is the number of vertices or for the purpose of this assignment e represents the number of connections and v represents the number of cities. We know that the time complexity of the depth-first search is exponential or given by O($2^{\sqrt{n}}$). Thus the newer implementation is incredibly fast compared to the version used in the last assignment and more specifically for larger graphs.

After this, we find the shortest time taken to go to all cities from Malmö. This is another benefit of this implementation that we can find the shortest path to all nodes inside the graph.
Here are my results for the shortest path to each city from Malmö:

Stockholm 273
Södertälje 252
Norrköping 193
Katrineholm 218
Linköping 169
Mjölby 153
Nässjö 114
Alvesta 81
Hässleholm 43
Lund 13
Malmö 0
Göteborg 153
Varberg 114
Halmstad 85
Åstorp 49
Skövde 209
Herrljunga 192
Falköping 193
Jönköping 148
Värnamo 120
Emmaboda 131
Kalmar 160
Kristianstad 69
Karlskrona 164
Hallsberg 249
Årebro 268
Arboga 289
Västerås 297
Uppsala 324
Gävle 383
Sundsvall 600

Ånge 548
Åstersund 612
Umeå 790
Boden 976
Gällivare 1095
Kiruna 1162
Luleå 1002
Borlänge 410
Mora 484
Sveg 645
Sala 359
Avesta 363
Storvik 405
Fagersta 339
Frövi 289
Ludvika 375
Eskilstuna 264
Strömstad 307
Uddevalla 227
Trollhättan 197