DSA Assignment Documentation

# Introduction

This document contain information about how the program is to be used and tested, as well details on the role of each of the java class and why they were implemented in that way.

Contents

[Introduction 1](#_Toc496274694)

[Usage 3](#_Toc496274695)

[Compiling 3](#_Toc496274696)

[Running 3](#_Toc496274697)

[Testing 4](#_Toc496274698)

[Class descriptions and decision 5](#_Toc496274699)

[Place.java 5](#_Toc496274700)

[Country.java 5](#_Toc496274701)

[State.java 5](#_Toc496274702)

[Location.java 5](#_Toc496274703)

[DSALinkedList.java 5](#_Toc496274704)

[DSALinkedListIterator.java 6](#_Toc496274705)

[DSAListNode.java 6](#_Toc496274706)

[DSAGraph.java 6](#_Toc496274707)

[Dijkstra’s algorithm 7](#_Toc496274708)

[DSAGraphNode.java 8](#_Toc496274709)

[DSAGraphEdge.java 8](#_Toc496274710)

[DSAEdgeWeight.java 9](#_Toc496274711)

[FileIO.java 9](#_Toc496274712)

[Menu.java 9](#_Toc496274713)

[UML Diagram 11](#_Toc496274714)

# Usage

## Compiling

To compile the program simple type “make” into the terminal in which the .java files are saved

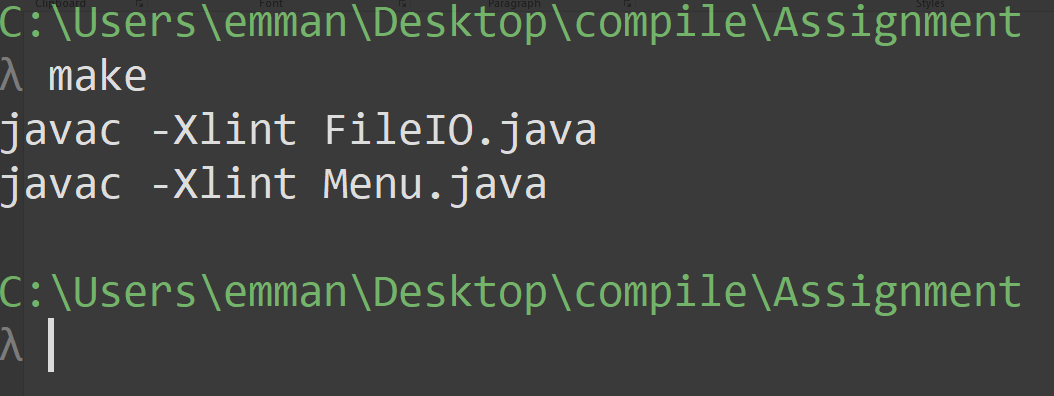


Figure : Compile

## Running

The user should see the program compile all the relevant classes, with no errors been displayed. The user can then enter “make run” to run the program.

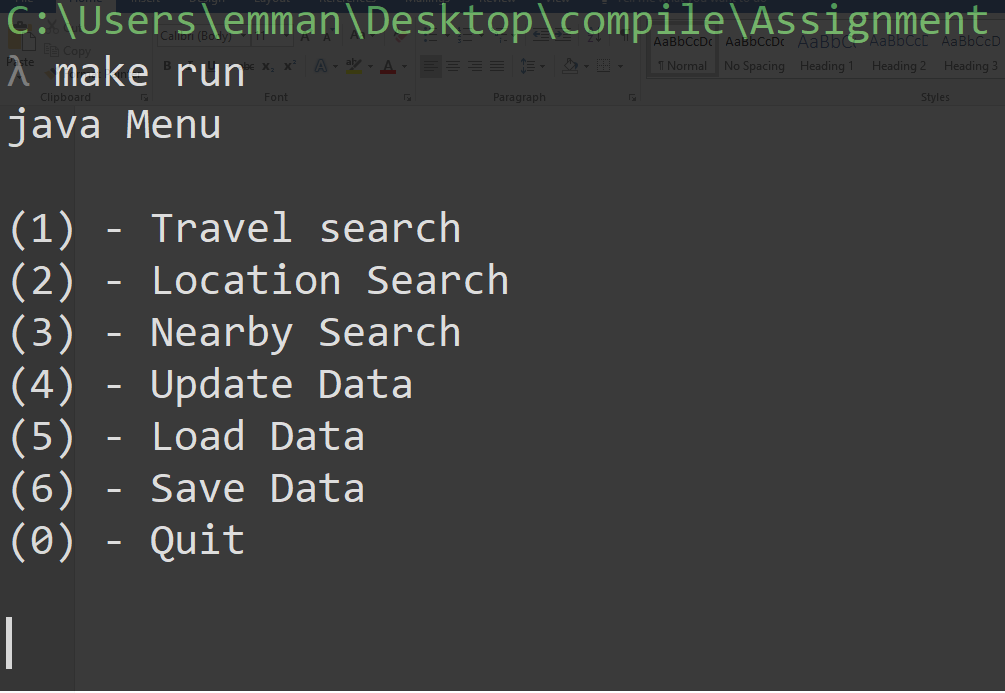


Figure : Menu

After typing this the user will be shown the main menu screen where they can choose one of the following options; Travel Search, Location Search, Nearby Search, Update Data, Load Data, Save Data or Quit. No other options will run, except Quit, unless a file has been susses fully loaded. Carefully follow the instructions displayed to the terminal, and enjoy.

## Testing

For the testing of the program Juint was used, just compile the file “UnitTest.java” using javac, then run the class file using junit, as see below.

A close up of a logo

Description generated with very high confidence

Figure : Junit testing

Testing of everything was not possible, so some of the testing will have to be while the user uses the program.

# Class descriptions and decision

## Place.java

This is an abstract container class that stores and manages the name, abbreviated name, area, population and population reference of the countries and states.

This class was kept quite bare as it isn’t used much throughout the program, the only method I felt it needed were the default and alternate constructors, getters and setters for the name and abbreviated names and validation for area and population input.

## Country.java

This is a container class extends the Place class, this mean it inherits all its functionally stated in Place.java, as well as language.

Because the Place class handles most of the functionally, the Country class does very little and only has 2 methods, the default and alternate constructors.

## State.java

This is a container class extends the Place class, this mean it inherits all its functionally stated in Place.java, it also has a Country object.

Because the Place class handles most of the functionally, the State class does very little and only has 2 methods, the default and alternate constructors, but, unlike the Country class, it must also initialise the Country class within these methods.

## Location.java

This is a container class that stores and manages the name, Country, State, coordinates and description of a location, Country and State are aggregated.

Although this is an important class used extensively throughout the program, all I really need was the name of the location, so my only methods are; the default and alternate constructors and a getter for the name, no values validation id needed as coordinates have a near infinite range and strings cannot be validated.

## DSALinkedList.java

This is a data structure that is often used in place of an array for storing a collection of data throughout the program, it is made from connected nodes. This data structure was adapted from the lecture notes. This list is doubly linked and doubly ended, the tail node is to allows for a better big O notation when trying to access nodes at the end of the list(removeLast, insertLst etc.), for the scope of the project a doubly linked list is not necessary as traversal is only done in one direction.

This class has 2 private class fields, head and tail, both of weight are aggregated inner classes DSAListNode. A detail of all its methods can found in the UML as well as throughout the .java file, but I will explain the more complex methods in detail here.

#### InsertAt

This method is used to insert a node of the a given point in the graph, it imports a generic value, of the node been inserted, and a count, where the new node will be inserted. First a new node is created with the given value and If the list is empty then the head and tail both point to the new node, if it’s not empty then the list traverse until it reaches the right location in the linked list, this node is called “temp”, the new node is then inserted into the list by setting it previous pointer to “temp”, and the new nodes next pointer is set to the next node of “temp”, after this the node after “temp” has its previous pointer set to the new node and finally the next pointer of “temp” is set to the new node.

#### removeAt

This method is like the insertAt method, but instead of inserting at given node, it finds a node containing the given value, then deletes it from the list. It has 3 main sections, if the value is in the head node, it removes the node at the start, the same goes for the tail with removing last, but if it’s somewhere else in the list a recursive function, recRemove is called.

#### recRemove

This is a recursive function that takes in the current node been checked and the vale that is been searched for. It’s terminating conditions are if the node is equal to null, the end of the list has been reached with the node containing the value not found (an exception is thrown), and if the value is found, the node containing the value is removed by setting it previous and next nodes to point to each other instead of the current node.

#### removeFirst

Although this is a simple method, I get a “NullPointerException” when I try to set the previous pointer of the head to null, so I simply left it out and it seems to run fine, but this could potentially be a problem in special cases.

### DSALinkedListIterator.java

This is a private inner class in DSALinkedList, it is a direct copy of the found in the lecture notes. This class enable the list to be traversed outside it’s class.

### DSAListNode.java

This is a private inner class of the class DSALinkedList, it’s what makes up the list. It’s private class fields are; a generic value and a next and previous pointer to other DSAListNode’s. This is a container class and simply stores and maintains the contents of the node, the classes methods consist of an alternate constructor and getters and setters for each of its class fields.’

## DSAGraph.java

The graph is a data structure that represents the connections between items, in this case nodes. In the scope of this program it is used for a map, and traversing a map. Its private class fields are two linked list of nodes and edges.

For this class I will only be addressing how I implemented Dijkstra’s algorithm. For a full list of the methods refer to the UML diagram or in the .java file.

### Dijkstra’s algorithm

For the implantation of my shortest path and nearby search I implemented Dijkstra’s algorithm, with a handful of adjustments to fit in with the specifications of the program.

C:\Users\emman\Desktop\compile\Assignment\DSAGraph.java A screenshot of a social media post

Description generated with very high confidence

Figure :Dijkstra’s algorithm   
Image source: <https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm>

The image above was the primary source of information for the algorithm, from this I could formulate a way to make implement it with the tool and knowledge I had.

#### shortestPath/ shortestTimePath

##### initial tentative values

For more detail on how the algorithm work look at the above image and the comments in my code, what will be explained here are some of the decision made. To begin with all the tentative values were set to infinity (object.MAX\_VALUE), and the source to infinity, this is to ensure that when traversing the graph, the smallest value is ways put into the node, and the source is set to zero as it has no distance from itself.

##### Finding the smallest tentative value

For finding the current smallest tentative value in the queue I simply used a for loop to compare the vales and created a temporary node to mark the current smallest value. This node was then extracted from the list. After implementing this I realised that I can be done much easier using a minimal priority heap, but my knowledge of heaps was limited and I didn’t want to risk the functionally of my code.

##### Peak/mode/nearby

This are things the user can use to alter the behaviour of the search. Peak means that the peak time was taken from the edge and not normal time. Mode is weather all weights are considered for the shortest path of just one. Nearby ignores the path and simply updates the tentative values of all nodes from the source. Refer to the comments in the .java file for more information.

##### Redundancy

These 2 methods and even some of functionality seem to have a lot of redundancy, given more time, all section that have a similar process should be combine and possible put in a separate method, this would also help with the readability of the code.

### DSAGraphNode.java

This is a container class that manages each node in the graph, its’s class fields are; label, tentative distance and time (both used for Dijkstra’s algorithm to keep track of the current minimal distance from the source), previous pointer to another graph node (used in Dijkstra’s algorithm for to find the path), links (all the other graph nodes this node is connected/adjacent to), edges (a list of all the edges of the node) and visited (used in Dijkstra’s algorithm to prevent an endless loop).

Most of this class are getting and setting, but there are some methods specific to Dijkstra’s algorithm. For a full list of the methods refer to the UML diagram or in the .java file.

#### getTentConvertedTime

refer to getConvertedTime/getConvertedPkTime for full details on this method.

#### getEdge

this method imports the label of an edge, the list of edges it iterated until the label matches on of those in the list.

### DSAGraphEdge.java

This is a container class for the edges between the nodes of the graph class, it’s an important class for a lot of the functionally in the program, it contains the following private class fields; label (the name of the edge), from and to (the graph nodes that the edge connects) and weights (a linked list of the different modes of transport the edge contains). A constant, MODES, which is an array of Strings, is the name of all the possible modes for easier input validation.

As with all classes, only the method that were necessary for the functionality of the program were implemented, a full list if the implemented methods can be found in the UML diagram or in the .java file.

#### findWeight

This method is used to get the DSAEdgeWeight in the edge of a given mode, it simply iterates through the list of DSAEdgeWeight until the mode matching the imported mode is found, it none it isn’t found, null is returned. This method can be optimised by replacing the for each loop with an iterator and a while loop, so that as soon as the desired node is found, the loop will be exited.

#### getShortestTime

This is a method to obtain the weight with the lowest time. The method imports a Boolean of peak, if it’s peak or not, based on this import the shortest time is found from the list of weights using a for each loop, that edge class is then exported. getShortestLength is very similar, but it isn’t effected by peak.

### DSAEdgeWeight.java

This is a container class that hold information about the weight of an edge, it stores the mode of transport, time of trip, length of trip, pkTime(time of trip in peak) and the impendence of the edge. Although this is a short class it us very important during a lot of operations in the program, for this reason I didn’t make it a private inner class of DSAGraphEdge.

For the full list of methods in this class please refer to the UML diagram or the .java file, but below will be the methods that may need some extra explanation, not all getters and setters were included as some were not needed in the program.

#### Alternate Contractor

Due to the nature of some entries in the distances files not having a peak time, therefore pkTime = time, the FileIO class sets entries peak time to 0, as it’s not possible to have a travel time of just 0(setting it to Integer.MAX\_VALUE, would have had the same effect), therefore in this class if the peak time is 0, its peak time is simply reset to the actual time. The default impedance is also set to 0, this means there is no effect on the travel time.

#### getTime / getPkTime

as already stated in the previous section, when there is an impedance on the edge, both the time and the peak time. If the impedance is 100% (1.0), the time is returned as infinity, meaning that path is untravellable, but from 99 – 0(0.99 – 0.0) it’s a linear progression. The equation used for this operation was: , The exported value is then returned.

#### getConvertedTime/getConvertedPkTime

The time is read in from the file as hours and minutes, but because storing it as both, would add unwanted complexity to other calculations in the program, time is simply stored as minutes. When the time is being displayed it would be more appropriate to have it in the same format as the input, so these functions simply convert the minutes to hours and minutes and exports it as an array to be used elsewhere in the program, most likely to be displayed to the user.

## FileIO.java

This is the class that handles all input and output of both csv and serial files (NOTE: writing to csv is not yet implemented). This is what the menu replies on too get its information to function. Details on these processes are in the .java file

## Menu.java

This is the part of the program that handles everything, it is where all the needed functions are called, and where a few key methods are located.

#### main

This is where the main menu is located, this section will infinitely loop unless the exit condition is met, from here the user can access all the functionality of the program, give a serial or csv file has been loaded.

#### locationSearch

This method is used throughout this class to search for and find a location. It keeps looping until the user has found their location, or want to stop searching. It uses a for each loop and the Pattern and Matter object to display options based on what the user inputs, it puts these options into an array for the user to select which location they would like.

#### nearby

This method is that displays all the locations within the range of a given location. The locationSearch function is called to select a location, if nothing is found the method is closed, else it calls the shortestPath method from the map. This sets the tentative values of each node to the minimal from the source, a for each loop is then run to display all the location within the range.

#### travelSearch

this displays to the user the most efficient path from one node to another given a few circumstances. These include; time/distance, peak/non- peak time and modes. Because of these options, I feel the method is longer than it should be a could have some redundancy, given more time, some of the functionally can be extended to another method and called rather then been typed twice. This first part of this method is getting user inputs for the two locations, if one of these 2 inputs are invalid the method returns to the menu, else a few more question regarding peak time and mode of transport are asked, based on these the appropriate shortest path method is called, as well as the path been displayed based on those inputs.

#### updateData

this method prompts the user too input 2 nodes as well as a value of impairment for a given mode of transport. The method first asks the user to input a location, based on this location, all the adjacent node of the node are displayed with an index for the user to select, once that’s done a mode of transport is to be selected and the impairment inputted.

# A close up of a map Description generated with very high confidenceUML Diagram

Figure :UML diagram