1-D Android Project: Travel App

Cohort 2 Group 12 | Ho Reuben | Lim Jing Yun | Wang Nian Yu. Cyrus

Our app is targeted at animal-lover tourists. It implements the following functions:

1. **Recycler View** – to display the list of attractions available and the itinerary list which consists of attractions selected by the user.
2. **SQLiteDatabase** – to store the list of available attractions, Place IDs and descriptions. The database is queried when displaying the list of attractions and itinerary list.
3. Querying suitable APIs
   1. **Google Places API** – to obtain Place objects of the attractions from the locations’ Place IDs. Information about each location (address, latitude, longitude, website) is obtained via the API and displayed below the Map Fragment of the attraction.
   2. **Google Maps API** – to display attractions on a Map Fragment.
4. **Implicit Intent** – to view each location’s web page. The implicit intent is launched when the user clicks on each location’s website link.
5. **Function 2 (Tourist Attraction Locator)** – The user can search for their desired attraction by utilizing a search bar at the top of a list of attractions. The search bar works in two ways:
   1. The attraction list changes dynamically and only shows the attractions with names containing the string the user entered in the search bar.
   2. If no attractions appear, the user can click the search button, and the app will predict and suggest a list of attractions the user might have been searching for. The Longest Common Substring and Lowenstein Distance algorithms were used to implement this prediction.

Details are displayed once the user clicks on an attraction in the list.

**Function 3 (Daily Itinerary Planning)**

For the itinerary planner, all costs and times needed to travel between attractions are stored in the form of adjacency matrixes. Data was collected from gothere.sg and maps.google.com. Both our brute force solver and fast solver take a budget as an argument, and will find the fastest route that is within the budget.

The brute force solver enumerates through all the possible routes and transport modes exhaustively, and selects the one with the minimal total time within the budget constraint. It does this by recursively creating and testing all possible permutations of the supplied list of attractions.

The fast solver we have implemented uses Simulated Annealing, a randomization heuristic based on neighbourhood search that permits moves that make a solution worse. By allowing worse solutions, SA avoids the problem of getting stuck in local minima that naïve hill climbing faces. We initialize the solver with a large starting “temperature”, which determines the probability of accepting a worse solution. At each simulation step we randomly swap two attractions in the traveling order. If the new route is better than the previous best, we save it as the best. Otherwise, we check if the Boltzmann factor is less than a random value, or . If yes, we revert the swap of the cities. If not, we keep the new order of the cities. We end each step by reducing the temperature by a cooling factor. The probability of moving to a worse new solution is thus progressively changed towards zero, and the solver stops when temperature is very low. By moving from a large potential search range to a small one, SA can estimate a solution close to the global optimum.

The brute force algorithm runs at factorial time or O(n!), as there are n! permutations of routes for n number of attractions. While it can solve the given example of 6 attractions with an average execution time of 3ms, this time scales up quickly with every additional attraction added. With 10 attractions, the average execution time is 1357ms. With 20 attractions, the number of attractions in our app, it is expected to take 77 years. This is obviously impractical for a tourist.

The fast solver, on the other hand, runs at constant time. The average execution times for 6, 10 and 20 attractions are each 7ms. However, the fast solver does not always provide the fastest possible route, and the discrepancy between the estimation and the actual fastest route becomes worse as the number of attractions increase. For example, while with 6 attractions, the fast solver always produces the fastest possible route, with 10 attractions, the average route produced took 784.2 minutes, but the actual optimal route takes 763 minutes. However, this error is acceptable, as it is marginal in comparison to the benefits of a much shorter runtime.