Approach -

1. Initially, we approached this problem as the find route program from assignment 0. We introduced a fringe to keep track of current visiting city, current distance travelled, total probability of accident and total time travelled. We introduced a function named routes to find out all possible cities that can be travelled to and from a provided city, and a function named ‘probability\_of\_accident’ to calculate the cost of accident of travel in a selected route and another function named ‘travel\_time’ to calculate the time taken to travel in a selected route. We maintained a list of travelled\_routes so we do not have to travel between two cities twice. There were two issue with this approach – First the computation speed was slow for cities that are very wide apart, another is that we might be able to find an shorter intermediate route from between two cities in within our original path, but since we were not allowing to travel the same routes twice, we were missing out on these better routes.
2. To improve the computation time of the program, we decided to add another function that prioritized the next cities to travel which were in the direction of the goal city. We added a factor of 1.2 to the Euclidean distance between the two cities as a shorter route might be available via a city which might be farther from the goal. This did improve the computation time; however, we were still missing out on the best path.
3. After some googling around, we decided to implement the Djikstra’s algorithm to find the shortest route. This helped us overcome the issue of updating a shorter route if found between two cities. This [article](https://benalexkeen.com/implementing-djikstras-shortest-path-algorithm-with-python/) gave us clarity how to approach implementing Djikstra’s algorithm. When a shorter route is found between two cities, the shorter path is updated in our ‘shortest\_route’ dictionary.
4. State space – All cities that are available in the city-gps.txt file

Successor function – Set of all possible cities that has a route to travel to/from the current city

Edge weight – The number of cities, distance, time, accident probability between two cities are considered as the edge weights based on the selected cost function of segments, distance, time, cycling respectively

Initial state – The given start city

Goal state – The given end city

Heuristic function - f(x) = g(x) + h(x), where f(x) = total cost to travel from the start city to a given city, g(x) = total cost of travelling to a parent city from the start city and, h(x) = total cost to travel to a next city from current parent city, where the cost could be segments, distance, time or accident probability based on chosen cost function. The heuristic is optimal as it manages to find the shortest route between two cities for a selected cost function, and it is admissible as it will never overestimate the path to the goal by finding a longer route if a shorter route is available.

Code Explanation –

1. The main function reads the city-gps.txt file and road-segments-txt file. Then validates if the input cities and cost function provided are valid or not. Next, we call the ‘find\_path’ function.
2. Here, we initialise a dictionary named ‘shortest\_path’ with the start\_city name as the key and having the following value (None, 0, 0, 0, 0), and the shortest\_path a stores dictionary of 5 values–
3. The parent city from which we arrived at the current city
4. Cities/segments travelled
5. Distance travelled
6. Time travelled
7. Expected Accident

We set current\_city as start\_city and initialise an empty set named ‘visited\_cities’ to track all the visited cities.

1. Then we run the while loop till we find the end\_city and add the current\_city to visited city list. We get all possible cities that can be travelled using ‘routes’ function along with other details like the distance, time and accident. If a child city is not available in the shortest\_path list, we add the new city to it. And if it is available, based on the chosen cost function, we check if the current\_shortest\_value to the current city is less than any other route that has previously been found to the current city, and update the shortest\_path list with the new current\_value if it is less than the previous one [Line 50-65].
2. The next task if to select all the cities that can be travelled and have not been travelled already. [Line 68]
3. Then we select the city with the least value of the selected cost function as our current\_city. [Line 74-81] And then we rerun the while loop till we find the end city.
4. Once we find the end city, we break out of the while loop and construct the shortest route that is stored in routes []. [Line 84-91]
5. Finally, we call the ‘find\_routes\_detials’ function that calculates all the total costs alone the found shortest\_route according to the provided cost function.

Note : We could upgrade the current code by preferring the next cities to travel based on how close they are to the goal, by calculating Euclidean distance using the coordinates of the cities.