Graph Report.

Adrian Jonsson Sjödin

Fall 2022

1 Task

In this task we will read a file in CSV format that contains cities, there neighboring city and the time it takes to travel between them. We will then implement different graph methods to find the shortest path between cities.

- Take the CSV file and turn it into a graph (map) that we will later use to find the shortest path between cities. For this you will need two other classes `City` and `Connection`. Also create a quick lookup method that will be used to add cities to the map and when traversing the graph.
- Implement a simple program that finds the shortest path between two cities, regardless if loops and double back paths are present. Do some benchmarks and present the minimum path found, and how long it took to find the path. What are the limitations of this implementation?
- Implement another program that finds the shortest path but that can avoid loops by keeping track of which cities we have already passed. Rerun the benchmarks from the previous program and see if there's any improvements.
- Finally implement an improvement in which we use the found path to set a time limit for the future. If you have that to update the max value when you try different direct connection then you no longer ned to set a max value but can rather let it be null until you find a path. Do the benchmarks again and see if there is an improvement.

2 Method & Theory

2.1 Map

We first created the different classes that was needed to create the map over the train system. These where the `City` class, the `Connection` class and the `Map` class itself.

The `Connection` class is the simplest out of them and is used by the `City` class to track which cities are connected to that city. It consists of two private final fields `City connectingCity` and `Integer distance`, as well as a constructor and two getters to retrieve the values stored in the fields.

The `City` class is what will be used in the `Map` class to build up the train network. It has two private fields `String name` and `LinkedList<Connection> neighbors`, as well as a constructor to initialize the fields. It also have two getters to retrieve the values of the private fields, and a public method called `connect(City next, int distance)` used to add the neighboring stations for city to the private field `neighbors`. Using a linked list for that field allows the station to have as many connecting cities as needed, and we don't have to know in advance the maximum amount of connecting stations. Furthermore adding them is just one line of code since we can simply use the method `add()` provided by the `LinkedList` class.

Finally for the `Map` class we have two private fields, `City[] cities` to hold the train stations in, and `final int mod = 541` used for hashing the cities into the `cities` array. When reading the CSV file we separate the line by the "," and store them in a string array. Then we send the first city to our `lookupOrAdd(String name)` method, then the second city and lastly use the `connect` method from `City` class to connect city one to two and city two to one.

The `lookupOrAdd` method takes the city's name and hash it and then first check if there is a city already stored at that index. If there's not it add that city as a new city there. If there is something stored there we check if it is the same city as the city name parameter, if it is then we want to return that city, if it is not then that means we have a collision and we then modify the index and redo the process.

2.2 Naive

We have a functioning map over the train system, so next is to implement a search method that gives the shortest path between the cities. This is done by using a *breath first approach.* To avoid getting stuck in infinite loops we also have a parameter called `max` that is the maximum time we allow the trip to take. So when we search for a path between for example Malmö and Göteborg with a maximum allowed time of 500, so will we start with going to the neighboring city that's first in the list and pass along (max - distance). Then go from there to the next city that's first in that list, and so on all the way until we either find Malmö or the value (max - distance) that we pass along becomes less than zero. If that happens we back up one city and try from there and so on. This is achieved in a similar way as the depth first search in the binary tree assignment, in that we utilize recursive method calling. Doing it this way though means that we can end up passing the same cities multiple times in some of the searches which will add to the execution time of the program.

2.3 Paths

The above search method works but it isn't very effective. In this search method we make sure to also track the cities that we have already been to. This should make it so that we don't get stuck in any loops since we can never take a path that go back to a city that we have already passed. This is implemented by simply adding the code provided in the assignment before the code that calls the method recursively. An improvement that we can add is to also set the max time to the current found short time. This will eliminate all path that we haven't yet traversed but that when we go to the first next station in will be longer than what we set short to.

3 Result

Table 1: Search time for the `Naive` class using a LinkedList and an array implementation of `neighbors` in the `City` class

Route	**Travel time**	**LinkedList impl.**	**Array impl.**
Malmö to Göteborg	153 min	2 s	1 ms
Göteborg to Stockholm	211 min	3 ms	2 ms
Malmö to Stockholm	273 min	3 ms	1 ms
Stockholm to Sundsvall	327 min	50 ms	31 ms
Stockholm to Umeå	517 min	41,045 ms	32,874 ms
Göteborg to Sundsvall	515 min	15,222 ms	12,446 ms
Sundsvall to Umeå	190 min	1 ms	1 ms
Umeå to Göteborg	705 min	3 ms	2
Göteborg to Umeå	Probably 705 min	10+ min	10+ min

Table 2: Search time for the `Paths` class with and without the improvement regarding the `max` variable

Route	**Travel time**	**Before improv.**	**After improv.**
Malmö to Göteborg	153 min	303 ms	2.6 ms
Göteborg to Stockholm	211 min	156 ms	1.3 ms
Malmö to Stockholm	273 min	255 ms	0.97 ms
Stockholm to Sundsvall	327 min	183 ms	13 ms
Stockholm to Umeå	517 min	267 ms	32 ms
Göteborg to Sundsvall	515 min	252 ms	14 ms
Sundsvall to Umeå	190 min	533 ms	378 ms
Umeå to Göteborg	705 min	244 ms	3.8 ms
Göteborg to Umeå	705 min	280 ms	59 ms
Malmö to Kiruna	1,162 min	840 ms	158 ms

4 Discussion

All the code can be found here: GitHub