ICP Individual Project

Richard Quayson

Department of Computer Science, Ashesi University

CS313_B: Reflection

September 30, 2022

In finding an optimal route for a given city, country start location and city, and country destination, you first need to read the data on the airports, airlines, and routes from the CSV files. Here, I created Airport, Airline and Route classes with instance variables representing the data in the columns of the data files. Afterwards, I created the readWriteFile, which defines a readFile method for reading CSV files and a writeFile method that takes the file path and a StringBuilder class of the data and writes to the specified path. I then created the createObjects class that defines methods for creating Airports, Airlines and Routes object from a given String array (gotten from reading the file). This method casts values into their respective data types and provides an exception handler for the airport class where there were extra commas in the airport name and city. In order to improve time complexity, the data read from the files are all being stored in HashMaps, which links the key to an entire Airport and Airline object for the airports, and airline data. Since none of the column data was unique for the route data, I used the source Airport ID as the key and mapped it to an array list of all possible Route objects with that airport ID as their source airport ID. Also, the Route class implements a user-defined interface (Routable) which returns a Boolean value on whether a given Route object is valid (has a valid source and destination airport IDs).

Afterwards, I created an abstract generic Problem class that defines methods for a Problem object. This class was then extended by the AirRoutePlanning class, where I defined the data types and implemented the methods. From here, I created a Node class that represents a given state while keeping track of the Parent that created it, the action taken to create it and the combined path cost of the Parent and itself. My Node class implements the Comparable interface for later sorting in the priority queue for the search algorithm.

Finally, I created the search algorithm class that defines the uniform cost search algorithm as a method. The method uses a priority queue of Node objects for the frontier (open list) and a hash Set of integers for the explored set (closed list). It takes a problem object as an argument and then initialises a Node from it. It then generates the possible routes from the given initial state, generates children nodes out of those, adds them to the frontier and repeats the process. Each time a node is popped off the frontier, a goal test is performed on it. If the node's state is the same as the destination Airport id, it backtracks and returns the solution path. Else, it continues the process of generating children nodes. In cases where we realise two nodes are equal (have the same airport ID), we check their path cost, and if the current node has a better path cost, we replace the old one. In the end, the uniform cost search algorithm returns a Solution object. The user-defined Solution class has an array list of airport IDs (Integers) and a path cost (double) as instance variables. Once the Solution is returned, the program calls the createSolutionString, which uses the StringBuilder class to create a string representation of the Solution. This information is then passed into the writeFile method to be written into the file. In all, the project enabled me to explore search algorithms, the priority queue, array list and set data structures. Overall, it was a fun experience.