Kai Li Shi
19157364

Documentation

LOCATIONGRAPH

LocationGraph is a program that hold information on locations and the paths that exist between these locations. It uses a graph data structure to store all location nodes and edges. The program reads information from two csv files to build the graph, using many different data structures and algorithms to allow the user to interact with and use the information stored in the graph. The graph can also be serialized and deserialized.

COUNTRY

Country stores the name, short name, language, area, population and population reference of the countries listed in the locations csv file. When the location file is read in, all country objects are created first using the second alternate constructor in country class, and are inserted into a binary search tree of countries, where the name is the key. Using a binary search tree allows all countries to be stored quickly in an ordered way O(log N), where an extra sorting algorithm is not required. This allows fast searching (also O(log N)), which is useful when creating state and location objects. Country objects are read in and created first since they are used as a class field in state and location objects. Country objects are matched to their states and location via the matchCountry() method inside the class FileIO. This method uses the state or location's country name and searches through the binary search tree of countries to find the corresponding country object. This country object is then returned to be used to initialise a class field in state or location. If no existing country is found in the tree, then a new country object will be created using the name from the location or state's country name in the csv file. This new object is created using first alternate constructor in country class and is then inserted into the binary search tree. Although countries are not used by the user in the program, they are stored as country objects instead of strings so that other information such as the country's population and area would be available to the user if it is needed.

STATE

State stores the name, country, short name, area, area unit, population and population reference of the states listed in the locations csv file. The locations csv file is read a second, time where all states objects are created in a similar way to country objects. States are created using the second alternate constructor and are stored in a binary search tree of state objects. Each state object is used to initialise a class field in location objects. States are matched to their locations via the matchState() method inside the class FileIO. This method uses the name of the location's state and searches through the binary search tree of states for the corresponding state object. This object is then returned to be used to initialise a class field in location. If no matching state is found in the tree of states, then a new object will be created using the read in state name from the csv file. This new object is created using the first alternate constructor in state class and is then inserted into the binary search tree. Although states are not used by the user in the program, they are stored as objects instead of strings so that other information such as short name and area would be available to the user if it is needed.

Although state and country have many fields in common (name, short name, area, population, population reference + all related getters), I did not decide to create a parent class and have state and country inherit from it since it would only save a few lines of code.

LOCATION

Location stores the name, state, country, coordinates and description of the locations listed in the locations csv file. Location objects are created last and are also stored in a similar way to states and countries. They are created using the alternate constructor and are inserted into a binary search tree of locations, where the location name is the key. This allows all locations to be stored quickly (O(log N)) in an ordered way, where an extra sorting algorithm is not required. This also allows fast searching (also O(log N)), which is useful when implementing Location Search. The location's state name and country name are matched to an existing state and country object, using matchState() and matchCountry() in FileIO. These matching objects are used to initialise the state and country fields inside location class. If no matching state or country is found, then a new one is created and is inserted into the corresponding binary search tree, and is then used to initialise the field.

State and country class fields were used in location class instead of having a list of locations inside state and a list of states inside country. This was chosen because I didn't see any reason for the program needing a list of locations within a given state, or a list of states within a given country. A given location's country or state would be needed to be accessed however, for example when giving the Travel Detail Report to the user. The state and country name is used to determine whether the start and end locations are in different states/countries, and this information is used to output whether the user will need a passport to travel or not.

DSABINARYSEARCHTREE

A binary search tree was used to store all country, state and location objects. A tree was chosen since all elements are fast to access and are inserted in sorted order, so a new sorting algorithm did not have to be implemented if we needed the objects to be in order. Inserting elements in a binary tree is also a lot faster than algorithms to sort the elements (O(log N) for inserting compared to $O(N^2)$ and O(N log N) for sorting). If a list of sorted objects is needed, a queue in sorted order can easily be exported and used. It is also assumed that all countries, states and locations are unique, so a binary search tree would be good to store them in since we do not want multiple copies of the same object if they exist in the location file. The worst-case scenario for a binary search tree is that it turns into a linked list, which is the data structure that I would have used if I did not use a binary search tree, since it can easily grow in size. Inserting and finding in a linked list would be O(N), which is the same as an array. I would not have used an array however since it has a fixed size, and allocating a larger than needed array size would be a waste of space, and an extra sorting algorithm would be needed if we want a sorted list.

The binary search tree of locations is used when the user selects the Location Search option. Some locations do not appear in the graph, since they do not have any edges connected to it. Using a binary search tree of locations will allow the user to search for all locations, whether they can be travelled to or not. The tree lets locations to be traversed easily, as the nodes can be exported as queue.

The DSABinarySearchTree, DSATreeNode, UnitTestDSABinarySearchTree and UnitTestDSATreeNode classes were previously submitted for practical 5 but have been edited since submission. The methods in DSABinarySearchTree and DSATreeNode are based on the lecture slides.

DSAGRAPH, DSAGRAPHNODE, DSAGRAPHEDGE

A graph was used to store the locations and distances, with the locations being the graph nodes and the distances being the edges between the nodes. It contains the private class fields vertices (a linked list of graph nodes) and edges (a linked list of graph edges).

A graph was used because it seemed like the most logical data structure to use, since we could store both the locations as well as the connections between those locations. DSAGraph does not use generics since the graph will only be a graph of locations and will not be used for anything else in this assignment. DSAGraph includes all normal methods (as specified in the lecture notes) and as well as variations on these methods, such as dfsDistance() and dfsTime(). These two methods use a depth first search approach to find all location nodes that can be reached within a specific time or distance, and is used for the Nearby Search option in the program. There are also multiple getter methods for retrieving a specific edge in the graph (getEdge(), getNewEdge() used for Nearby Search, getNewEdgeWithMode() used for Update Data and getEdgeWithDistance() used for Travel Search). Since edge labels were not unique I couldn't find a way to retrieve one specific edge based only on the label name so multiple methods were created. However, I'm not sure if making multiple methods that do almost the same thing was the best way to solve this problem.

DSAGraphEdge objects were created when reading the distances csv file. This class stores the label, distance, mode, time, peak time, start vertex, end vertex and visited fields. The time and peak time were stored as the time in seconds, since this was the only way I could think of storing the time in a way that could allow two times to be compared easily (not as a string) and with high accuracy. This time in seconds was used when implementing Nearby Search, as numbers are easily comparable and manipulable. When outputting the time back to the user it is converted back into minutes, or in a string formatted as hh:mm:ss (used in Travel Search). This is the purpose of the convertFromSeconds() method inside DSAGraph class.

DSAGraphNode class stores the label, value, links, edge list and visited. The two additional class fields, distance from source and previous, were added to help store information when finding the shortest path between two location nodes. Location graph nodes are created when they appear in the distances csv file. When the file is read in, an existing location object that is stored in the binary search tree is used to create a graph node. This means that if a location (eg Geraldton) appears in the location csv file and not the distances csv file, then it will only exist in the tree and not the graph. Storing the locations in both will help perform the different tasks that the user may choose to do.

The method shortPath() in DSAGraph uses Dijkstra's algorithm was to find the shortest algorithm between two edges. I chose to use this algorithm over others since it was a simple and well known algorithm. Used in travel search, this method finds the shortest path between two nodes based on the distances along edges in between. After the shortest distance has been found, the locations along this path are added to a stack, then are popped off and added to a string to be exported in the correct order. The extra methods

getVertexFromTreeWithMinDist() and getEdgeWithDistance() were added to assist this algorithm.

Dijkstra's algorithm works by lowering the distanceFromSource field in DSAGraphNode with the shortest distance that has been found to that location so far. The private class field previous, inside DSAGraphNode, is used to record the previous node that was visited, in order to reach that location. Once the algorithm has finished and the destination has been marked as visited, the shortest path can be found by following all previous node fields, starting from the destination. My method shortPath() was adapted from the Java code from Lafore, "Data Structures & Algorithms" (2003) and the pseudocode from Wikipedia, https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=805572049 (accessed October 16 2017).

The classes DSAGraph, DSAGraphNode, DSAGraphEdge and their test harnesses were previously submitted for practical 6, but have been heavily edited since then to further suit the specifications of the assignment. Most of the methods in these classes are based on the pseudocode from the lecture slides.

DSASTACK, DSAQUEUE

DSAStack and DSAQueue were implemented using a linked list instead of an array. This made it easy to implement the stack and queue operations since they could be replaced with corresponding linked list methods that were already implement. This is useful since you wouldn't have to keep a count of how many elements there are in the stack/queue, and they can shrink and grow as required. There isn't any memory overhead wasted with allocated array space that hasn't been used.

A stack was used in the methods dfsTime() and dfsDistance() to find locations within a specified distance or time. Locations are pushed onto the stack if they are still within this limit, and are popped off when they don't have an unvisited vertex in their adjacency list. A stack was used instead of a queue so that only locations that were within and slightly more than the search boundary are searched. Since we do not traverse the whole graph, the time used to search is slightly shorter that using the normal depth first search algorithm or breadth first search. However, in terms of time complexity, constants are not taken into consideration so all searches will still take the same amount of time.

A queue was used in DSABinarySearchTree to export a list of all locations. It was used so that the locations would be added and removed/used from the list in the correct order. If a stack was used, then the locations would be in reverse order if they were popped off the stack to be used.

DSAStack and DSAQueue (implemented with a linked list) were previously submitted for practical 4 and are based on the pseudocode from the lecture slides.

DSALINKEDLIST

A linked list was used in DSAGraph instead of an array to hold the list of edges and list of vertices because it does not have any limits on the size. The number of edges and nodes are unknown when reading in the csv, so we could not give a good estimation for the size of the array. If an array was used then generics would not be possible, which is needed in the list.

DSALinkedList has the private inner classes DSAListNode and DSALinkedListIterator. These private classes were used to promote information hiding. The iterator cannot be exposed externally so it was implemented as a private class. I did not implement a sorting algorithm for the linked list since it wasn't needed and the binary search trees that held all the Country, State and Location classes were already sorted.

DSALinkedList was previously submitted for practical 4 and is based on the pseudocode from the lecture slides.

FILEIO

FileIO class stores all methods relating to the reading, writing, serializing and deserializing of objects and files. The methods processCountry(), processState() and processLocation() read the locations csv file and creates the country, state and location objects that are inserted into binary search trees. These methods call the private methods matchCountry() and matchState() as described previously, where existing state and country objects are matched to their locations and states to be used to initialise the class fields. The method processDistances() reads the distances csv file to create all the edges between the locations. It also retrieves locations in the location tree and stores them in the graph as location graph nodes.

The method writeReport() is called when the user wants to write a report to a text file. It imports the strings that are to be written to file, as well as the file name. All reports are written to files called 'TravelSearch.txt', 'TravelReport.txt', 'LocationSearch.txt' and 'NearbySearch.txt.

The methods load() and save() are used to load and save the graph object only. If the user loads the graph from the file, then location search will not work since it traverses the binary search tree of locations to find the locations, rather than traversing the graph. If the user would like to use location search then they will need to build the graph from the csv files, as it creates all country, state and location trees as well as the location graph.

MENU

The Menu class interacts with the user, outputting information based on user input. Each menu option has been placed in its own method. Menu options 1-4 (inclusive) will not work unless the user has selected option 5, to load information from the csv files or serialized file. The menu's corresponding methods are travelSearch(), locationSearch(), nearbySearch(), updateData(), loadDistances(), loadSer() and saveSer().

Other methods that have been used are validateInput() and askWriteReport(). Since all user input is a number to represent the option that they have selected, the validateInput() method imports the lower and upper bound and will repeatedly ask the user to enter an option until it is valid. This option is then exported to the calling method. The methods menu(), travelSearch(), nearbySearch(), updateData() and askWriteReport() use this private method to validate the user input.

The method askWriteReport() asks the user whether they would like to save the reports to a file. It imports the file name, header and information (all as strings), and writes the header

and information to a text file. The Travel Search, Travel Detail, Location Search and Update Data reports are written using this method.

TRAVEL SEARCH

The user is asked for the start and end locations. If any of the input values are not found in the graph, then the user is asked to enter the name of the location again until it is valid. The user is then asked whether it is peak hour or not, and whether they would like a Travel Search or Travel Report. The travel search will output the travel locations that are in the route between the start and end location. This includes the location names, running total distance, running total time and travel mode. If peak time is selected, then the peak time will be used to calculate the running total time if the travel mode is car. This information is found using the shortPath() algorithm in DSAGraph. If the Travel Report option is selected, then all information in the Travel Search, as well as information on the start and end locations will be shown to the user. If the start location's state and the end location's state are not the same, then the program will also tell the user that they may need a passport to travel. Finally, the program will ask the user if they would like to save this information to a file. If the user selects yes, the header (title of the file eg "Travel Information From <startVertex> to <endVertex>") and the results (all travel information) will be saved to a text file. The Travel Search information will be saved to a file called "TravelSearch.txt" and the Travel Report information will be saved to a file called "TravelReport.txt".

A change that I would have made to this algorithm is checking whether the edge was traversable. If the user updates the impairment time to 100% for all edges between two specific nodes, then the travel search should say that there doesn't exist a path between the two locations. Inside the shortPath() algorithm I would have to first check whether the time along the edge is not equal to 0 (if update data was 100% then I set the travel time to 0).

LOCATION SEARCH

If location search is selected then the user will be asked to input a location name. All locations that start with the entered string will be displayed to the user. User input is case insensitive, so if 'Per' and 'per' were entered then they would yield the same results. A queue of all locations in the binary search tree of location is used to output all matching locations. The binary search tree is traversed instead of the graph because some locations such as Geraldton do not have any edges going to or from it. This means that the locations that exist in the tree may not exist in the graph. The user will be able to search for all locations, whether they exist in the graph or not. The total number of locations found starting with the entered string is also recorded and displayed to the user. If no matches are found then the message "No matches were found" is displayed. The user will then be asked whether they would like to save this information to a file. Like Travel Search, the header ("Locations starting with '<stubstring>'") and results (list of locations starting with the substring + the number of results found) will be saved to a text file called "LocationSearch.txt".

NEARBY SEARCH

Nearby Search allows the user to find locations within a specified distance or time. The user will be prompted to input the name of the start location, and whether they would like to find locations within a distance or time. The distance will be entered in kilometres and the time will be entered in minutes. The corresponding method, dfsTime() or dfsDistance(), will be

called and the results will be returned as a string. The names of all locations will be outputted to the user in a list. The methods dfsTime() and dfsDistance(), found in DSAGraph, use a depth first search approach for traversing the graph and finding all the relevant locations. As with the previous two options, the results of Nearby Search can also be saved to a text file called "NearbySearch.txt".

UPDATE DATA

The Update Data option allows the user to enter an impairment percentage (as an integer), that will change the total travel time of a specific edge between two locations. The user will be prompted to enter the name of two locations on the graph. If a single edge does not exist between the two locations, or the start and end locations are the same, then an error message will be shown. Otherwise, the user will be asked to enter a transportation mode, and the travel time will be updated for this edge.

The formula used for updating the time is:

edgeTime * 100/(100-impairmentPercentage)

where impairmentPercentage is an integer between 0 and 100.

If the impairment percentage entered is 50(%), then the travel time will double. If the impairment percentage is 100(%), then the travel time will be set to 0, meaning that the edge cannot be traversed. The update data will affect the Nearby Search option. If all edges between two locations are set to 100%, then the Nearby Search will not show the other location since it cannot be traversed. This should also affect the Travel Search option, but I did not check that the time was more than 0 when implementing the algorithm, so this is something that I would have changed.

LOAD DATA

The load data option allows the user to select between loading information from the csv files or a serialised file. If the csv file option is entered, then the user will be prompted to enter the name of the location csv file and distances csv file. If either file does not exist, then an exception is thrown and the error message shown.

If the user chooses to load from the serialised file then the function will automatically deserialize the graph object from the file "locationGraph.ser". If the serialised file does not yet exist (hasn't been saved), then an error message will be shown. Otherwise, the serialised file will only create the graph and not the binary search trees of country, state and location. This will affect the Location Search option, since it traverses through the tree of locations to find all matching locations. Since this tree is not saved, then this option will not work. A better way to implement this option would be to save all trees as well as the graph.

SAVE DATA

The user may decide to serialise the location graph. The object will automatically be saved to "locationGraph.ser".

Kai Li Shi
19157364

References

Lafore, Robert. 2003. Data Structures & Algorithms in Java. Indiana. Sams Publishing

Wikipedia contributors. "Dijkstra's algorithm". Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Dijkstra%27s_algorithm&oldid=805572049 (accessed October 16, 2017)