Graphs

Abdinasir Abdi

October 2024

Introduction

This report concludes the ninth and final assignment in course ID1021, taken in the fall of 2024. The report explores a more general data structure called *a graph* and compares different handling implementations.

Graph

A graph is a versatile data structure where nodes(vertices) are connected by edges(links), which can either be directed or undirected. Connections between nodes are more complicated with graphs than with trees and linked lists because graphs allow paths between nodes that are not directly connected. A real-world example from the Swedish railway network will be used in this exercise to create a graph of 52 cities with 75 connections and calculate the shortest path between them. This will enable one to implement and optimize algorithms like depth-first search (DFS) to handle graph traversal while addressing performance challenges like loops.

Constructing the graph

The graph is built by extracting information from a CSV(Comma Separated Values) file. Each row lists two cities and the time it takes to travel between them. To construct the graph, cities are represented as nodes and train routes as edges. Two classes called *City* and *Connections* were made. Each city is represented by the *City* class and contains the name of the city and its connections to neighboring cities. The *Connections* class stores the travel time and destination time between cities.

Hashing

A hash table was used to improve storage efficiency and city retrieval. It takes the city's name as input and generates a unique hash by reviewing the characters and using a 31 multiplier and a modulus. Mod 577 was used for

this assignment. As studied in the previous assignment, prime numbers give better hashing because they reduce the amount of collisions.

```
     private static Integer hash(String name, int mod) {
int hash = 0;
for (int i = 0; i < name.length(); i++) {
hash = (hash*31 + name.charAt(i)) % mod;
}
return hash;
}
```

The map

A map class containing a constructor that initializes the map from the file was given. In the class, cities are identified by their name and stored in a hash table. The **mod** determines the value of the hash table and each city is placed at specific indices based on its hash code. In the case that multiple cities hash to the same index, chaining is used to handle collisions. This is done by chaining them together in a linked list.

```
    public City lookup(String name) {
    int index = name.hashCode() % mod;
    if (index < 0) index += mod;  // Will ensure positive index
    City city = cities[index];

    while (city != null) {
        if (city.name.equals(name)) return city;
        city = city.next;
    }

    // If no city was found, create a new one
    City newCity = new City(name);
    newCity.next = cities[index];
    cities[index] = newCity;

    // Return the newly created city
    return newCity;
}
```

The graph is constructed as an object(Map) where each given connection in the file is transferred into new city nodes if the given city has not occurred previously. The method retrieves a city by its name from the hash table, and if it is not found, it will simply create a new one.

Shorted path

In this assignment task, different implementations were used to find the shortest path between two cities.

The Naive way

For this implementation, a depth-first search (DFS) is used to find the shortest path between two cities. Starting from a given city, the method recursively explores all possible paths to find the target city, tracking the cumulative distance time. DFS explores every possible path without prioritization, leading to excessive back-and-forth traversal, which makes it less efficient and could result in inaccurate results. To prevent this, a time limit constraint, also called the max depth, is set. The max depth constraint helps avoid infinite loops and reduces inefficient paths by disregarding longer paths that exceed the given time limit. The method attempts all neighboring connections for each city, adjusting the remaining time by subtracting the travel time of each connection. All paths explored are compared to find the shortest path.

```
    private static Integer shortest(City from, City to, Integer max){
    if (max < 0) return null;
    if (from == to) return 0;

    Integer shrt = null;
    for (Connection conn : from.neighbors) {
        Integer dist = shortest(conn.city, to, max - conn.distance);
        if (dist != null) {
            dist += conn.distance;
            if (shrt == null || dist < shrt) {
                shrt = dist;
            }
        }
    }
    return shrt;
}
```

A benchmark for travel between different cities was constructed and the following results were obtained:

From	To	Shortest Path	Runtime
Malmö	Göteborg	153 minutes	<1ms
Göteborg	Stockholm	211 minutes	3 ms
Malmö	Stockholm	271 minutes	3 ms
Stockholm	Sundsvall	327 minutes	29 ms
Stockholm	Umeå	517 minutes	42050 ms
Göteborg	Sundsvall	515 minutes	6 ms
Sundsvall	Umeå	190 minutes	<1 ms
Umeå	Göteborg	705 minutes	7 ms
Göteborg	Umeå	705 minutes	42300 ms

Table 1: Cost of determining the shortest path

Detect Loops

The runtime to find the shortest path between different cities varies much. This is because DFS searches all possible destinations, including all potential combinations without keeping track. This could lead to revisiting the same cities multiple times, even in cycles. One key observation made from this implementation is that the starting point greatly impacts the runtime. One can think of it as a tree where the root is the city that the shortest path is evaluated from. It is possible that one direction may branch into several early on, and since DFS investigates all potential combinations, it may lead to a poorer performance. If the direction is changed and the evaluation is from the opposite direction, fewer paths need to be explored, resulting in better runtime. A good example of this is the search for the shortest path between Göteborg and Umeå. A new implementation was created that addressed these drawbacks to improve on the previous one. Now each node that has been passed through is saved and used as a reference. This is done by checking if the current node is equivalent to any previous and if true the program stops the investigation of that path. The following results were obtained from that change:

From	To	Shortest Path	Runtime
Malmö	Göteborg	153 minutes	<1ms
Göteborg	Stockholm	211 minutes	<1ms
Malmö	Stockholm	271 minutes	<1ms
Stockholm	Sundsvall	327 minutes	<1 ms
Stockholm	Umeå	517 minutes	5 ms
Göteborg	Sundsvall	515 minutes	5 ms
Sundsvall	Umeå	190 minutes	<1 ms
Umeå	Göteborg	705 minutes	<1 ms
Göteborg	Umeå	705 minutes	2 ms

Table 2: Cost of determining the shortest path

Better Improvement

The method can be further improved by working with a dynamic maximum path. Starting off, it is set to null giving it no restrictions. When the program finds the first path to the given destination, that length becomes the maximum allowable path. When a shorter path is discovered the maximum path is updated to the lower length. This process is done recursively, continually narrowing the search scope. A benchmark for finding the shortest path from Malmö to Kiruna gave the following results:

From	To	Shortest Path	Runtime
Malmö	Stockholm	273 minutes	220 µs
Malmö	Uppsala	324 minutes	400 µs
Malmö	Sundsvall	600 minutes	5200 µs
Malmö	Kiruna	1162 minutes	110000 µs

Table 3: Shortest path times and runtimes for various routes from Malmö

These improvements have made the program much more efficient for shorter and longer paths, but still limited to smaller data sets. Because the method still uses a depth-first search it will follow an exponential time complexity $O(2^n)$ and will therefore not be suitable for bigger more complex maps.