Monitoring Bushfires with UAVs

COMP1002 Data Structures and Algorithms Final Assignment

# A Brief Introduction to the Task:

This report outlines the implementation and testing of a program that aims simulate a situation in which a scaling number of Unmanned Aerial Vehicles (UAVs) are used to monitor an area for bushfires. The area consists of any number of key locations with any number of connections between these locations. When a UAV is deployed to a location, it collects data regarding the temperature (between 25-48oC), the humidity (between 15-60%) and the wind speed (between 30-100km/h) and determines a bushfire risk factor accordingly.

# Outline of program design

## Checking command line arguments:

The program is designed to take 3 additional command line arguments, the name of the file containing the locations, the name of the file containing the UAV data for each location and the number of UAVs deployed for the area. To ensure the correct number of additional arguments are entered, the program checks the length of the argv array containing the command line inputs. If this length is not equal to 4 (includes the program name) the program prints a block of text outlining the usage of the program before exiting.

The usage() function in the utils file is used to print the usage of the program to the screen

## Loading the location data:

To load in the locations of the area the program assumes the file is in the correct format with the first line indicating the number of locations and connections, and subsequent lines indicating the distance and the locations joined by a connection. If an incorrect file name is passed to the location loading function an error will be raised, then caught by the main program before exiting.

The program passes the location file name to the loadLoactions() function (in utils.py) which reads in the number of connections (corresponding to the number of additional lines in the file). The function then reads in the connections and their distances one at a time appending each location and edge to an undirected graph. The graph is then returned to the main function.

NOTE: if a location has no connections it is added by the UAV data loader

## Loading the UAV data:

To load in the data collected by the UAVs upon searching a location, the program assumes the file is in the correct format with each line indicating the location as well as the temperature, humidity and wind speed at that location. It is assumed the data for every location appears within the file. If an incorrect file name is passed to the data loading function an error will be raised, then caught by the main program before exiting.

The program passes the data filename to the loadData() function (in utils.py) which reads the file line by line until the end of the file. Each line is then split into its components (location and data), with the data combined into a CSV string. The function then appends this data to a hash table using the combined ASCII values of the location as a key. The function also checks if the locationGraph does not contain the listed vertex and appends it as required. Finally, the hash table and the locationGraph are returned.

## Creating the UAVs:

Before creating any UAVs the program first checks if the command line input for the number of UAVs is an integer. If the input is not an integer an error will be raised, then caught by the main program before exiting.

Once verified the main program calls the createUAVs() function (in utils.py) which creates the corresponding number of UAVs requested by the user. For each UAV the function continuously asks the user to input a starting location that appears within the area. After a valid location is entered the UAV object is added to an array which is returned to the main program.

## Creating a risk heap:

The final initialisation step in the main program is to create an empty heap that will contain only the locations with the highest risk factor. To do this the main program creates a DSAHeap object defined in the heap.py file.

## Menu:

The key functionality of the main program is captured by an interactive menu which continually prompts the user to input a valid command. If an invalid command is entered the menu calls the printCommands() function (in utils.py) to print the valid commands to the terminal before prompting the user to re-enter a command. The possible methods accessible through the menu are as follows:

### Displaying locations as an adjacency list:

When the user enters the “ds” command into the interactive menu, the displayAsList() function within the locationGraph object is called. For each location this prints the label of the location followed by a “|” and then every location it is directly connected to.

### Inserting a new location:

When the user enters the “il” command into the interactive menu, the insertLocation() function (in menu.py) is called. The first step in this function continually prompts the user to enter a label for the location until a label that does not exist is entered. Once the location is verified it is added to the locationGraph. The next step of the code continually prompts the user to enter a temperature, then a humidity and then a wind speed within the given bounds for the new location. This time the verified data is added to the uavData hash table. The final step in this process is asks the user to add any connections to the new location. If the user decides to add a connection, they are continuously prompted to enter a valid existing location and a distance between these locations. This data is then added as an edge to the locationGraph.

The updated hash table and graph are then returned to the main program.

### Inserting a new connection:

When the user enters the “ic” command into the interactive menu, the insertConnection() function (in menu.py) is called. Similar to the last stage in the “inserting a new location” method, the user is continuously prompted to enter two existing locations for the new connection and the distance between them. Once, verified the data is added as an edge to the locationGraph.

The updated graph is then returned to the main program.

### Deleting a current location:

When the user enters the “dl” command into the interactive menu, the deleteLocation() function (in menu.py) is called. The user is then continuously prompted to enter the label of an existing location for which they wish to delete. After an existing location is entered both the location and its data are removed from the locationGraph and the uavData hash table. The method then checks all the current UAV locations and continually prompts the user to enter a new existing location for any UAVs that resided over the recently deleted location.

The updated graph, hash table and UAVs are returned to the main program.

### Deleting a current connection:

When the user enters the “dc” command into the interactive menu, the deleteConnection() function (in menu.py) is called. The user is then continuously prompted to enter the label of two existing locations joined by the connection they wish to delete. The edge is then deleted from the locationGraph, which is updated and returned to the main program. If the connection does not exist, the deleting process is still carried out, however, no changes are made, and the graph remains the same.

### Completing a UAV search:

When the user enters the “sl” command into the interactive menu, the searchLocation() function (in menu.py) is called. This function asks the user to enter the type of search they wish to complete, a search of all locations (“entire”) or a search between locations (“between”). If an incorrect input is entered, the search is cancelled, and the function returns to the interactive menu. The process for each search case is outlined below:

#### Entire:

When “entire” is entered into the search prompt, the method continually prompts the user to select which UAV is completing the search. Once a valid UAV number is entered, the corresponding UAV location is passed through the depth-first search algorithm (within the graph class). This algorithm returns the path (which is printed to the terminal) of the UAV as it searches all locations in the area. After the path is obtained the getData() function (in utils.py) uses it to calculate the risk factor for each location visited and updates the risk heap accordingly. Finally, the UAV location is updated to the end location of the search.

The updated graph, UAV data hash table, UAVs and risk heap is returned to the main program.

NOTE: If a location exists with no connections the program will indicate that it has not been searched.

#### Between:

When “between” is entered into the search prompt, the method continually prompts the user to select the UAV completing the search and the end location of the search. Once a valid UAV Number and an existing location is entered, the function passes the corresponding UAV location and the end location through the breadth-first search algorithm (within the graph class). This algorithm returns the path that the UAV takes from its current location to the specified end location (which is then printed to the terminal). After the path is obtained the getData() function (in utils.py) uses it to calculate the risk factor for each location visited and updates the risk heap accordingly. Finally, the UAV location is updated to the end location of the search. Just like the “entire” search, the updated graph, UAV data hash table, UAVs and risk heap is returned to the main program.

NOTE: If a location exists with no connections, the program will indicate that no path has been found and the UAV will not move.

### Create an itinerary

When the user enters the “it” command into the interactive menu, the getItinerary() function (in menu.py) is called. This method provides an itinerary or ‘path’ for each UAV to visit each high-risk location in the shortest distance. The itinerary or ‘path’ provided only indicates the high-risk locations and not the intermediate steps. This is done to mimic the search “between” function where each location listed on the itinerary is an end location input. For example, if location A, C and E are at high risk and a single UAV is located at F then a possible itinerary could be F->A->E->C. The user can than use the “between” search function 3 times first between F and A, second between A and E and finally between E and C. In this instance the itinerary only shows locations A, C, E and F, mimicking the starting and final locations of the breadth-first search and thus omitting intermediate locations of low risk (such as B and D) that the UAV might pass through while searching. To put it simply the itinerary provides a set of instructions for the user to complete a search to all high-risk areas following the shortest path.

Within this method there is 3 different categories which are defined below:

* One where the risk heap is empty (no locations have been explored);
* Another where the risk heap contains only one location and;
* The final case where the risk heap contains more than one location.

For the first case, the method does not return an itinerary (since no locations are at risk) and instead prints the following string "No locations explored! Please use the search function to add location data".

For the second case, the function iterates over each UAV, each time printing the UAV’s number, its starting location and the at-risk location. Since there is only a single location at a high-risk the itinerary provided for each location only contains the starting location and the end location (user should complete search straight to that area. See explanation above).

For the final case the function creates a separate ‘risk’ graph which fully connects each high-risk location with each other. To do so, the method loops through all locations in the risk heap. For each location the function loops through all the other locations again first adding each location to the risk graph. Next, the method uses Dijkstra’s algorithm to find the distance between the two high risk areas (in the original graph) and then creates a connection within the risk graph with a value of this distance. Once all the locations are fully connected with their respective distances the function then iterates over each UAV fully connecting its location to the risk graph. Finally, the method calls the nearest neighbour algorithm to find the shortest path through the fully connected graph starting at the UAV location and prints this itinerary to the terminal. The function also removes each UAV location from the risk graph after an itinerary has been created.

For each case the method returns the unmodified location graph, UAVs, and risk heap to the main program.

### Close the menu

When the user enters the “cm” command into the interactive menu, the main program prints “closing menu” and exits the menu loop, causing the program to end.

# UML diagram

A picture containing text, diagram, plan, technical drawing

Description automatically generated

# List of Data Structures and Algorithms Used:

The following list of data structures were implemented within this program. These data structures will be described in the “Description of classes”:

* Double Ended, Doubly Linked Link List
* Graph
* Hash table
* Heap
* Stack
* Queue

The following list and descriptions of algorithms were implemented within this program.

* Breadth-First Search:

The breadth-first search algorithm is implemented within the graph class and is a method of searching a graph. The search works by starting with a specified vertex of the graph and marking all other vertices as unvisited. The algorithm then visits all the vertices adjacent to the current vertex, before visiting all the unvisited vertices adjacent to those vertices and so on, until the end vertex is found. Each time a node is visited its previous field is set to the parent node, this allows the algorithm to back track from the end node until the starting node is found in order to find the shortest unweighted path.

* Depth-First Search:

The depth-first search algorithm is implemented within the graph class and is a method of searching a graph. The search works by starting with a specified vertex of the graph and marking all other vertices as unvisited. The algorithm then visits the first adjacent unvisited vertex setting it as the current vertex, before visiting the first unvisited vertex that is adjacent to that current vertex. If no unvisited vertex is found adjacent to the current vertex the algorithm back tracks until an adjacent unvisited vertex is found. This process is repeated until no unvisited vertices are found and the path in which the vertices were visited is returned.

* Dijkstra’s Algorithm:

Dijkstra’s Algorithm is a modified version of the breadth-first search algorithm that calculates the distance from a starting vertex to all other vertices. This search works by starting with a specified vertex of the graph and marking all other vertices as unvisited. The algorithm then visits the vertices adjacent to the current vertex in order of shortest distance to longest distance, before visiting all the unvisited vertices adjacent to those vertices in a similar pattern. This process is repeated until the end vertex is found. Each time a node is visited its previous field is set to the parent node and an accumulative distance from the starting vertex. This allows the algorithm to back track from the end node until the starting node is found in order to find the shortest path and distance.

* Nearest Neighbour Algorithm

The variant of the Nearest Neighbour Algorithm used in this program is known as the linear search or naive variant. This form of search takes a specified vertex in a fully connected graph and moves to the adjacent vertex with the shortest distance. The search then moves to the adjacent vertex with the shortest distance from this new vertex and so on until all vertices have been visited. The shortest path in which the vertices were visited is then returned.

# Description of classes

## Double Ended, Doubly Linked Link List (linkedList.py):

This class file consists of two classes the DSADoublyListNode and the DSADoublyLinkedList class. The main class of this file is the DSADoublyLinkedList which is a double ended, doubly linked link list that stores a series of DSADoublyListNodes. This type of link list contains pointers to the head and tail of the list, along with pointers to the next and previous nodes within the nodes themselves. This allows iteration from head to tail or tail to head over the list.

The DSADoublyListNode consists of an object that contains a value, a pointer to the next DSADoublyListNode and pointer to the previous DSADoublyListNode. Along with these private attributes the class also contains the relevant getter and setter functions.

The DSADoublyLinkedList consists of an object that contains a pointer the starting DSADoublyListNode of the list, a pointer to the end DSADoublyListNode of the list and a count of how many nodes are in the list. Along with these attributes the class contains the following functions:

* isEmpty: Checks if the list is empty, returning a Boolean value as required.
* insertFirst: Creates a new DSADoublyListNode with the input value and inserts it at the beginning of the list.
* insertLast: Creates a new DSADoublyListNode with the input value and inserts it at the end of the list.
* peakFirst: Returns the first DSADoublyListNode in the list.
* peakLast: Returns the value in the last DSADoublyListNode in the list.
* removeFirst: Removes the first DSADoublyListNode from the list and returns its value.
* removeLast: Removes the last DSADoublyListNode from the list and returns its value.
* remove: Removes the DSADoublyListNode with the value corresponding to the input value from the list. This function only removes objects from the list with a “getValue()” function (specialised for UAV program).
* printList: Prints the value of all DSADoublyListNode in the list from head to tail.
* \_\_iter\_\_: Iterator function that allows iteration over the list using a for loop.

## Heap (heap.py)

This class file contains two classes the DSAHeapEntry and the DSAHeap class. The main class of this file is the DSAHeap which is a priority heap that only stores elements of the highest priority.

The DSAHeapEntry has two attributes, a value and a priority, along with the relevant getters and setters.

The DSAHeap also has two attributes, a heap consisting of an empty array with a size of 10 000 and a count that stores the number of DSAHeapEntrys stored on the heap. Along with these attributes the class contains the following functions:

* add: Checks if the new DSAHeapEntry priority is equal or greater than the current entries and adds it to the heap.
* remove: Removes the item at the top of the heap and returns its value. The item is replaced with the entry at the bottom of the heap which is then trickled down.
* isAdded: Checks if a value has already been added to the heap.
* display: Prints all the values of the DSAHeapEntrys in the heap from top to bottom.
* trickleUp: Reorders the DSAHeapEntry based on its value and the DSAHeapEntrys above it.
* trickleDown: Reorders the DSAHeapEntry based on its value and the DSAHeapEntrys below it.

## Hash table (hash.py)

This class file contains two classes the DSAHashEntry and the DSADoubleHashTable class. The main class of this file is the DSADoubleHashTable which is a resizable hash table containing a series of DSAHashEntrys. This type of hash table uses a double hash function to deal with collisions.

The DSAHashEntry contains three attributes only (no functions), a key, a value, and a state.

The DSADoubleHashTable contains a lfUpper and lfLower bound which defines the upper and lower bound for the load factor before resizing, a count indicating the number of DSAHashEntrys in the table, a max step used in the second hashing function and a hash array that stores a series of objects. The methods contained within the class are as follows:

* put: Creates a new DSAHashEntry and uses the input key and hash functions to place it within the table. Checks if the table is too full and resizes it accordingly.
* get: Returns the value of the DSAHashEntry with the corresponding input key.
* remove: Removes the DSAHashEntry with the corresponding input key, but leaves the state unchanged.
* getLoadFactor: Calculates and returns the load factor of the table based on the count and table length.
* export: Prints all the values of the DSAHashEntrys in the table with a state of 1.
* resize: Copies the data from the old table and creates a new table with a load factor within the bounds.
* hash: Returns an index for the hash table based on Bernstein hash function.
* stepHash: Returns a second index for the hash table based on the max step and previous hash index.
* findNextPrime: Finds and returns the next prime number closest to the starting value.

The benefits of a hash table over an array or list is that it combines the best features from both data structures. Arguably, the best attribute of an array is its ability to directly access an item without having to iterate over any other items within the array. The downside of an array, however, is that it has a fixed size and copying the array can be computationally slow. A list on the other hand is the direct opposite of an array, it has no limit to its size, but it cannot directly access the items within it. The ability of a hash table to resize allows it to combine these benefits resulting in an overall ‘better’ data structure.

NOTE: Resizing a hash table can be quite time costly and thus a hash table is not always the best data structure to use

## Stack (stackQueue.py)

The class DSAStack makes use of a link list to create a data structure with a last in first out behaviour. Along with this attribute the DSAStack makes use of the following functions to define this behaviour:

* isEmpty: Checks if the stack is empty, returning a Boolean value as required.
* push: Uses the insertFirst function of the link list to put a new item on top of the stack.
* top: Uses the peekFirst function of the link list to return the value of the top item in the stack.
* pop: Uses the removeFirst function to remove the item on top of the stack and return its value.
* printStack: Prints all values of the items in the stack from the top to the botom
* \_\_iter\_\_: Iterator function that allows iteration over the stack using a for loop.

## Queue (stackQueue.py)

The class DSAQueue makes use of a link list to create a data structure with a first in first out behaviour. Along with this attribute the DSAQueue makes use of the following functions to define this behaviour:

* isEmpty: Checks if the stack is empty, returning a Boolean value as required.
* enqueue: Uses the insertLast function of the link list to put a new item at the front of the queue.
* dequeue: Uses the removeFirst function to remove the item at the front of the queue and return its value.
* peek: Uses the peekFirst function of the link list to return the value of the item at the front of the queue.
* printStack: Prints all values of the items in the queue from the front to the end.
* \_\_iter\_\_: Iterator function that allows iteration over the queue using a for loop.

## Graph (graph.py):

This class file contains three classes DSAGraphEdge, DSAGraphVertex and DSAGraph. The main class in this file is the DSAGraph, which is a graph that makes use of DSAGraphEdge and DSAGraphVertex to store its nodes and connections.

The DSAGraphEdge consists of 3 attributes, from a pointer to the starting vertex, to a pointer to the end vertex and a value. The class also contains the relevant getters for these attributes.

The DSAGraphVertex contains a label, a value, a list of links to store adjacent vertices, a visited flag, a distance, and a previous pointer used in Dijkstra’s algorithm. The functions of this class include the relevant getters and setters, as well as a function to add a vertex to the adjacency list.

The DSAGraph only contains two attributes, a list of vertices and a list of edges (that use the link list class). The functions that define the classes behaviour is as follows:

* addVertex: Creates a new DSAGraphVertex and adds it to the graph.
* deleteVertex: Removes the vertex with the corresponding input label and all of its connections.
* addEdge: Adds a connection between vertex with label 1 and 2 with a value of input value.
* deleteEdge: Removes the connection between the locations with the input labels.
* hasVertex: Checks if a vertex exists with the input label and returns the corresponding bool.
* getVertexCount: Calculates and returns the number of vertices within the graph.
* getEdgeCount: Calculates and returns the number of edges within the graph.
* getVertex: Returns the vertex from the graph with the input label.
* getAdjacent: Returns a list of vertices adjacent to the vertex with the input label.
* isAdjacent: Checks if one location is next to another.
* displayAsList: Prints the adjacency matrix of the graph.
* depthFirstSearch: Returns the resulting path of the depth-first search on the graph.
* doBFS: Returns a hash table containing the parent vertices of each location from the breadth-first search.
* buildPath: constructs the shortest path between locations from the breadth-first search.
* breadthFirstSearch: Returns the resulting path of the breadth-first search on the graph.
* dijkstraSearch: Calculates the distance of all locations from the starting location.
* doDijSearch: Returns the resulting path and distance of the Dijkstra search on the graph.
* nearestNeighbour: Returns the resulting path of the naïve nearest neighbour search on the graph.

# Testing Methodology:

## Individual Classes:

### Graph:

To individually test the graph class, the following test case was developed. It consists of four fully connected vertices as shown in the image below. The expected results of this test are as follows:

#### Adjacency List:

1

1

1

1

0.5

3

A | B C D

B | A C D

C | A B D

D | A B C

#### Depth-first search:

A->B->C->D

#### Unweighted breath-first search (A to C):

A->C

#### Weighted breath-first search (A to C):

Path: A->B->C Distance: 2

#### Nearest neighbours

A->B->D->C

### Hash:

To individually test the hash class, the following test case was developed:

A hash table initialised with a starting value of 6. Seven entries (1-7) will then be added forcing the hash table to resize. Four entries (4-7) will then be removed to force the table to resize again.

The load factor, export and get functions will also be called throughout the test.

NOTE: This test is taken from Prac07

### Heap:

To individually test the heap class, the following test case was developed:

A heap will be initialised and three entries (A-C) with a priority of 1. Three more entries will then be added (X-Z) with priority of 3 forcing the old entries to be removed. All the entries will then be removed.

The display function will also be called throughout the test.

NOTE: This test is taken from Prac08

### Link List:

To individually test the link list class, the following test case was developed:

A link list will be initialised and checked if empty (should be true). The value ‘a’ then ‘b’ should be inserted first while a DSADoublyListNode with value ‘c’ followed by just the value ‘d’ will be inserted last. The test will then peek first and last, before removing the object with value ‘c’. The rest of the values will then be removed, and the list will be checked if it is empty. The print list function will also be called throughout the test.

NOTE: This test is taken from Prac04

### Stacks and Queues:

To individually test the stack and queue classes, the following test case was developed:

A stack and queue will be initialised and checked if empty (should be true). The values of an array from 1-10 will be added individually to each data structure. They will then be checked if empty again (should be false) and the top or front of each structure will be peeked. Finally, all the items will be removed and the stack and queue will be checked if they are empty again (should be true).

The print list function will also be called throughout the test.

NOTE: This test is taken from Prac04.

### UAV:

Since this is a basic object developed to store the location of a UAV, a very primitive test harness was designed. This test harness consists of a main function that initialises a new UAV object and prints the object to the terminal to confirm its initialisation. Next, the harness asks the user to input a location, which can be any string, to test the setter method. To confirm the setter’s success and to test the getter method of the object, the test harness calls the getter and prints the output to the terminal. If this output is equal to the users input the test is considered a success.

## Main Program:

### Current data:

|  |  |  |
| --- | --- | --- |
| **Files** | **Location File** | **Data file** |
| **Name of File** | location.txt | UAVdata.txt |
| **Contents** | 11 15  A B 3.5  A C 2.1  A E 1.8  B C 4.2  B F 2.5  C D 1.3  C G 3.1  D H 2.9  E F 1.2  E G 2.6  E I 3.4  F H 1.9  G H 3.5  G J 2.8  I J 2.2 | A 32 45 90  B 26 50 35  C 38 55 75  D 45 30 80  E 29 40 65  F 31 20 85  G 42 60 50  H 36 25 95  I 27 50 40  J 33 35 60 |

### Unconnected node:

### Large data set:

# Results

This section shows the output of unit tests.

## Individual Classes:

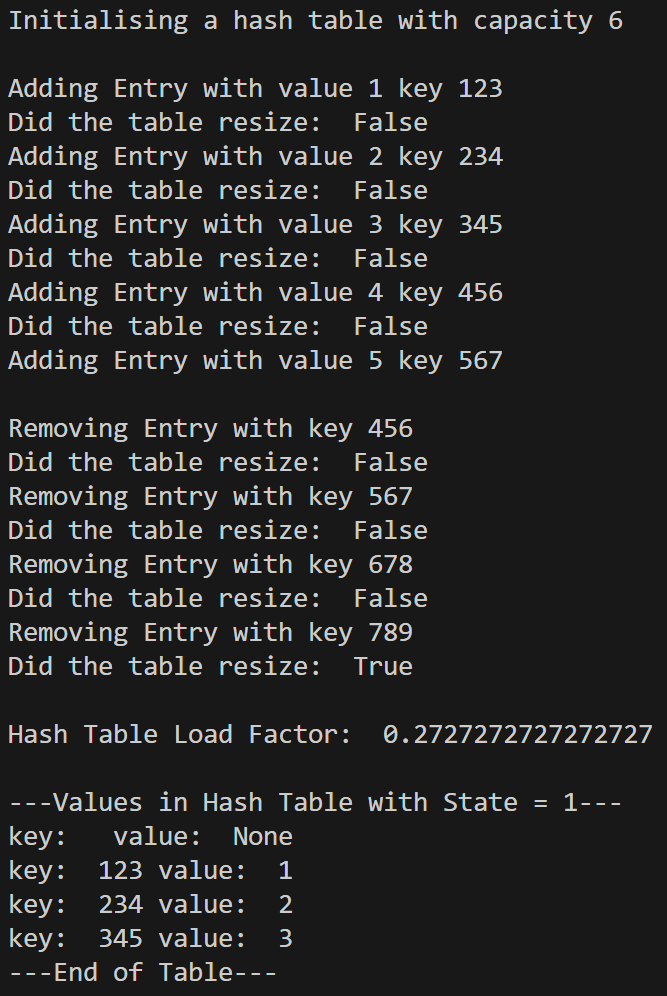
As shown by the images below, all tests behaved as expected, including the link list where an object was purposely added to test the remove function.

### Graph:

A screenshot of a computer program

Description automatically generated with medium confidence

### Hash:



### Heap:

A screenshot of a computer program

Description automatically generated with medium confidence

### LinkList:

A screenshot of a computer screen

Description automatically generated with medium confidence

### Stacks and Queues:

A screenshot of a computer program

Description automatically generated with medium confidence A screenshot of a computer program

Description automatically generated with medium confidence

### UAV:

A black screen with white text

Description automatically generated with low confidence

## Main Program

# Future Improvements

Better search algorithm

Itinerary include intermediate steps