SafeWayz Mobile Application

SE 2XB3: Binding Theory to practice,
CAS Department, McMaster University
Version 1.0
April 12, 2020
Group #: G02

Group Members:
Aditya Sharma
Anando Zaman
Daniel Di Cesare
Jash Mehta
Zackary Ren

Revision Page:

By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

Aditya Sharma:_____ _____
Student Number: 400189221

Anando Zaman:_____ _____
Student Number: 400180288

Daniel Di Cesare:_____ _____
Student Number: 400179249

Jash Mehta:_____ _____
Student Number: 400185652

Zackary Ren:_____ _____
Student Number: 400192420

Report Revision History:

Version	Primary Authors(s)	Description of version	Date completed
Draft Stage 1	Anando Zaman, Daniel Di Cesare	Initial draft with class diagram/description, executive summary, and Module USE relations complete	04/04/2020
Draft Stage 2	Anando Zaman	UML State Diagrams complete. MIS for DFS,BFS, Android Activities complete. Some progress on review of the design section.	04/05/2020
Draft Stage 3	Jash Mehta Anando Zaman	Completed CrimeSorting, CrimeSearching MIS, and added information to the State Diagram and Implementation section. Changes to formatting of document. Added Table of contents, glossary, and references. Cleaned up formatting. Added the problems encountered during the design stage and possible future revisions in the design review section.	04/07/2020
Draft Stage 4	Aditya Sharma	Changes to document format. State Diagrams & Implementation Changes Review & Evaluation redone Completed MIS for	4/07/2020

		DFS Modified runDijkstra and Dijkstra MIS	

Contribution Page:

Name	Role(s)	Contributions
Aditya Sharma	- Project Leader - Tester - Developer	- Cleaned and Merged entire data set - Formatted entire data set from CSV to JSON - Established plan, sprints & checkups for the entire project - Implemented DFS Algorithm - Created an additional modular testing file for DFS/BFS
Anando Zaman	- Researcher - Technical lead - Developer - Tester/Validation	- Researched crime and street datasets -BFS Implementation - Testing of DFS/BFS - Application development and integration(DFS and Dijkstra plotting, hashing, dynamic UI) + Maps API
Daniel Di Cesare	- Designer - Developer - Tester/Validation	- Dijkstra Implementation - Testing of Dijkstra - Integration/porting of eclipse-terminal code into Android Application + Maps API
Jash Mehta	- Designer - Developer - Tester/Validation	- Searching/Sorting Implementation - Design and functionality of Application
Zachary Ren	- Developer	- Addition of weights to Dijkstra

Executive Summary

SafeWayz is an application focused on providing safety-oriented navigation to its users in San Francisco. The application provides routes determined based off of previous crime data trends as well as relative distance. Using the previous crime data, a weighted graph is created to represent all the connecting streets and intersections within the city of San Francisco. Once each intersection is determined, a crime weighting is assigned based off of frequency and severity of crimes at that location. Currently, BFS, DFS, and Dijkstra's algorithm can be run on the graph, with BFS and DFS connecting streets and Dijkstra's connecting between intersections.

Once the route is determined, the passed intersections are sent through the Directions API from google. A path is returned which can be plotted on google maps natively within the application. Users also have the option to view crime weights for a specific street, and also the option to view a sorted list of intersections with the lowest crime ratings.

Table of Contents

Glossary

App: An abbreviation for “application”. It is essentially a software product designed for mobile devices. [1].

API: An abbreviation for “Application Programming Interface”. It is a software product that contains definitions, protocols and written code that is frequently used to allow for modular programming [8].

BFS: An abbreviation for the shortest-path finding algorithm “Breadth-first Search”. This algorithm is generally used to find the shortest-route between two points. [7].

DFS: An abbreviation for the path-finding algorithm “Depth-First Search”. This algorithm is generally used to find any route between two points [10].

Efficient: This term is defined by the time and space complexity of all the algorithms that the software product is composed of. A software product is said to be efficient iff (defined below) the space/ time complexities are at a global minimum [9].

IFF: If and only if

JSON: An abbreviation for the word “JavaScript Object Notation”. This is a format that is used for storing data. It is similar to a hashmap/ hashtable [4].

Optimal Path: It is defined as the safest route between two locations specified by the user.

UI: An abbreviation for the word “User Interface” [3].

UX: An abbreviation for the word “User Experience” [11].

Weighting: Also termed a ‘cost’; impacts the viability of a point within a dataset [6].

Dataset: An organized collection of data [5].

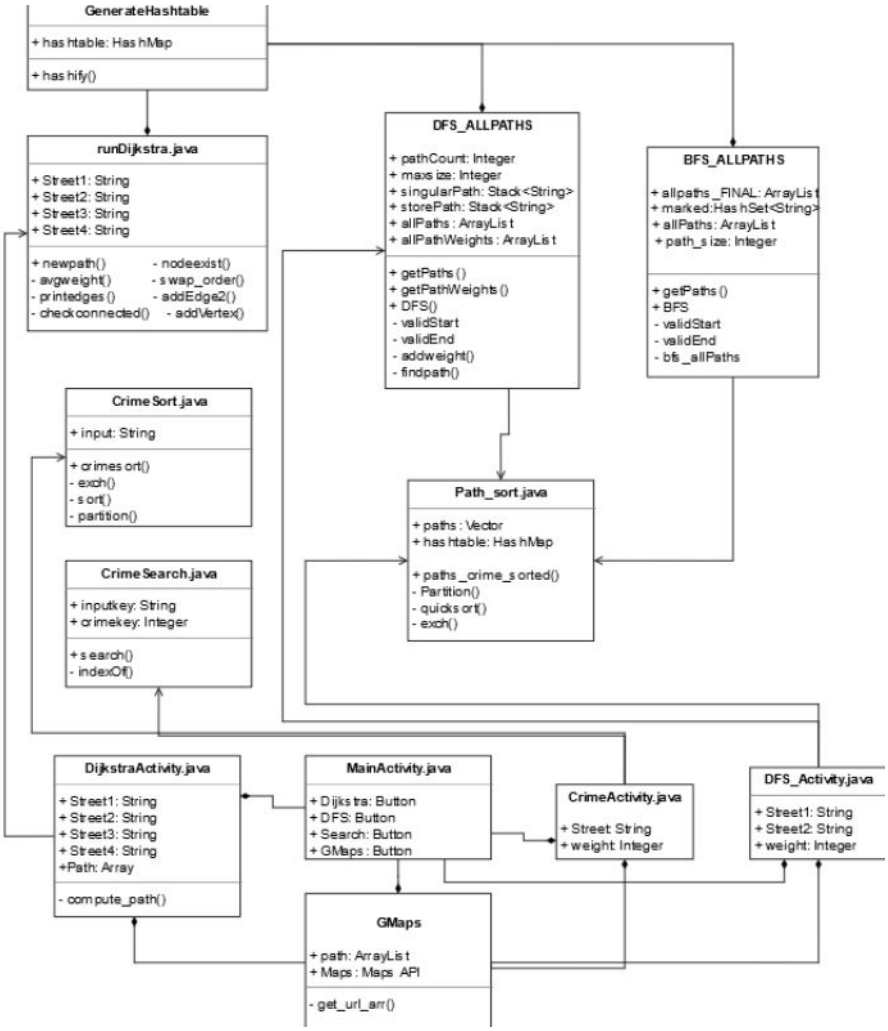
Dijkstra's Algorithm: Efficient solution for finding the shortest route between two points in a graph [2].

Class diagram and description

UML Diagram:

Clearer Image found on
(https://drive.google.com/file/d/107cKQsJVs7YnuSyPDJDqFavTE4OKNt7Z/view?usp=sharing)

Fig 1.1 Below: Application UML Diagram



High level description

SafeWayz consists of many classes that are shown in the UML diagram above. The following is a brief description of the various components. This will include both syntax, semantics, relationships between modules, and design decisions

The modules are separated into the following 5 categories:

- Hashtable
- GraphTraversals
- Dijkstra's Paths
- CrimeSort
- Android Activities

The modules were designed by these five categories to preserve modularity and promote separation of concerns. This way, specific features and characteristics can be traced to the respective module to debug and analyze issues during the unit testing and debugging stage. It also makes development more efficient as developers can work in different folders rather than one large folder.

These five categories are labelled this way since the application consists of 4 main features:

- Generating the optimal least crime path (Dijkstras)
- Generating the least crime path with a specific number of streets(DFS, BFS, Sort, Search)
- Computing the street that makes an intersection with a given street and crime weighting(Crime Sort/Search)
- Plotting and navigating through the Android view

Below is a summary of the modules:

The Hashtable category consists of only the GenerateHashtable module. This module reads the WeightData.json file which represents the undirected graph in JSON format. The end behaviour of this module is to output a hashmap from the JSON data to be used throughout the rest of the application

The GraphTraversals category consists of 2 files which are DFS_ALLPATHS and BFS_ALLPATHS modules. This category is used for traversing through the graph and computing all possible paths from start to end street. Both have similar behaviour as they both compute the possible paths and save them into a 2D vector to be accessed later.

Dijkstra's Paths category consists of the modules needed for computing the least crime path via Dijkstra's Algorithm. It is separate from the GraphTraversal category as it is more complex compared to the simpler DFS and BFS algorithms. It also uses different file dependencies. For these reasons, it is logical to keep the components of this algorithm separate, so that it is easier to trace back problems specifically for this algorithm alone. This module takes the start and end street intersections in which it then returns a hashmap of the computed path.

The CrimeSort category consists of the various Search and Sorting algorithms needed in the CrimeFeature. Specifically, these files are CrimeSorting.java and CrimeSearch.java. This category also contains the path_sort module, which finds the least crime path from a sequence of all possible paths. If the least crime path cannot be computed, it returns a NullPointerException.

The Android Activities category is specific for use with Android studio. The files here consist of mainly the activity files that represent different pages of the application. The behaviour consists of the general UI flow such as interacting with the application. This can include moving between pages by clicking the buttons, or interacting with the Maps API to zoom in on the plotted path.

Module Description/Structure/USES

GenerateHashtable Module

GenerateHashtable()

Description:

Parses the JSON data into a hashtable to be used in Java. This hashtable will represent the undirected graph along with its data

USES

WeightedData.json generated from implement_weights.py

Syntax

Exported Constants:

None

Exported Types:

Hashmap of JSON data

Exported Access Programs

Routine name	In	Out	Exceptions	Description
Hashify	WeightedData.json	HashMap	NullPointer Exception	Outputs hashtable format of JSON data

Semantics

State Variables:

None

Access Routine Semantics

Hashify(JSON data): // This method reads the JSON graph and outputs it in the form of a hashmap

- *Output: HashMap format of the data*
- *Exception: if file is not found or unparsable, NullPointerException is raised*

DFS_ALLPATHS Module

DFS_ALLPATHs()

Description:

A class that computes all possible paths from a given start street and end street via DFS algorithm

USES

GenerateHashtable

Syntax

Exported Constants:

None

Exported Access Programs

Routine name	In	Out	Exceptions	Description
DFS_ALLPATHs	Integer	DFS_ALLPATHs	NullPointer Exception	Creates object of class
getPaths		ArrayList<Vector<String>>		Returns 2D arraylist of all paths
getPathWeights		ArrayList<ArrayList<Integer>>		Returns 2D arraylist of weights of the paths
DFS	String, String	None	NullPointer	Computes the paths and saves it to allpaths 2D ArrayList

Semantics

State Variables:

pathCount = Integer;
maxSize = Integer;
hashtable = generateHashtable.hashifiy("WeightedData.json");
singularPath = Stack <String>;
storePath = Stack <String>();
allPaths = ArrayList<Vector<String>>()
allPathWeights = ArrayList<ArrayList<Integer>>()

Access Routine Semantics

DFS_ALLPAThs(maximum_size): //constructor

- *Output: new DFS_ALLPATH*
- *Exception: NullPointer if cannot create*

getPaths(): //returns all the possible paths computed from the DFS algorithm in the form of a 2D list

- *Output: this.allpaths*

getPathWeights(): //gets an array full of crime weights for allpaths

- *Output: this.allPathWeights*

DFS(): //Executes the DFS algorithm to compute the paths and store them in the allpaths arraylist

- *Transition: allpaths.add(this.storePath)*

Local Methods

validStart: String → Boolean
validStart(startPoint): Checks if the string is a valid starting point

validEnd: String → Boolean
validStart(endPoint): Checks if the string is a valid ending point

inStack: String → Boolean
inStack(street): Checks if the string is in the DFS call stack

addWeight: Integer X Stack<String>
addWeight(number, singularPath): Finds the overall crime weighting of the particular path

findPath: String X String
findPath(startPoint, endPoint): Actual implementation of DFS to find all the paths from start to end point

BFS_ALLPATHS Module
BFS_ALLPATHS()
Description:
A class that computes all possible paths from a given start street and end street via BFS algorithm

USES
GenerateHashtable

Syntax
Exported Constants: None
Exported Access Programs

Routine name	In	Out	Exceptions	Description
BFS_ALLPATHS	Integer	BFS_ALLPATHS	NullPointer Exception	Creates object of class
getPaths		ArrayList<Vector <String>>		Returns 2D arraylist of all paths
BFS	String, String	None	NullPointer	Computes the paths and saves it to allpaths 2D ArrayList

Semantics
State Variables:
allpaths_FINAL: ArrayList<Vector>
marked: HashSet<String> //marks seen streets
allPaths: ArrayList<Stack>()
path_size: Integer

Access Routine Semantics

BFS_ALLPATHS(size): //Constructor
- *Output: new BFS_ALLPATH*
- *Exception: NullPointer if cannot create*

getPaths(): //Getter method to extract arraylist containing all paths
- *Output: this.allPaths*

BFS(): //appends each path to a an arraylist of all possible paths
- *Transition: allpaths_FINAL.add(newpath)*

Local Methods

validStart: String → Boolean
validStart(startPoint): Checks if the string is a valid starting point

validEnd: String → Boolean
validStart(endPoint): Checks if the string is a valid ending point

getPaths_size: String → Boolean
inStack(street): Checks if the string is in the DFS call stack

bfs_allPaths: String x String
bfs_allPaths(startPoint, endPoint): Actual implementation to run BFS to compute all possible paths

PATHS_SORT Module
PATH_SORT()
Description:
A class that sorts all the paths given in 2D arraylist form

USES
BFS_ALLPATHS, DFS_ALLPAThs

Syntax
Exported Constants: *1D ArrayList that represents shortest path*
Exported Access Programs

Routine name	In	Out	Exceptions	Description
paths_crime_sorted	ArrayList<Vector<String>>, HashMap<String,Object>	Vector<String>	NullPointer Exception, IndexOutOfBoundsException	Sorts the 2D arraylist containing all possible paths by crime weights. Returns the least crime path.

Semantics
State Variables: None

Access Routine Semantics

paths_crime_sorted(paths, hashtable):

- *Output: 1-Dimensional Vector containing least crime path*
- *Exception: NullPointer if cannot create. And IndexOutOfBounds*

Local Methods
Quicksort: Vector<Vector<Integer>> x Integer x start x end
Quicksort(arr, start, end): sorts the array recursively

Partition: Vector<Vector<Integer>> x Integer x start x end
Partition(arr, start, end): partitions the array into two halves(smaller on left, greater on right)

exch: Vector<Vector<Integer>> x Integer x Integer
exch(a, i, j): Exchanges the value of index i and index j for the vector

CrimeSort Module

CrimeSort()

Description:

A class that sorts a hashmap based on a given street in ascending order.

USES

GenerateHashtable

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
crimeSort	String, Map<String, Object>	Map<String, Integer>	None	Uses local methods to sort a hashtable, and return a sorted Map of street names and crime keys.

Semantics

State Variables:

None

Access Routine Semantics

crimeSort(input, hashtable):

- *Output: Sorted Map of Street names mapped to crime keys*
- *Exception: None*

Local Methods

validKey: String x Map<String, Object> → Boolean
validKey(key, hashtable): Checks if the key is contained in the hashtable

exch: String[][] x int x int → void
exch(a, i, j): Exchanges the value of index i and index j

sort: String[][] → void
sort(a): Randomly shuffles the array and calls sort(a, 0, a.length - 1)

sort: String[][] x int x int → void
sort(a, lo, hi): Sorts the array in ascending order

partition: String[][] x int x int → int
partition(a, lo, hi): Partition the array and return the position of the last element of the partition

CrimeSearch Module

CrimeSearch()

Description:

Searches for the closest intersection based on given street and crime index

USES

None

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
search	String, int, Map<String, Object>	String	None	Sorts the Map given and creates a 2D string array to be searched from. Returns the street based on the crime index
indexOf	String[][], int	int	None	Finds the closest index based on the key.

Semantics

State Variables:

None

Access Routine Semantics

search(intputKey, crimeKey, hashtable):

- *Output: A String of the street that has a crimeKey closest to the crimeKey given*
- *Exception: None*

indexOf(a, key):

- *Output: An integer representing the closest index to the crimeKey*
- *Exception: None*

Vertex Module

Vertex()

Description:

Data type representing a vertex in the Dijkstra graph

USES

None

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
New Vertex	String, String, int	Vertex		Create new vertex object
getId		String	None	Returns the ID of the vertex
getName		String	None	Return the name of the vertex
getWeight		int	None	Return the weight of the vertex
hashCode		int	None	Calculates the hashcode of the vertex
equals	Object	boolean	None	Check if this object equals another object
toString		String	None	String representation of the vertex

Semantics

State Variables:

id: String
street: String
weight: int

Access Routine Semantics

Methods, followed by their output[or transition] AND exceptions
Vertex(id, street, weight): // Constructor

- *Output: new Vertex*
- *Exception: None*

getId():

- *Output: id*
- *Exception: None*

getName():

- *Output: street*
- *Exception: None*

getWeight():

- *Output: weight*
- *Exception: None*

hashCode():

- *Output: calculate hashcode of the vertex's id*
- *Exception: None*

equals():

- *Output: Override built-in equality to check if this object is equal to another object*
- *Exception: None*

toString():

- *Output: street*
- *Exception: None*

Edge Module

Edge()

Description:

Data type representing an edges inthe Dijkstra graph

USES

Vertex

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
New Edge	String, Vertex, Vertex, int	Edge		Create new Edge object between two vertices
getId		String	None	Returns the ID of the edge
getDestination		Vertex	None	Return the vertex at the destination of the edge
getSource		Vertex	None	Return the vertex at the source of the edge
getWeight		int	None	Return the weight of the edge
toString		String	None	String representation of the edge

Semantics

State Variables:

id: String
source: Vertex
destination: Vertex
weight: int

Access Routine Semantics

Methods, followed by their output[or transition] AND exceptions
Edge(id, source, destination, weight): // Constructor

- *Output: new Edge*
- *Exception: None*

getId():

- *Output: id*
- *Exception: None*

getDestination():

- *Output: destination*
- *Exception: None*

getSource():

- *Output: source*
- *Exception: None*

getWeight():

- *Output: weight*
- *Exception: None*

toString():

- *Output: source + "---" + destination*
- *Exception: None*

Graph Module

Graph()

Description:

Data type representing a graph consisting of edges and vertices

USES

Vertex, Edge

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
New Graph	List<Vertex>, List<Edge>	Graph		Create new graph object
getEdges		List<Edge>	None	Returns the list of edges in the graph
getVertexs		List<Vertex>	None	Return the list of vertices in the graph

Semantics

State Variables:

edges: List<Edge>

vertexs: List<Vertex>

Access Routine Semantics

Methods, followed by their output[or transition] AND exceptions

Graph(vertexs, edges): // Constructor

- *Output: new Graph*
- *Exception: None*

getEdges():

- *Output: edges*
- *Exception: None*

getVertexs():

- *Output: vertexs*
- *Exception: None*

RunDijkstra Module

RunDijkstra()

Description:

Finds a path between two intersections utilizing Dijkstra's Algorithm with crime weighted paths.

USES

Vertex, Edge, GenerateHashtable, Dijkstra

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
New runDijkstra	String, String, String, String	runDijkstra		Creates runDijkstra object between two intersections.
newPath		LinkedList<Vertex>	FileNotFound, IOException, ParseException	Generates a new path between the intersections

Semantics

State Variables:

start1: String
start2: String
end1: String
end2: String
nodes: List<Vertex>
Nodes: HashMap<String, Vertex>
Edges: List<Vertex>
edges: List<Vertex>
fileName: String
Hashtable: Map<String, Object>
Intersection: Hashmap<String, ArrayList<String>>

Access Routine Semantics

Methods, followed by their output[or transition] AND exceptions
runDijkstra(s1, s2, e1, e2): // Constructor

- *Output: new runDijkstra*
- *Exception: None*

newPath()
- *Output: LinkedList<Vertex>*
- *Exception: FileNotFoundException, IOException, ParseException*

Local Functions

checkConnected: Boolean
checkConnected(): Determines if a graph is connected to itself

printIntersections: Changes environment variable of the screen
printIntersections(): Prints all the intersections in the graph

addVertex: Null
addVertex(): Parses all intersections and adds them to a graph.

addEdge2: Null
addEdge2: Creates edges between all intersections within a graph.

checkPermutation: Boolean
checkPermutation(S): Determines if some permutation of the intersection is within the graph already.

swapOrder: Null
swapOrder(S): Switches the order of streets composing an intersection.

nodeExists: Boolean
nodeExists(S1, S2): Checks if an intersection exists between two streets.

getNode: String
getNode(S1, S2): Returns the string name of the node containing the two inputs streets.

printEdges: Null
printEdges(): Outputs all edges containing a set street. Used for testing.

avgWeights: Integer
avgWeights(A, B): averages the value of two weights.

validKey: String x Map<String, Object> → Boolean
validKey(key, hashtable): Checks if the key is contained in the hashtable

Dijkstra Module

Dijkstra()

Description:

Description Here

USES

GenerateHashtable, Edge, Vertex, Graph

Syntax

Exported Constants: *None*

Exported Access Programs

Routine name	In	Out	Exceptions	Description
New Dijkstra		Dijkstra		Initializes the algorithm with a set dataset.
execute	Vertex			*Relaxes the graph.*
getPath	Vertex	LinkedList<Vertex>		Returns a calculated path to the input vertex from the start.

Semantics

State Variables:

fileName: String
Hashtable: Map<String, Object>
Weights: Map<String, Object>
Edges: List<Edge>
Nodes: List<Vertex>
SettledNodes: HashSet<Vertex>
UnsettledNodes: HashSet<Vertex>
Distance: HashMap<Vertex, Integer>
Predecessors: HashMap<Vertex, Vertex>

Access Routine Semantics

Methods, followed by their output[or transition] AND exceptions
Dijkstra

- *Output: new Dijkstra()*
- *Exceptions: None*

getPath(vertex target)

- *Output: LinkedList<Vertex>*
- *Exceptions: None*
- *Transitions: None*

Local Functions

weights: Null

weights(): Reads crime data from the dataset and assigns weights to each edge in the graph.

findMinimalDistances: Vertex

findMinimalDistances(V): Finds the shortest distance between the start and the input vertex.

getDistance: Vertex, Vertex

getDistance(V1, V2): returns the weight of the path between two nodes.

getNeighbors: Vertex

getNeighbors(V): Returns a list of neighbor vertices to the input vertex.

getMinimum: Set<Vertex>

getMinimum(vS): returns the vertex with the shortest distance.

isSettled: Vertex → Boolean

isSettled(V): returns true if the vertex is settled.

getShortestDistance: Vertex

getShortestDistance(V): returns the distance from the start to the input vertex.

MainActivity Module

MainActivity()

Description:

Homepage module of the Android application. Essentially acts like a View and Controller to navigate between the various activity pages(DFS_Activity,Dijkstra_Activity,CrimeFeature, GMaps).

USES

None

Syntax

Exported Constants: *None*

Exported Access Programs

None

Semantics

State Variables:

Dijkstra Button = new Button
DFS Button = new Button
CrimeSearch Button = new Button
GMaps Button = new Button

Access Routine Semantics

No method behaviour as there are no unique methods for this class. However, the general behaviour of the class is to intent/transition to a new page/activity from specific button clicks. This class just acts as an intermediary menu to let the user decide which activity they want to try.

Dijkstra_Activity Module

Dijkstra_Activity()

Description:

Dijkstra least crime path page of the Android application. Consists of 4 searchable spinners(dropdown selection menu) along with "compute", "route info", and "Map it" buttons

USES

RunDijkstra, MainActivity

Syntax

Exported Constants: *Array containing shortest path, sent to GMaps*

Exported Access Programs

None

Semantics

State Variables:

Street1 = String from Spinner1
Street2 = String from Spinner2
Street3 = String from Spinner3
Street4 = String from Spinner4
Compute = new Button
Route_info = new Button
Map_it = new Button

Access Routine Semantics

No public methods available. This class consists of 4 spinner dropdown items that help the user to specify the start and end intersections. The end behaviour of this class is to execute RunDijkstra with the specified parameters of start and end given by the user. After computation, it saves the route information for easy access via the Route_info button. This same route can also be plotted via GMaps API by clicking the "Map It" button to transition to the Map View.

DFS_Activity Module

DFS_Activity()

Description:

Executes DFS_ALLPATHS to retrieve an arraylist with least crime path alongside a given path size between 3 and 7.

USES

DFS_ALLPATHS, Paths_sort, MainActivity

Syntax

Exported Constants: *Array containing shortest path, sent to GMaps*

Exported Access Programs

None

Semantics

State Variables:

Street1 = Street from spinner1
Street2 = Street from spinner2
Size = size from spinner3
FloatingActionButton GO = new Button
Output = new TextView

Access Routine Semantics

No public methods available. This class consists of 2 spinners that are used to specify the start and end streets. Once the GO button is pressed, DFS_ALLPATHS gets executed with the given size provided by spinner3. If computation is successful, then the end behaviour results in a transition from DFS_Activity page to GMaps Page. If computation fails, an "Android Toast" message is displayed to the user that prompts them to 'provide different streets or try a different size"

Design Choice

The application was designed with a max path-size of 3-7, so that computation time can be limited. Yes, larger path sizes do work. However, the time to do so is lengthy which can frustrate the user as it is O(N + E), which is linear complexity. A path size range of 3-7 greatly speeds up performance.

CrimeFeature Module

CrimeFeature()

Description:

Executes CrimeSort.java and CrimeSearch.java to find the closest street that makes an intersection with the street provided by the user along with the crime weighting.

USES

MainActivity, CrimeSort, CrimeSearch

Syntax

Exported Constants: None

Exported Access Programs

None

Semantics

State Variables:

FloatingActionButton start = new Button
TextView output = new TextView
EditText crimeRating = new EditText
int crime_rating = Integer
String inputString = String from spinner
String out = String

Access Routine Semantics

No public methods available. Takes in a user input from the spinner dropdown and saves to inputString. The user enters a crime weight in the EditText box. This crime weight will be used to find the street that makes an intersection with the given street inputted previously from the spinner. The end behaviour results in CrimeSort and CrimeSearch to be executed which then output the street with the given crime.

GMaps Module

GMaps()

Description:

Plots the given sequence of intersections(intersections array) via Google Maps Waypoints API.

USES

Dijkstra_Activity or MainActivity or DFS_ALLPATHS

Syntax

Exported Constants: *None*

Exported Access Programs

None

Semantics

State Variables:

intersections : Array of Strings representing intersections of the path to be plotted
mMap : represents GMaps Map Object

Access Routine Semantics

This class doesn't have any public methods that are designed by the developer. It does include the standard "public void onMapReady" method as this is needed to display the map on screen via the API. This class uses the exported path arrays from Dijkstra_Activity or DFS_Activity if available. If a path is available, then it plots it via GMaps Waypoints API URL Request. The end behaviour results in the API to receive the mapping information in the form of a JSON that gets parsed and then plotted to the map for the user to see.

Module USES Relations:

This summarizes the Uses relations shown above for each module in one table.

Table 2.1: Uses relationship between the various modules and their dependencies

Module	USES
GenerateHashtable	WeightedData.json generated from implement_weights.py
DFS_ALLPATHS	GenerateHashtable
BFS_ALLPATHS	GenerateHashtable
paths_sort	DFS_ALLPATHS, BFS_ALLPATHS
CrimeSearch	None
CrimeSort	None
RunDijkstra	GenerateHashtable
MainActivity	None
DFS_Activity	MainActivity, DFS_ALLPATHS, paths_sort
Dijkstra_Activity	MainActivity, RunDijkstra
GMaps	MainActivity, DFS_Activity, Dijkstra_Activity

State Diagrams And Implementation

As explained earlier in the Module Description section, there are 5 categories in which each module was grouped into prior to the implementation stage.

These 5 categories are:

- HashTable
- GraphTraversals
- Dijkstra's Paths
- CrimeSort
- Android Activities

HashTable
This consists of the GenerateHashtable module. This module only has one method which is both public and static since the graph is only created once, and read throughout the application. This class has no private methods.

GraphTraversal
GraphTraversal consists of DFS_ALLPATHS and BFS_ALLPATHS modules. Both of these modules contain some local functions to assist the public functions explained in the Module Description section.

BFS_ALLPATHS contains validStart,validEnd, and bfs_allPaths methods. The validStart and validEnd are used to check if the streets provided by the user are valid. If not, it returns false, and does not continue proceeding with the computation. bfs_allPaths is the method that runs the BFS algorithm using a Queue in the backend. All three of these methods get called when the public BFS() gets called by external classes to compute a path for valid start and end points.

DFS_ALLPATHS works in a similar manner but has its local methods using Stacks instead of a Queue. It too also has a validStart and validEnd method to check if the start and end streets are valid prior to finding a path. It has addweight() and findpath() local methods. findpath() works similar to bfs_allPaths() of the BFS module. It takes two string parameters that represent start and end streets. It then computes all the possible paths of a specified size. In addition to this, the addweight() local method is used to update the paths with the running total of weight.

Dijkstra's Paths:
Dijkstra's path consists of the following classes: Edge, Vertex, Graph, Dijkstra, and runDijkstra. All classes contain local methods utilized during execution.

Edge, Vertex, and Graph are type modules used to create a weighted graph framework that can be accessed and modified by the other classes.

Dijkstra contains its constructor, which creates an instance of the class. The object is populated with data from the dataset which is stored in hash tables. The execute method is used to start the algorithm and begin relaxing edges. This utilizes the findMinimalDistances to relax an edge if a shorter path is available to the destination. Private methods are used to assign weights to created edges and provide access to graph information; this includes weights, path lengths, and whether the vertex has been visited. Lastly, getPath will traverse the created graph and return the lowest weighted path from the assigned start position to an input end position.

CrimeSort:
CrimeSort category consists of the CrimeSorting, CrimeSearch, and paths_sort classes. These classes all have local methods that were used throughout the application.

The paths_sort class consists of private sorting methods and their counterparts. This includes quicksort(), partition(), and exch methods. These local methods are crucial in having a correctly functioning public *paths_crime_sorted()* method. The methods help to find the least crime path by first sorting the paths via quicksort. Quicksort consists of the quicksort(), partition(), and exch() methods as it is a recursive sorting algorithm. It partitions the array and does the swaps appropriately until the 2D vector of paths are sorted. Once sorted, the public *paths_crime_sorted()* method can easily find the path with the least crime since that occurs at index 0 of a sorted vector in ascending order.

In addition, the CrimeSorting and CrimeSearch programs are also of great importance and contain various local methods. CrimeSorting class consists of mostly private methods that are extremely crucial for the public method to function. These include, *validKey(), exch(), sort(), sort(), partition()*. The only public method in this module is the *crimeSort()* method. The crimeSort() is an implementation of the quicksort algorithm which randomizes the 2D array and recursively partitions and swaps appropriately until the array is sorted. The CrimeSorting module sorts a 2D array of Map objects provided by the hashtable.

CrimeSearching, the only module that uses a searching algorithm consists of two public methods. *search()* uses *indexOf()* to provide a string representation of the street found based on a crime index. The search() method is relient on the indexOf() method due to the fact that the linear search algorithm uses it to find the corresponding street.

Android Activities:
The android activities section also consists of a few modules, however, only the *GMaps* and *Dijkstra_Activity* classes contain local methods.

The local method for GMaps is *get_url_arr()*. This method is important as it is responsible for sending an HTTP Request to the Google Waypoints API to parse the intersections_array so that it can be plotted on GMaps. Aside from local functions, it only consists of the private *intersections* array variable which contains the path information passed from either Dijkstra or DFS activity classes. This array stays constant as this is the path that will be plotted via GMaps.

In addition, the local method for *Dijkstra_Activity* is the *compute_path()* method. This method is responsible for executing Dijkstra's path algorithm only when the *"compute"* button is pressed. This method was made to keep the code organized and easy to read since large blocks of code inside the button-click call proved to be difficult to read. This method uses the concept of threading to compute the path while showing a "loading" animation to the user. Threads were implemented to keep the system running concurrently while also providing the user with useful information so that they are aware that the computation is "loading". This prevents the user from thinking the app is frozen or that the app has crashed.

State Diagrams

Fig 2.2: UML State Diagram for DFS_Activity

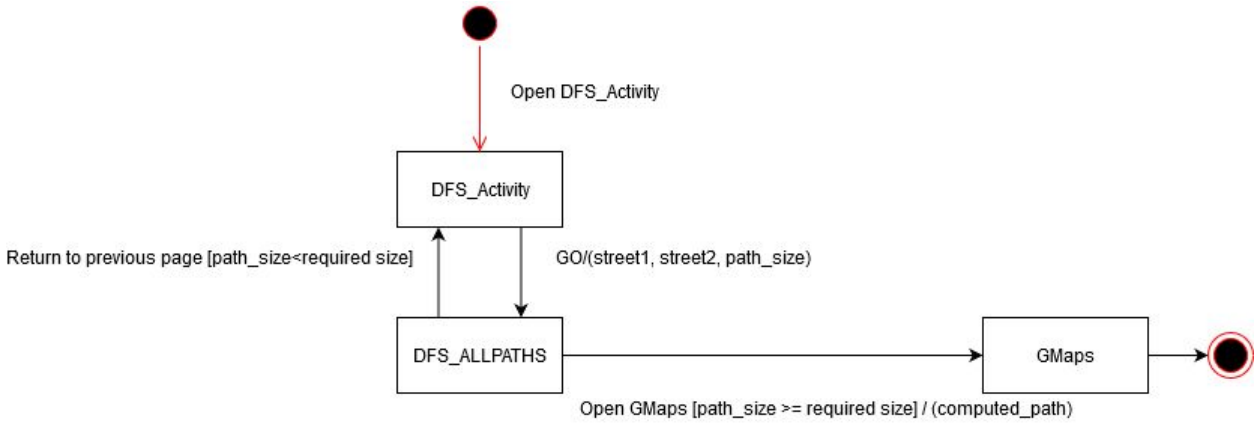
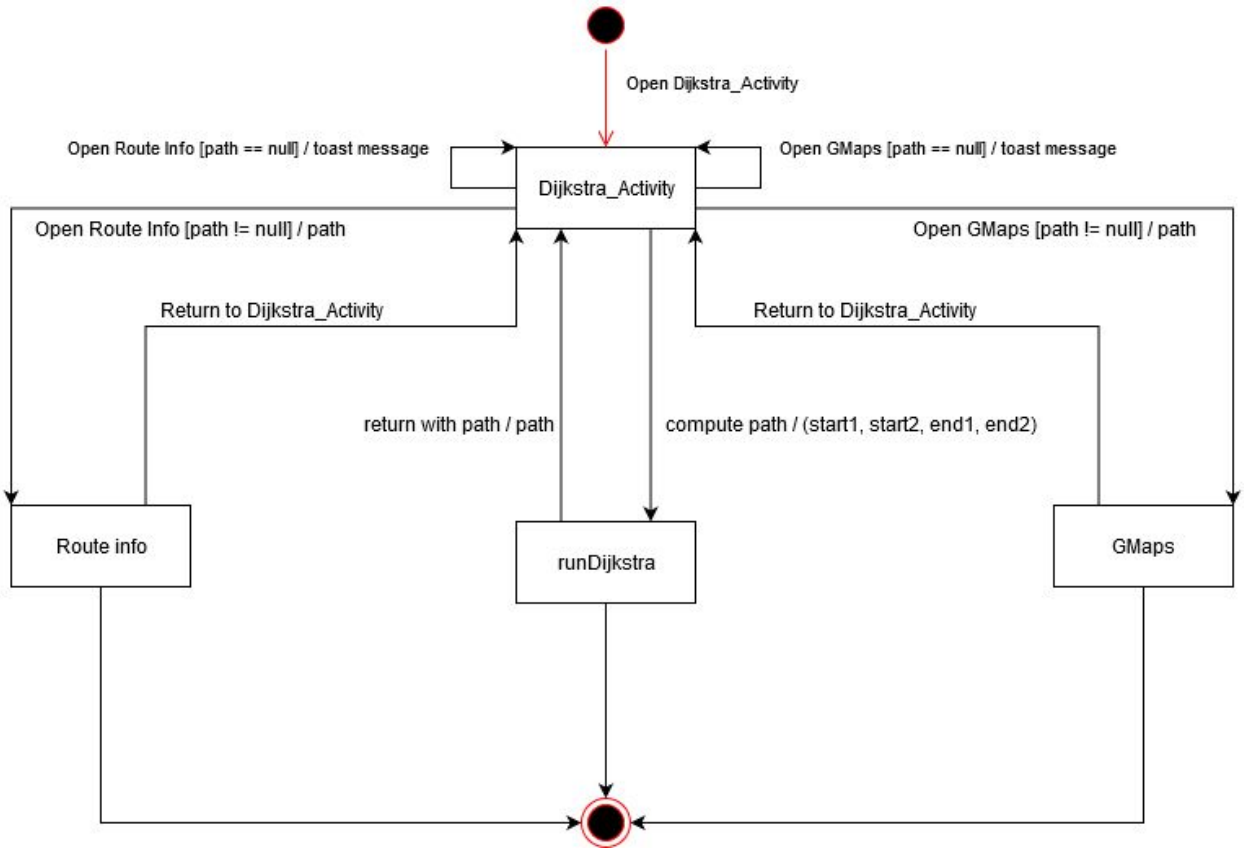


Fig 2.3: UML State Diagram for Dijkstra_Activity



Review/Evaluation of Design

Overall, the application was designed well using the concepts of software qualities and modular design taught in the *SE 2AA4 introductory software developmen*t. In addition, concepts of algorithmic practice/analysis taught in *SE 2XB3* and *2C03* courses were effectively used for optimizing performance.

The development process was done in an agile manner rather than waterfall as it allows for rapid prototyping which contributes to the incremental approach in development. This allowed for changes to be implemented throughout the entire development stage. Throughout the development process, we leveraged Git via GitLab. This proved to be extremely effective in dividing work, ensuring everybody was able to contribute concurrently, and to ensure that everybody was on the same page at all times. This resulted in an overall smooth development process.

An issue that the team faced was the lack of research on the tools that would be used. During the project, the team established the fact that the Google Maps API would be used to visualize the paths. However, not enough research on the API was conducted. As a result, towards the end of the project, the team learned of a major API limitation - that the API could only plot a route that had a maximum of 25 intersections. This was a major roadblock because the Dijkstra algorithm would sometimes generate paths that have more than 25 intersections.

In order to resolve this issue, the team chose to plot a path using every 5 intersections from the path generated by Dijkstra. As a result, the accuracy of the paths being visualized decreased. However, the team also chose to visualize the paths in text format so the client can have access to the path. In the future, this could be changed so that different waypoint HTTP requests are called consecutively so that they can all be combined into one path at the end, without losing intermediate data.

Another issue was the quality of the data and the plotting nature of Google Maps. The team chose two datasets from the San Francisco Government that had crimes on each intersection. However, during the plotting process, the team realized that not all intersections had geographical coordinates. As a result, they could not be plotted onto the Google Maps API. A solution that was developed was a change in the nature of plotting. Rather than providing geographical coordinates to plot, the team chose to provide the intersections in a *street1@street2* format.

In addition, another issue was the interpretation of the data by Google Maps. As mentioned above, the graphing algorithms generated a path via intersections and the path would be visualized by providing the data in a *street1@street2* format to Google Maps. However, it turned out that some of the provided intersections would exist in different areas of the United States rather than just in San Francisco. For instance, the street *Alabama St@Ripley St* would be part of a path in San Francisco, but this intersection would also exist in the state of Florida. This would cause the Google Maps API to plot a path to Florida from San Francisco.

In order to resolve this issue, during the plotting process, the team chose to concatenate the keyword "San Francisco" to every intersection. So *Alabama St and Ripley St* became *Alabama St,San Francisco and Ripley St,San Francisco.* This resolved the problem with the Google Maps API because the API would now be notified that the intersection exists in San Francisco rather than in Florida.

Another issue that was encountered was the discrepancy between the dataset and the current Google Maps API data. As mentioned above, a path would be generated using the dataset and the intersections would be plotted on the API. However, the data set was generated in 2013 and was not updated since. So, when the paths would be plotted, Google Maps would oftentimes fail to recognize a particular intersection. This was due to several reasons. One of the reasons was because of a construction or demolition project that took place between 2013 - Today. Since this issue was out of the team's control, there was no solution being implemented. For future revisions, it may be of benefit to gather location data in the form of longitude and latitude as it is a more accurate means of plotting and gathering more recent data.

Another issue was that of plotting Dijkstra's algorithm. It was found at the end of the development process that the algorithm did not recognize some streets. This resulted in the program to completely ignore them when trying to compute the paths. This could have been due to inadequate testing on this module for the various streets as only a handful were tested. In addition, since this was also a *top-down* approach to development, it is another reason why lower level modules such as Dijkstra's were not tested in-depth. In the future, it may be beneficial to spend more time on the testing stage before moving onto the actual integration and final development stages.

Aside from these few issues, the development process went relatively well. Although the testing phase could have been more rigorous for some modules, it still went relatively well for most. The use of *Top-Down* approach in incremental testing proved to be useful as it eliminated the idle time waiting for all modules to be complete. The time saved was used for other aspects of development such as module porting/integration for android studio.

In addition to this, the use of modular concepts such as high cohesion and low coupling were beneficial in modular design. It allowed each of the modules to be closely related without being completely dependent on each other. Having a good level of modularity proved to be useful as it isolated the concerns to separate modules. This way, bugs related to certain features can be traced back and debugged to their respective modules.

In the end, the mobile application was a success as it met the objectives and requirements from the SRS document while also making use of all three graph algorithms (BFS, DFS, Dijkstras) along with Search (BinarySearch) and sorting algorithms (QuickSort). It also included excellent software development principles(software qualities and testing) that proved to be useful. Time and planning was well managed with multiple sprints to get the tasks done with minimal delay. Even though there were some problems along the way, the overall application works quite well and meets the design requirements.

References

[1] “App,” *Merriam-Webster*. [Online]. Available: https://www.merriam-webster.com/dictionary/app. [Accessed: 15-Mar-2020].

[2] P. E. Black, “Dijkstra's algorithm,” *NIST*, Sep-2006. [Online]. Available: https://xlinux.nist.gov/dads/HTML/dijkstraalgo.html. [Accessed: 15-Mar-2020].

[3] “Chapter 4 Graphical User Interfaces,” *Chapter 4. Graphical User Interfaces*. [Online]. Available: https://cs.stanford.edu/people/eroberts/jtf/tutorial/GraphicalUserInterfaces.html. [Accessed: 15-Mar-2020].

[4] “Introducing JSON,” *JSON*. [Online]. Available: https://www.json.org/json-en.html. [Accessed: 15-Mar-2020].

[5] OECD Statistics Directorate, “DATA SET,” *OECD Glossary of Statistical Terms - Data set Definition*, 25-Sep-2001. [Online]. Available: https://stats.oecd.org/glossary/detail.asp?ID=542. [Accessed: 15-Mar-2020].

[6] R. Sedgewick and K. Wayne, “Shortest paths.,” *Princeton University*, 16-Nov-2018. [Online]. Available: https://algs4.cs.princeton.edu/44sp/. [Accessed: 15-Mar-2020].

[7] R. Sedgewick and K. Wayne, “Undirected Graphs,” *Princeton University*, 16-Apr-2019. [Online]. Available: https://algs4.cs.princeton.edu/41graph/. [Accessed: 15-Mar-2020].

[8] “What is an API?,” *Red Hat*, 2020. [Online]. Available: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces. [Accessed: 15-Mar-2020].

[9] D. Duran, “Algorithm Efficiency,” *Algorithm Efficiency*. [Online]. Available: http://www.cs.kent.edu/~durand/CS2/Notes/03_Algs/ds_alg_efficiency.html. [Accessed: 16-Mar-2020].

[10] “DFS algorithm,” *Programiz*. [Online]. Available: https://www.programiz.com/dsa/graph-dfs. [Accessed: 16-Mar-2020].

[11] “What is User Experience (UX) Design?,” *The Interaction Design Foundation*. [Online]. Available: https://www.interaction-design.org/literature/topics/ux-design. [Accessed: 16-Mar-2020].