Curtin College, Bentley

Trimester 3, 2024

Final Assessment:

Traffic Management Program

Mamert Vonn G. Santelices

90026174

Introduction

The project aims to build a traffic management system that optimizes traffic flow in urban areas to alleviate and enhance transportation efficiency. So, it performs the following by utilizing several algorithms and data structures to process the raw data within the software. Moreover, it converts a csv file or a text file using ',' as delimiters to a data structure, such as a graph network and a weighted hash table of the traffic data. Furthermore, the mentioned abstract data structure include algorithm: like, Dijkstra's algorithm implemented with Breadth First Search to find best path given a source location to every other road; and Depth First Search to find every other path from a point to another point. Additionally, traffic data can be sorted through heaps to prioritise intersections with higher traffic volume or congestions. Hence, the program is used to extrapolate and infer data to optimize traffic flow. Also, the project provides a user-friendly command-line interface to create simulations testing ideas in dynamically managing traffic. In conclusion, analysing the interpreted data leads to deducing possible traffic management strategies and deploying of traffic control measures.

Terminologies

Abbreviation	Term	Description
	Traffic Network	Refers to the use of a Graph Matrix to represent the road intersections
	Traffic Data	Refers to a structure storing the road intersection's distance in miles, traffic volume, road capacity, and congestion level, typically store in a Hashmap.
	Traffic Flow	Refers to a list of Traffic Data that specify a source road to a destination road. It may also store the total weight of the entire path.
BFS	Breadth First Search	is an algorithm for searching a tree data structure for a node that satisfies a given property
DFS	Depth First Search	is an algorithm for searching a tree data structure
	GraphMatrix	An abstract data structure, that specifically is an undirected graph represented connection between nodes
	Dijkstra's Algorithm	Is an algorithm that finds the shortest path between a given node and all other nodes in a graph using the weights of the edges to find the path that minimizes the total distance.
	Weight	May refer to the distance, traffic volume, traffic factor, or a value derive from a formula represents the likelihood of the program to take the path with that specified weight.
	Edge	Refers to the relationship between two graph nodes
	HashMap	Is a key value pair data structure that uses hashing to access elements
	Heap	Is referring to the data structure that behaves like a queue but sorted based on the assigned priority of the value

Dependencies

Requires the following:

- Linux OS
- openjdk 19.0.2
- OpenJDK Runtime Environment
- OpenJDK 64-Bit Server VM

Traffic Menu.java and TestHarness.java uses TrafficNetwork.java, which relies on traffic_network.txt and traffic_data.txt within the project directory.

Project uses the java CustomTypes package for its data structures and java CustomTypes.CustomException package as the program exceptions.

Methodology

A Test Harness can be run through the command “java TestHarness.java <options>” to ensure that the program is completing all the required assertions. For example, a possible assertion could throw a IncorrectShortestPathError, RetrieveNoneExistingpathError, and PrioritiseTrafficControlError, on which it uses the default traffic_network.txt and traffic_data.txt data as a test sample. Furthermore, additional flags include the following:

Flags	Description
-q	Quite doesn’t display an output but only prints failed assertions for debugging
-d	Display road intersections in detail along with the traffic data
-l	Displays traffic network adjacency list
-p	Displays sorted traffic networks base on the highest to the lowest traffic volume
-s	Displays best path from all roads to every other road based on the weighted formula of (distance*congestion lvl*traffic vol/road cap) using BFS and Dijkstra
-sm	Displays shortest path from all roads to every other road using BFS and Dijkstra
-st	Displays best path from all roads to every other road based on the weighted formula of (congestion lvl*traffic vol/road cap) using BFS and Dijkstra
-sv	Displays best path from all roads to every other road based on the traffic volume using BFS and Dijkstra
-a	When using the -s<optional> flag it prints all the other nonoptimized path from a source to a destination using DFS and asserts by comparing best path and ensuring it is the best path from all the other path

Also, "bash sweep.sh" can be ran to sweep the Test Harness to output text files of the network adjacency list, the shortest path of every path to every other path, and the best path to take based on the traffic_data, additionally it outputs the priority queue based on the intersection with the most traffic volume. All the output is stored in the ./docs directory, and within the ./docs directory running “bash find_diff.sh” outputs a path_difference.txt of the difference in path of the best path and the shortest path.

The user-friendly console interface Traffic Management Program run through the command "java TrafficMenu.java", include features of:

1. Display Traffic Network Adjacency List
2. Find Path from Two Locations
3. Display Intersection in order of Traffic Volume
4. Add New Road
5. Add New Intersection
6. Toggle Display Traffic Data in Detail

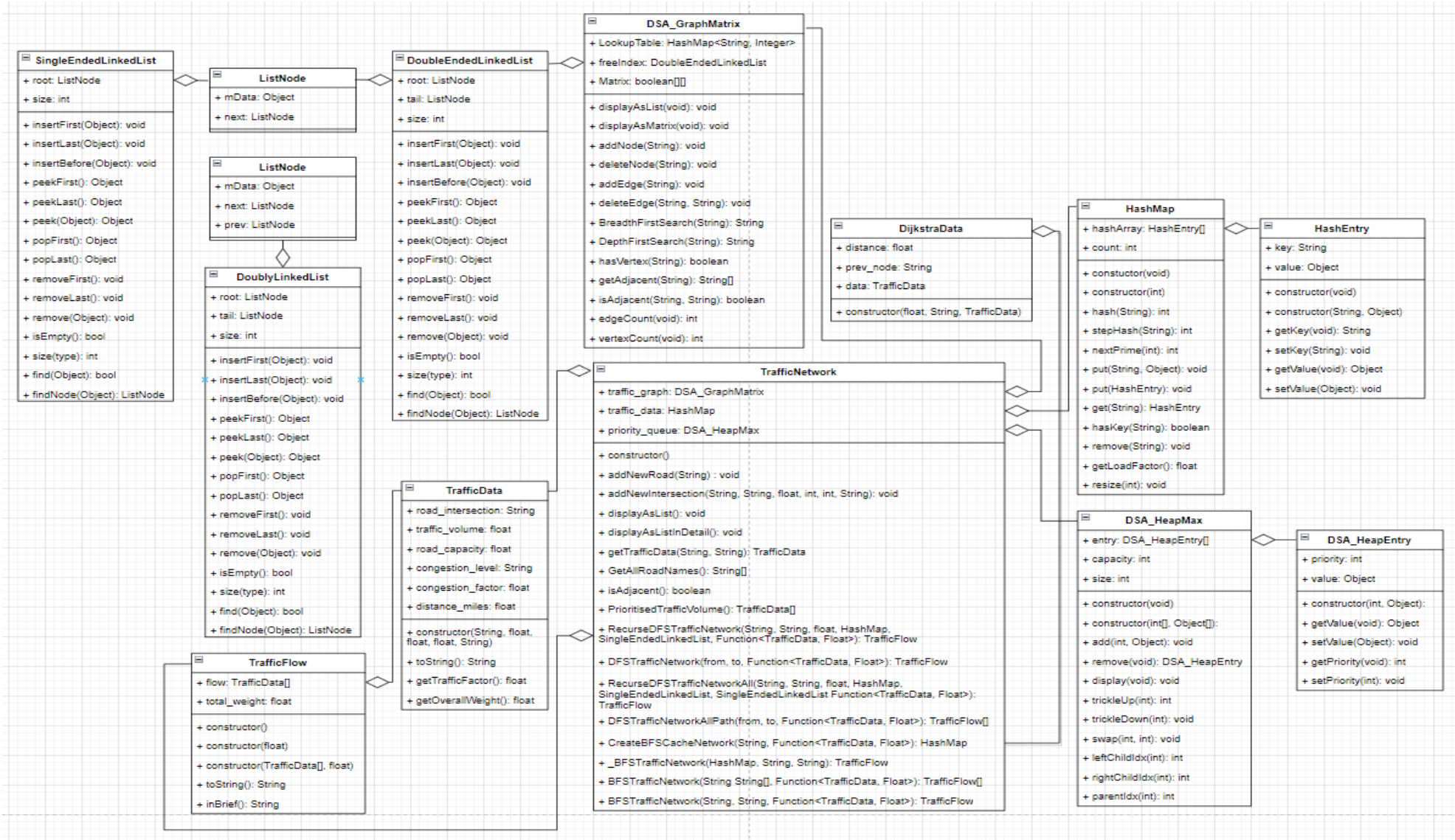
Time Complexity Analysis

When using string values as an index to store traffic data, using an array to access an element is $O(\mathrm{n})$, since string comparisons are done for each element. Similarly for linked list, it would be $O(\mathrm{n})$ with the downside of more memory calls when transversing fragmented memory. On the other hand, HashMaps has $O(1)$ access due to hashing, but in the worse case it would be $O(\mathrm{n})$ when entries form clusters. Instead, techniques like linear probing, quadratic probing, or double hashing can reduce clustering.

A weighted DFS potentially is $O(n^2)$ when all nodes are connected to all other nodes, or generally in situations where there are plenty of forks within the nodes. Since, a DFS visits every possible combination of a path from a source to a destination, and even at times visits a segment several times. Similarly, a BFS that implements Dijkstra’s algorithm at a worst case is $O(n^2)$ if the shortest path is frequently modified within the Dijkstra routing table, since every change of shortest path requires pushing already visited nodes back to the queue. But, in best case scenario, it is $O(\mathrm{n})$. Furthermore, implementing a min-heap priority queue drops it down to $O(\mathrm{V} + \mathrm{E} \log V)$ wherein V is the number of vertex and E is the number of edges.

BFS and DFS without weights should be $O(\mathrm{V} + \mathrm{E})$.

UML Diagram


SingleEndedLinkedList
+ root: ListNode
+ size: int
+ insertFirst(Object): void
+ insertLast(Object): void
+ insertBefore(Object): void
+ peekFirst(): Object
+ peekLast(): Object
+ peek(Object): Object
+ popFirst(): Object
+ popLast(): Object
+ removeFirst(): void
+ removeLast(): void
+ remove(Object): void
+ isEmpty(): bool
+ size(type): int
+ find(Object): bool
+ findNode(Object): ListNode
ListNode
+ mData: Object
+ next: ListNode
ListNode
+ mData: Object
+ next: ListNode
+ prev: ListNode
DoublyLinkedList
+ root: ListNode
+ tail: ListNode
+ size: int
+ insertFirst(Object): void
+ insertLast(Object): void
+ insertBefore(Object): void
+ peekFirst(): Object
+ peekLast(): Object
+ peek(Object): Object
+ popFirst(): Object
+ popLast(): Object
+ removeFirst(): void
+ removeLast(): void
+ remove(Object): void
+ isEmpty(): bool
+ size(type): int
+ find(Object): bool
+ findNode(Object): ListNode
DoubleEndedLinkedList
+ root: ListNode
+ tail: ListNode
+ size: int
+ insertFirst(Object): void
+ insertLast(Object): void
+ insertBefore(Object): void
+ peekFirst(): Object
+ peekLast(): Object
+ peek(Object): Object
+ popFirst(): Object
+ popLast(): Object
+ removeFirst(): void
+ removeLast(): void
+ remove(Object): void
+ isEmpty(): bool
+ size(type): int
+ find(Object): bool
+ findNode(Object): ListNode
DSA_GraphMatrix
+ LookupTable: HashMap<String, Integer>
+ freeIndex: DoubleEndedLinkedList
+ Matrix: boolean[][]
+ displayAsList(void): void
+ displayAsMatrix(void): void
+ addNode(String): void
+ deleteNode(String): void
+ addEdge(String): void
+ deleteEdge(String, String): void
+ BreadthFirstSearch(String): String
+ DepthFirstSearch(String): String
+ hasVertex(String): boolean
+ getAdjacent(String): String[]
+ isAdjacent(String, String): boolean
+ edgeCount(void): int
+ vertexCount(void): int
DijkstraData
+ distance: float
+ prev_node: String
+ data: TrafficData
+ constructor(float, String, TrafficData)
HashMap
+ hashArray: HashEntry[]
+ count: int
+ constuctor(void)
+ constructor(int)
+ hash(String): int
+ stepHash(String): int
+ nextPrime(int): int
+ put(String, Object): void
+ put(HashEntry): void
+ get(String): HashEntry
+ hasKey(String): boolean
+ remove(String): void
+ getLoadFactor(): float
+ resize(int): void
HashEntry
+ key: String
+ value: Object
+ constructor(void)
+ constructor(String, Object)
+ getKey(void): String
+ setKey(String): void
+ getValue(void): Object
+ setValue(Object): void
TrafficNetwork
+ traffic_graph: DSA_GraphMatrix
+ traffic_data: HashMap
+ priority_queue: DSA_HeapMax
+ constructor()
+ addNewRoad(String) : void
+ addNewIntersection(String, String, float, int, int, String): void
+ displayAsList(): void
+ displayAsListInDetail(): void
+ getTrafficData(String, String): TrafficData
+ GetAllRoadNames(): String[]
+ isAdjacent(): boolean
+ PrioritisedTrafficVolume(): TrafficData[]
+ RecurseDFSTrafficNetwork(String, String, float, HashMap, SingleEndedLinkedList, Function<TrafficData, Float>): TrafficFlow
+ DFSTrafficNetwork(from, to, Function<TrafficData, Float>): TrafficFlow
+ RecurseDFSTrafficNetworkAll(String, String, float, HashMap, SingleEndedLinkedList, SingleEndedLinkedList Function<TrafficData, Float>): TrafficFlow
+ DFSTrafficNetworkAllPath(from, to, Function<TrafficData, Float>): TrafficFlow[]
+ CreateBFSCacheNetwork(String, Function<TrafficData, Float>): HashMap
+ _BFSTrafficNetwork(HashMap, String, String): TrafficFlow
+ BFSTrafficNetwork(String String[], Function<TrafficData, Float>): TrafficFlow[]
+ BFSTrafficNetwork(String, String, Function<TrafficData, Float>): TrafficFlow
TrafficData
+ road_intersection: String
+ traffic_volume: float
+ road_capacity: float
+ congestion_level: String
+ congestion_factor: float
+ distance_miles: float
+ constructor(String, float, float, float, String)
+ toString(): String
+ getTrafficFactor(): float
+ getOverallWeight(): float
TrafficFlow
+ flow: TrafficData[]
+ total_weight: float
+ constructor()
+ constructor(float)
+ constructor(TrafficData[], float)
+ toString(): String
+ inBrief(): String
DSA_HeapMax
+ entry: DSA_HeapEntry[]
+ capacity: int
+ size: int
+ constructor(void)
+ constructor(int[], Object[]):
+ add(int, Object): void
+ remove(void): DSA_HeapEntry
+ display(void): void
+ trickleUp(int): int
+ trickleDown(int): void
+ swap(int, int): void
+ leftChildIdx(int): int
+ rightChildIdx(int): int
+ parentIdx(int): int
DSA_HeapEntry
+ priority: int
+ value: Object
+ constructor(int, Object):
+ getValue(void): Object
+ setValue(Object): void
+ getPriority(void): int
+ setPriority(int): void

Requirement

Req. ID	Requirement	Justification	Test Case	Test Result
1	Test printing shortest path for all roads to every other road. *TestHarness -sm -a*	It is important to be able to compare that the shortest path to the best path based on traffic and distance	**Test 1, Test 2**	Pass
2	Test print best path base on a formula.*TestHarness -s -a*	It is important to receive the optimal path using the program and guarantee that the program works.	**Test 1, Test 2**	Pass
3	Test priority of road intersections base on the traffic vol. *TestHarness -p -d*	Control measure must be implemented on high traffic volume intersections first	**Test 3**	Pass
4	Traffic Menu add new road	When simulating a traffic network, we want to be able to easily modify the program's state	**Test 4**	Pass
5	Traffic Menu add new intersection	When simulating a traffic network, we want to be able to introduce new edges to experiment with new road intersections	**Test 4**	Pass
6	Traffic Menu display intersection in order of traffic network	Control measure must be implemented on high traffic volume intersections first	**Test 3**	Pass
7	Find Best/Shortest Path between two roads	Used to dynamically perform Traffic Management	**Test 1, Test 2**	Pass

Test 1: Correct Shortest Path. **Test 2**: Retrieve An Existing Path. **Test Case 3**: Correct Priority of Traffic Volume. **Test 4:** list the correct graph nodes, and their corresponding adjacency list for node.

Conclusion

Given the set of sample data, a noticeable difference between only finding the shortest path to considering the traffic data, shows some path with higher congestion levels are avoided despite being closer to the set destination. For example, looking at the Downtown-Broadway intersection experience higher traffic and generally a longer intersection. Thus, most calculated best path avoids this intersection, and instead goes around the Museum Street, City Hall to Central Park path due the lower congestion level. Despite, this traffic flow having longer travel distance but conjectured to have a shorter travel time than taking the Downtown-Broadway Road. This idea is reinforced by the priority heap suggesting that Downtown-Broadway has one of the highest traffic volumes requiring immediate traffic control measures to mitigate the congestion.

```
4,5c4,5
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall, Broadway-Museum Street]
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall]
---
> [Downtown-Broadway]
> [Downtown-Broadway, Broadway-Museum Street]
9,10c9,10
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall, Museum Street-Lakeside, Park Street-Lakeside]
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall, Museum Street-Lakeside]
---
> [Downtown-Broadway, Broadway-Park Street]
> [Downtown-Broadway, Broadway-Museum Street, Museum Street-Lakeside]
14,16c14,16
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall, Museum Street-Lakeside, Park Street-Lakeside, Park Street-Industrial Zon
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall, Museum Street-Lakeside, Lakeside-Suburban Area]
< [Downtown-Central Park, Central Park-City Hall, Museum Street-City Hall, Museum Street-Lakeside, Lakeside-Suburban Area, Suburban Area-Residentia
---
> [Downtown-Broadway, Broadway-Park Street, Park Street-Industrial Zone]
> [Downtown-Broadway, Broadway-Museum Street, Museum Street-Lakeside, Lakeside-Suburban Area]
> [Downtown-Broadway, Broadway-Museum Street, Museum Street-Lakeside, Lakeside-Suburban Area, Suburban Area-Residential Zone]
19,20c19,20
< [Downtown-Central Park, Downtown-Riverside]
< [Central Park-City Hall, Museum Street-City Hall, Broadway-Museum Street]
```

Image from output of running diff command showing the different path taken if prioritising shorter travel distance, instead of taking the traffic data into account.

```
Displaying Traffic Network Graph as Adjacent List

Downtown => Central Park, Riverside, Broadway,
Central Park => Downtown, University Avenue, City Hall,
University Avenue => Central Park, Riverside, Central Station,
Riverside => Downtown, University Avenue, North Avenue,
Broadway => Downtown, Museum Street, Park Street,
Museum Street => Broadway, City Hall, Lakeside,
City Hall => Central Park, Museum Street, Airport Road,
Central Station => University Avenue, Business District,
North Avenue => Riverside, Business District,
Park Street => Broadway, Lakeside, Industrial Zone,
Lakeside => Museum Street, Park Street, Suburban Area,
Airport Road => City Hall,
Business District => Central Station, North Avenue, Highway Junction,
Highway Junction => Business District,
Industrial Zone => Park Street,
Suburban Area => Lakeside, Residential Zone,
Residential Zone => Suburban Area,
```

The outputted adjacency list also shows a lack of intersection between the Airport Road which is only connected to City which experiences high congestion, similarly to the Highway Junctions. Hence, the roads would tend to be avoided unless if a traffic control measure is employed. Also, despite the industrial zone only experiencing a mild congestion, it should be noted for future developments.

Best Path base on shortest travel distance

Best Path taking into account both travel distance and traffic data.

More data is available in the docs folder of the project directory after running “bash sweep.sh”. In conclusion, the program can interpret raw data into easily analysable data to help infer information about the current traffic flow of urban area, thus assisting in the development of traffic management strategies and traffic control measures for the foreseeable future.