Graphs and Shortest Path Java

Alexander Alvarez

HT P1 2024

Introduction

A graph data structure will be implemented and the shortest path between different pairs of nodes will be measured. The general behaviours of the graph data structure are described as well as how the operations were created and implemented for benchmarking.

Graph

A graph data structure manages data using a cluster of nodes or *vertices* where the connections between two vertices are called *edges*. Edges can be directed to be either one- or two-way. A collection of edges that connects two vertices that are not directly connected is called a *path*.

There are several types of graphs but the type discussed in this report is called a connected undirected cyclic graph. A cyclic graph contains paths that form a cycle and an undirected graph has edges that are two-way. The graph being connected means that one can find a path from one vertex, that leads to any other vertex whithin the graph.

Map

A CSV file is provided containing several connections between different pairs of Swedish cities with the distance in minutes. The graph is to be implemented as a map consisting of an array with vertices where each vertex is a different city. Each city has their own array of connections to other cities.

City and Connection

The `City` and `Connection` class is declared before creating the map. The `City` class contains the instance variables:

- `name` which holds the name of the city
- `next` which is a reference pointer to another city

- `neighbours[]` which is an array of connections to other cities
- `size` which is the size of the `neighbours` array

The constructor applies the name to a city with a parameter variable and declares the `neighbours` array together with it's size. The use of the `next` variable will become important later, but for now it's set to `null` in the constructor.

```
public City(String name){
  this.name = name; // apply parameter name
  this.next = null; // next city is null
  this.size = 1; // start size
  this.neighbours = new Connection[this.size]; // declare array
}
```

Over at the `Connection` class we only have two instance variables and the constructor. The instance variables are `city` which holds a `City` variable, and `distance` which is an integer. The constructor has two parameters where one is a connected city which is applied to the `city` variable, and the other is the connection's distance which is applied to the `distance` variable.

Back to the `City` class we have an instance method `connect` which adds a city to the `neighbours` array. The method checks if the `neighbours` array is empty and adds the connection if it is. If the array is not empty, the method then compares to see if the connection already exists and discontinues if a match is found. If no match is found, then the method increments `size` by 1 and a temporary array `copy` is created with the new size and with the elements of `neighbours`. The bigger `copy` array is then set as the `neighbours` array where the new connection finally can be added.

```
public void connect(City nxt, Integer dst){
  int i = 0; // start at index 0
  if(this.neighbours[i] != null){ // if not empty
    while(i < this.size) // compare loop
      if(this.neighbours[i++].city == nxt) return; // return if match
    Connection[] copy = new Connection[++this.size]; // increment size/make copy
    for(int j = 0; j < i; j++) copy[j] = this.neighbours[j]; // copy
    this.neighbours = copy; // set neighbours as copy
  }
  this.neighbours[i] = new Connection(nxt, dst); // add connection
}
```

The `connect` method basically works like a dynamic stack that only adds new entries.

Creating the map

The `Map` class creates the map and includes the `City` instance variable `cities[]` which is the array containing all cities. The integer instance variable `mod` is also included and will become important shortly.

The constructor declares the `cities` array with size `mod` where it then reads each row of the CSV file. Each row has 3 columns where connections between cities are set up as $from, to, distance$. The constructor runs the instance method `lookup` on the cities of the first two columns.

The `lookup` method takes a city name from its string parameter together with the `mod` variable and generates a hash index using a hash function. The code used for the hash function in this implementation is provided from the assignment document. The method checks if the `cities` array has an empty entry at the generated index and adds a city if the entry is empty. If the entry is not empty, the method then compares the name of the stored city to the city name from the parameter for which the stored city is returned if there's a match. If the names don't match then there's a collision which is handled using the `next` variable from the `City` class to go to the next city with the same index to check for a match. If no match is found by the time the tail node is reached, then the new city is added to the `next` pointer of the tail node.

```java
public City lookup(String city){
  Integer indx = hash(city, mod); // generate hash index
  if(this.cities[indx] != null){ // if empty entry
    City current = this.cities[indx]; // current city
    City previous = null; // previous city
    while(current != null){ // until tail node is reached
      if(current.name.compareTo(city) == 0) return current; // if match
      previous = current; // set previous to current
      current = current.next; // move current to next city
    }
    previous.next = new City(city); // add to next of tail
    return previous.next; // return city
  }
  this.cities[indx] = new City(city); // add to empty entry
  return this.cities[indx]; // return city
}
```

Back in the constructor, the cities returned from the `lookup` method are stored in two variables. The integer value of the distance from the third column is also retrieved and stored in a variable. The `connect` method is then run on both city variables where both cities has the same distance but each other as the `connect` arguments.

```java
City one = lookup(col[0]); // get city "from"
City two = lookup(col[1]); // get city "to"
Integer dist = Integer.valueOf(col[2]); // get distance
one.connect(two, dist); // connect from and to
two.connect(one, dist); // connect to and from
```

Shortest path

In a new class `Paths` a method `shortest` was created to find the shortest path between two cities. I gave up after a couple of hours of trying to implement the naive version. This version has the instance variables:

- `path[]` which is used to keep track of visited cities
- `sp` which is a pointer that points at the latest element in `path`
- `opt` which keeps track of the most optimal path

The constructor declares the `path` array with size 54 which is the amount of different cities. The constructor also declares the variables `sp` and `opt` by setting them to `0` and `null` respectively. The `path` array is going to work like a stack with the `sp` variable as its stack pointer.

The `shortest` method includes a `Paths` variable `p` as a parameter which I found was necessary to reset `opt` when looping the method.

The method first compares the `City` parameters `from` and `to` and returns `0` of they match. It then checks if an optimal path exists where it then checks if the `time` parameter is greater than `opt` if it does exist. The method returns `null` if the `time` parameter is greater than `opt`. The method also returns `null` if `from` is equal to any city stored in `path`. This is all to discontinue unnecessary searches or to prevent entering an endless cycle.

An integer `shrt` is declared and the city `from` is added to the stack. For each of the `from` city's connections (`conn`), the `time` variable (initially set to `0`) and the distance of `conn` is added onto `shrt`. The method is then run recursively with the connected city `conn.city` and time `shrt` as arguments for the respective parameters `from` and `time`.

The recursive method either returns `null` or an integer value, but either is stored in a variable `dst`. If `dst` is not `null` then it's value is added onto `shrt` which the optimal path `opt` then is set to if either no optimal path exists or the distance of `shrt` is less than `opt`.

After going through each connection of the city `from` it is then removed from the stack. The variable `shrt` is set to as the most optimal path found so far (`opt`) and ultimately returned.

```java
private static Integer shortest(City from, City to, Paths p, Integer time){
  if(from == to) return 0; // destination found
```

```
    if(p.opt != null && time > p.opt) return null; // if time is greater than opt
    for(int i = 0; i < p.sp; i++) // for each city in the stack
      if(p.path[i] == from) return null; // if from is in the stack
    Integer shrt = null; // declare shrt
    p.path[p.sp++] = from; // add from to stack
    for(int i = 0; i < from.size; i++){ // for each connection to from
      Connection conn = from.neighbours[i]; // get connection
      shrt = time + conn.distance; // time so far
      Integer dst = shortest(conn.city, to, p, shrt); // recursive
      if(dst != null){ // if path exists
        shrt += dst; // add dst time onto time so far
        if(p.opt == null || shrt < p.opt) p.opt = shrt; // if no opt path
      }                              // exists or if shrt is faster than opt
    }
    p.path[p.sp--] = null; // remove from of stack
    shrt = p.opt; // set shrt to fastet path so far
    return shrt; // return shrt
  }
```

Benchmark

Running the benchmark for the shortest paths between the cities provided in the assignment document yields the results presented in table 1.

From	To	Distance	Runtime
Malmö	Göteborg	$153min$	$0ms$
Göteborg	Stockholm	$211min$	$0ms$
Malmö	Stockholm	$273min$	$0ms$
Stockholm	Sundsvall	$327min$	$1ms$
Stockholm	Umeå	$517min$	$6ms$
Göteborg	Sundsvall	$515min$	$2ms$
Sundsvall	Umeå	$190min$	$180ms$
Umeå	Göteborg	$705min$	$0ms$
Göteborg	Umeå	$705min$	$4ms$
Malmö	Kiruna	$1162min$	$52ms$

Table 1: Shortest path between cities.

The method managed to find paths that I, during benchmark observations, believe are the shortest - or atleast close to the shortest - paths. I unfortunately have no data to compare these results to. However, this version of the `shortest` method seem to include more efficient and/or comprehesive ways of finding a path compared to what I could think of for the naive version.

Running the benchmark for the shortest paths from Malmö to all other cities and sorting by distance yields the results diplayed in figure 1.

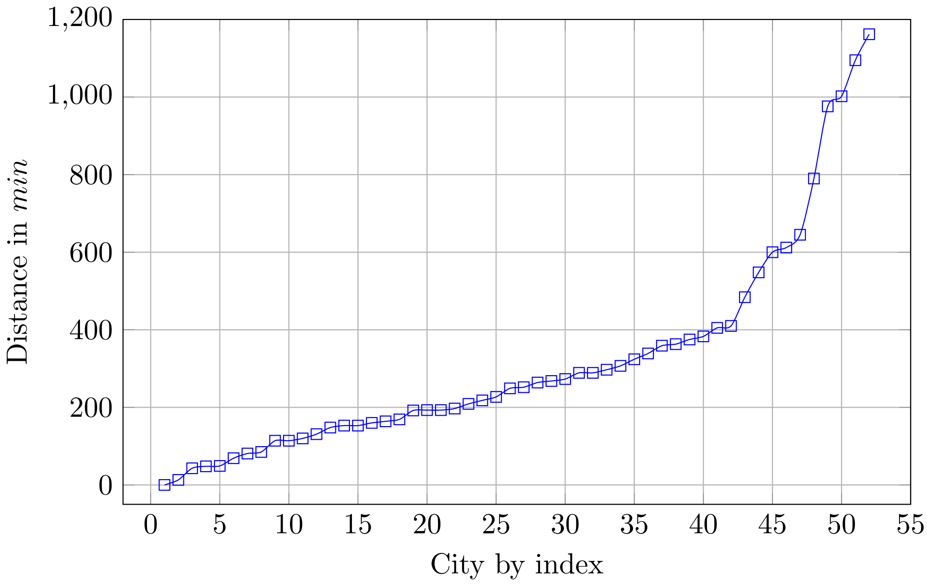

Figure 1: Distance from Malmö to cities increasingly further way.

Discussion

The shortest path algorithm is an algorithm commonly used for the graph data structure. The algorithm used in this report is called the depth-first-search (DFS) algorithm with the addition of the optimal path. The DFS algorithm has the time complexity $O(V + E)$ where V is the amount of vertices and E is the amount of edges. The method uses this algorithm when searching for a path. The inclusion of optimal path simply discontinues the search from a certain point when the path from that point becomes longer than the shortest path found yet.