# Dijkstras

Mahad Ahmed

Fall 2022

## Introduction

This is a continuation of a previous assignment called graphs. In that assignment we explored graphs and how to find the shortest path from one city to another. Now we are going to explore a better way of finding the shortest path with Djikstra's algorithm.

## Djikstra's

As said in a previous assignment we searched for the quickest train ride from city A to city B. A problem with this was that when we searched we did not remember which city we have been to. Which made us do the same thing over and over again. With help of a new algorithm called *Dijikstra's algorithm* we can tackle this problem. Not only does it tackle this problem but it can easily compute not only the quickest path from A to B, but to all other cities.

How does Dijkstra's algorithm work? The idea is to keep track of the shortest path to visited nodes. Then continue searching with the shortest path we have. If we have a shortest path and we notice it ending at the destination node then we are done with the search. Other paths probably will also lead to the destination but they will all be longer paths.

To implement this you need to keep track of a few things. Firstly to keep track of is the current shortest path found to each node. Also to keep track of the visited nodes.

## a priority queue

When expanding our search from the currrent shortest path we used a priorty queue to order our paths. The first entry we add to queue is the source city i.e where we are traveling from. The entry will hold the distance to the city, the city itself and where the path entered the city. For example the first entry would be: Stockholm, 0, null.

We also need to keep track of the paths we already explored. We do this by simply having a set. A hashSet is used to do this.

## Implementation

In this section I will take out some of the source code and explain it more in detail.

```
public Dijkstras() {
    map = new Map("trains.csv");
    dist = new Integer[52];
    settled = new HashSet<Integer>();
    pq = new PriorityQueue<Integer>();
  }
```

In a prreviouse task we used the same map to gives the connections from ciy to city. We used the same cvs file here. Then we made an array that holds the distance between cities. To have a set that is already done so we don't go back the same path we use hashSet that stores Integers. We also need a priority queue to check which is the source city. Also order the paths.

```
for (int i = 0; i < dist.length; i++)
      dist[i] = Integer.MAX_VALUE;
    dist[from] = 0;
```

Here we initialize our values to imitate infinity. This is how we begin the algorithm.

```
  for (Connection conn: current.neighbors){
        if(conn != null){
          if (!pq.contains(conn.city.number) && !settled.contains(conn.city.number))
            pq.add(conn.city.number);
            dist[conn.city.number] = dist[current.number] + conn.distance;
          }
          else{
            if (dist[current.number] + conn.distance < dist[conn.city.number])
              dist[conn.city.number] = dist[current.number] + conn.distance;
          }
        }
      }
```

Here we check every neighboring node. If there is a connection i.e a path from a city to another then we first check if it already been visited and if it is in the priority queue. If it isn't in either of them we add the path to the priority queue and calculate the distance. If it already is in either of them then we will compare the paths and choose shortest path.

## Benchmarks

We have compared the previous implemation in th graph assignment to Djiktra's. Here is how the stack up:

| **Graph (milliseconds)** | **Djiktra's(microseconds)** |
|---|---|
| 170 | 650 |

Table 1: Benchmark: From Malmö to Kiruna

This was to be expected. Our run time has significantly went down using the Dijktra's algorithm. It is much more time efficient then the previous assignment's implementation. Here we don't have to go down a path we already been to but the previous implementation doesn't save path it already has taken. So it essentially going down the same path numerous of time. Here is where we save a lot of time.

| **Djiktra's(microseconds)** |
|---|
| 750 |

Table 2: Benchmark: From Malmö to all other cites

Here we see why this algorithm is one of the best if not the best to find the shortest path. It finds the shortest path from Malmö to all other cities in our cvs.