Dijkstra

Dhruba Das

Spring Fall 2023

Introduction: Endgame

Ah, here we are, the final assignment for this course. This course has been special to me, to say the least. I met new people, made new friends, and learnt quite a bit. This assignment was not easy, but through the guidance of my brilliant lecturer, I was able to complete it, although encountering some minor bugs as usual. There are 3 main problems in this assignment: modifying the City class and the lookup function of the Maps class, implementing the array heap/priority queue, and implementing the Dijkstra class.

City class

This class remains the same as the previous previous assignment, with only the addition of an ID, a unique number given to each city.

```
import java.util.ArrayList;

public class City {
    String name;
    Integer id;
    ArrayList<Connection> neighbors;

    public City(String na, Integer i){
        this.name = na;
        this.id = i;
        this.neighbors = new ArrayList<>();
    }

    public void connect(City nxt, int dst){
        Connection c = new Connection(nxt, dst);
        neighbors.add(c);
    }
}
```

.

Map class

This class remains the same, with the exception of only the lookup method.

lookup method

This method is the same as that of the previous assignment, except now, if we don't find a particular city, we add, and also assign it an ID, which, I have chosen to be it's index in the cities array, given by the size variable.

```
public City lookup(String name){
        Integer indx = hash(name);
        while(true){
            if(cities[indx] == null) {
                City city = new City(name, size());
                size++;
                cities[indx] = city;
                return city;
            }

            if(cities[indx].name.equals(name))
              return cities[indx];
            System.out.printf("collision: %s and %s both has to %d\n",
               cities[indx].name, name, indx);
            indx = (indx + 1) % mod;
        }
    }
```

Path class

The path class contains the information as shown in the code below, with comments explaining the code:

```
private class Path{
        //Info we have on a found sofar shortest path

        private City city; //city we're at
        private City prev; //previous city
        private Integer dist; // total distance from destination
        private Integer index; //index in queue

        private Path(City cty, City prv, Integer dst) {
```

```
            this.city = cty;
            this.prev = prv;
            this.dist = dst;
            this.index = null;
        }
    }
```

Queue class

In this section, I will be going through the methods implemented in the Queue class.

Queue constructor

This is used to initiate, or construct a queue. We begin with a last variable, which points to the end of the queue, and an array which contains the heap. The heap here, is maintained according to the *dist* value of a path. The lower this value, the higher the priority of the path. Below is the code snippet:

```
private class Queue{
        //Array implementation

        private int last;
        private Path[] heap;

        private Queue(Integer n){
            last = 0;
            heap = new Path[n];
        }
```

add method

This method adds a path to the end of the queue. Below is the code snippet:

```
private void add(Path k){
            heap[last] = k;
            k.index = last;
            bubble(last);
            last++;
        }
```

remove method

This method removes the first path from the beginning of the queue. Below is the code snippet:

```
private Path remove(){
    if (last == 0)
      return null;
    Path nxt = heap[0];

    heap[0] = heap[(last - 1)];
    heap[0].index = 0;
    heap[(last - 1)] = null;
    last--;

    sink(0);
    return nxt;
}
```

getParentIndex method

This method returns the index of the parent of the current path we are looking at. Below is the code snippet:

```
private int getParentIndex(int t) {
    return t % 2 == 0 ? (t - 2) / 2 : (t - 1) / 2;
}
```

bubble method

This method bubbles up the path we are looking at to the correct index in the queue. Below is the code snippet:

```
private void bubble(int i) {
    if (i == 0)
      return;

    int parent = getParentIndex(i);

    if (heap[parent].dist < heap[i].dist)
      return;

    swap(i, parent);
    bubble(parent);
}
```

sink method

This method sinks down the path we are looking at to the correct index in the queue. Below is the code snippet:

```
private void sink(int i){
            int left = (i * 2) + 1;
            int right = (i * 2) + 2;

            if(left >= last)
              return;

            if(right >= last){
                if(heap[left].dist > heap[i].dist)
                  swap(i, left);
                return;
            }

            if(heap[left].dist < heap[right].dist){
                if(heap[left].dist < heap[i].dist){
                    swap(i, left);
                    sink(left);
                }
            } else{
                if(heap[right].dist < heap[i].dist){
                    swap(i, right);
                    sink(right);
                }
            }
        }
```

swap method

This method swaps two paths in the heap, given the indices. Below is the code snippet:

```
private void swap(int i, int j){
            Path k = heap[i];
            heap[j].index = i;
            heap[i] = heap[j];
            k.index = j;
            heap[j] = k;
        }
    }
```

Dijkstra class

In this section, I will be going through the methods implemented in the Dijkstra class, including the constructor.

Dijkstra constructor

The constructor simply contains an array called *done* and a queue. Below is the code snippet, with comments explaining the code:

```
public class Dijkstra {
 private Path[] done; //paths to cities where we know the shortest distance so far
 private Queue queue; //paths yet to be explored
 ...
 public Dijkstra(Map map){
        int n = map.size();
        done = new Path[n];
        queue = new Queue(n);
    }
```

dist method

This method returns the *dist* value of a given city in the *done* array. Below is the code snippet:

```
public Integer dist(City city){
        if(city != null && done[city.id] != null)
          return done[city.id].dist;
        else
          return null;
    }
```

cities method

This method returns the number of cities contained in the *done* array. Below is the code snippet:

```
public Integer cities(){
        Integer n = 0;
        for(int i = 0; i < done.length; i++)
          if(done[i] == null)
            n++;
        return n;
    }
```

from method

This method returns the previous city, of the given city in *done* array. Below is the code snippet:

```
public City from(City city){
        return done[city.id].prev;
    }
```

Search method

This method is used to search for the shortest route between two cities using the *shortest* method. Below is the code snippet:

```
public void Search(City from, City dest){
        Path ex = new Path(from, null, 0);
        queue.add(ex);
        done[from.id] = ex;
        shortest(dest);
    }
```

shortest method

This method finds the shortest path from a starting city to a destination city using a breadth-first search algorithm. It iteratively explores paths by dequeuing from the priority queue, updating the distance and predecessor information for neighboring cities. The process continues until the destination city is reached, and the shortest path is stored in the *done* array. Below is the code snippet:

```
public void shortest(City dest){
        while(!queue.isEmpty()){
            Path entr = queue.remove();

            City city = entr.city;

            if(city == dest){
                break;
            }

            Integer sofar = entr.dist;

            for(Connection con : city.neighbors){
                City to = con.city;
                if(done[to.id] == null){
```

```
                    Path ex = new Path(to, city, sofar + con.dist);
                    done[to.id] = ex;
                    queue.add(ex);
                } else{
                    Path ex = done[to.id];
                    if(ex.dist > sofar + con.dist){
                        ex.dist = sofar + con.dist;
                        ex.prev = city;
                        queue.bubble(ex.index);
                    }
                }
            }
        }
    }
```

Benchmarks

Below are the benchmarks which I ran to test the Search method of Dijkstra class.

Starting city	**Ending city**	**Shortest (min)**	**Found in (us)**
Malmö	Kiruna	1162	1399
Stockholm	Göteborg	211	966
Stockholm	Berlin	697	690
Stockholm	Madrid	1620	1360
Stockholm	Barcelona	1462	1003
Stockholm	Marseille	1276	1226
Stockholm	Lyon	1154	847
Stockholm	Hannover	668	729
Stockholm	Rom	1610	846
Stockholm	Milano	1403	848
Stockholm	Verona	1330	1057
Stockholm	Budapest	1354	1260
Stockholm	Manchester	1299	814

Table 1: (Shortest is in minutes; Found-in in microseconds)