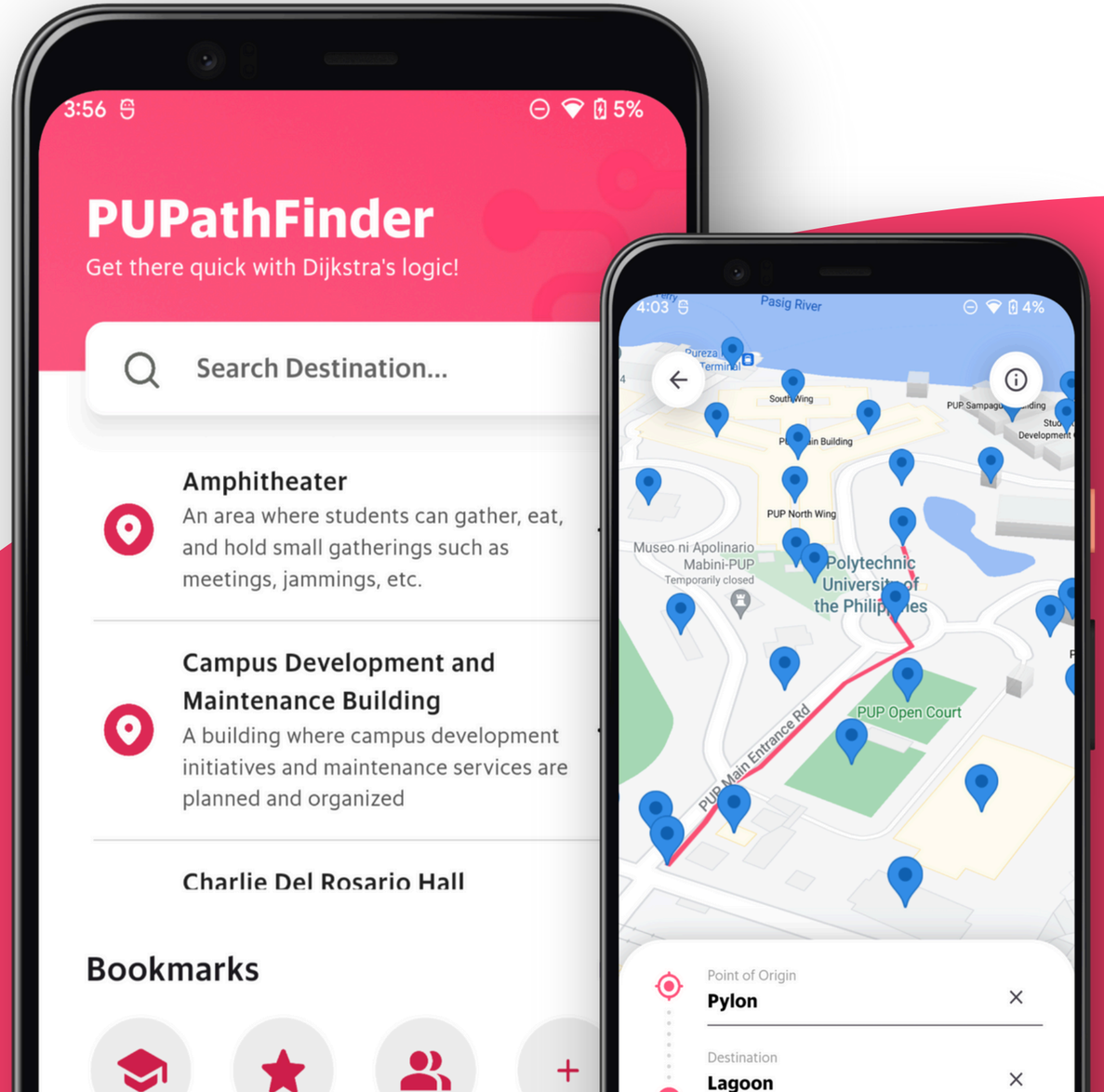
PUPATHFINDER

CARLOS DOMINGO LAGMAN RUSTIA

A Campus Navigation System for Shortest Routes between Facilities in PUP Mabini Campus using Dijkstra's Algorithm

Design and Analysis of Algorithms

2024



Abstract

With the objective of providing a campus navigation system demonstrating Dijkstra's algorithm to help students, faculties, and people entering the Polytechnic University of the Philippines (PUP) - Mabini Campus go to their desired destination in the shortest way possible, the project PUPathFinder demonstrates a mobile application developed with Flutter framework and Dart language. It works by allowing the user to search for their source and destination nodes, then the app displays the shortest route based on Dijkstra's algorithm. The results show an effective performance of the algorithm for PUP Mabini Campus in terms of response time, where it averaged 0.7420 milliseconds (ms) recorded through the program execution where it displayed the response time in milliseconds, and accuracy rate, where it recorded a 104.2641%, between the manual calculation and algorithm calculation.

PUPATHFINDER

Background

Noone, R. (2020). Going in a different direction: Critical arts-based approaches to Google Maps. Survival: A Journal of Comparative Studies in Society and History, 62(5), Article 11293. https://doi.org/10.5210/spir.v2020i0.11293

Limitations of Google Maps

Google Maps, a widely-used Geographic Information System (GIS) platform, has become essential for contemporary needs like reliable navigation. Over 1 billion people rely on it, especially for travel. However, for local areas like campuses, the lack of user-contributed map data can make it unreliable for directions. [1]

Potential Problems in PUP Campus

Navigating Polytechnic University of the Philippines - Mabini Campus can be challenging, especially for freshmen and visitors unfamiliar with the infrastructure. Faculty and staff may also experience arriving late and may need to take the shortest route to their point of destination.

Objectives

- To provide a campus navigation system, especially for freshmen students and visitors who might be unfamiliar with the campus facilities and paths.
- To enhance the efficiency of locating rooms and facilities by displaying the shortest path, thus saving time and reducing confusion
- To suggest an accessible way for individuals to find facilities within the campus

Statement of the Problem

- How can Dijkstra's algorithm be efficiently and effectively implemented in a mobile application to provide real-time shortest path navigation for users within the Main Campus?

- What are the challenges and limitations in integrating Dijkstra's algorithm into a campus navigation system?

- How can a campus navigation system ensure accurate and reliable route suggestions, considering the dynamic changes in campus infrastructure such as construction or facility closures?

Review of Related Literature

Dijkstra’s Algorithm

Rachmawati and Gustin (2020) conducted a comparative analysis of Dijkstra's algorithm and the A* algorithm in solving the shortest path problem. A* algorithm, which employs the heuristics method, can also be used for finding the shortest path. Their findings indicated that while both algorithms performed similarly on small to medium-scale maps, the A* algorithm showed a minor advantage in efficiency for larger-scale maps due to its heuristic approach. However, since this project will not be dealing with a regional scale map, Dijkstra's algorithm will be enough. These two algorithms perform almost the same, only differing by a fraction of nanoseconds in terms of the running time. Therefore, Dijkstra's algorithm remains a reliable option for applications requiring guaranteed optimal solutions without heuristic bias.

Review of Related Literature

Campus Navigation

The paper "Coimbatore Institute of Technology Campus navigation" discusses the developmenet of a campus navigation system that uses digitial maps and real-time data to find optimal paths within the campus. It aims to provide accurate navigation guide to users through a user-friendly interface. Leaflet.js library is used in this research for map rendering and geographical data display. MongoDB is used for storing feedback data, and the web pages are developed with HTML/CSS, Javascript, and React.js. For the data integration, GeoJSON data are parsed in building and pathway coordinates extraction. Their application also includes supplementary features such as orientation assistance for visitors and conference attendees and timely emergency navigation.

Review of Related Literature

Campus Navigation

Another promising campus navigation paper is "An Indoor Campus Navigation System for Users with Disabilities", which presents a solution for indoor navigation, especially for those users who have disabilities. The authors address the challenges of indoor positioning and navigation with a system based on Bluetooth Low Energy (BLE) transmitters and a hierarchical map of the building. This system enables users with smartphones to read signals from these BLE transmitters, allowing them to determine their position within the building. It also includes a server application with a database and API, and a mobile application designed for various users, including those with visual, auditory, and mobility impairments. The focus is on providing accurate positioning and safe navigation, especially during emergencies.

Review of Related Literature

Campus Navigation

(cont.) Existing indoor navigation technologies including streovision camera, image recognition, lidars, and Ultra Wideband (UWB) are discussed as well as their limitations, highlighting the advantages of BLE transmitters with their low cost and long operation time. Potential users are then included in the preliminary research as validation of the system's efficacy.

Review of Related Literature

Campus Navigation

A study by Dusakaeva et al. (2023) proposed a mobile app for finding optimal routes within a university campus, using three well-known graph algorithms and various campus buildings as vertices. The app included an auxiliary service for graph formatting. It introduced features like map manipulation and suggested combining different algorithms for increased navigation accuracy.

Review of Related Literature

Flutter

Flutter is a useful open-source software development kit (SDK) that enables cross-platform application development. It was developed by Google and chosen as their application-level framework for the succeeding applications on the next generation to cut cost and complexity of application creation in Android and iOS. This kit utilizes the “Just-in time compilation” feature to compile the code while the program is running. It also features the hot reload feature in building user interfaces and fixing bugs. This hot reload feature functions as the tool that inserts the updated source code into Dart Virtual Machine (DVM) as this kit uses Dart as the programming language. Through this language, the framework automatically rebuilds the structure and immediately displays the changes in the work.

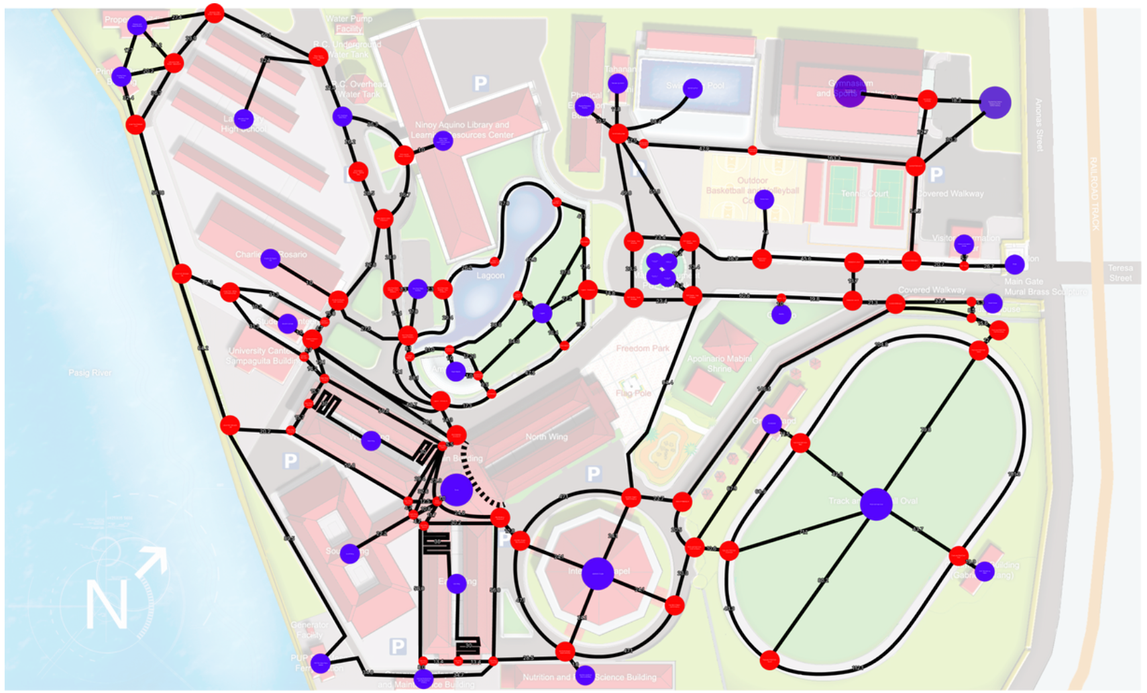
Methodology

Map

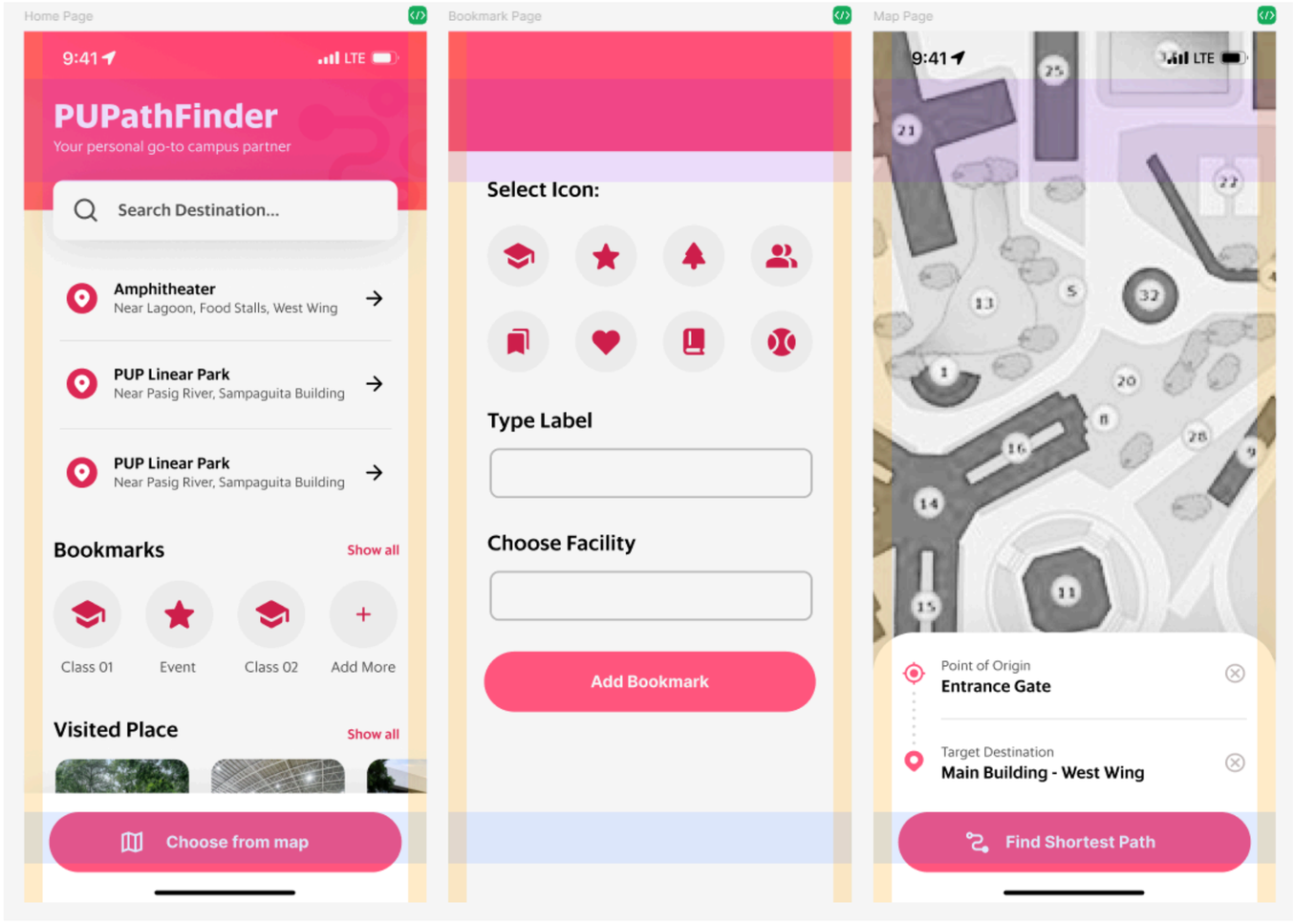
The preliminary stage starts with initial drafting of nodes and vertices. The nodes are the buildings and facilities around the campus, but do not include the interior rooms of these infrastructures. The edges are the paths connecting these infrastructures. Once the nodes and vertices are set, the weights of the edges are determined by using the Google Maps feature of selecting a point and another point in the map and it displays the distance in meters (m). This serves as the basis of the algorithm in the weights of each edge, which will be used in calculating the shortest path through Dijkstra's algorithm.

Methodology

Map



Methodology

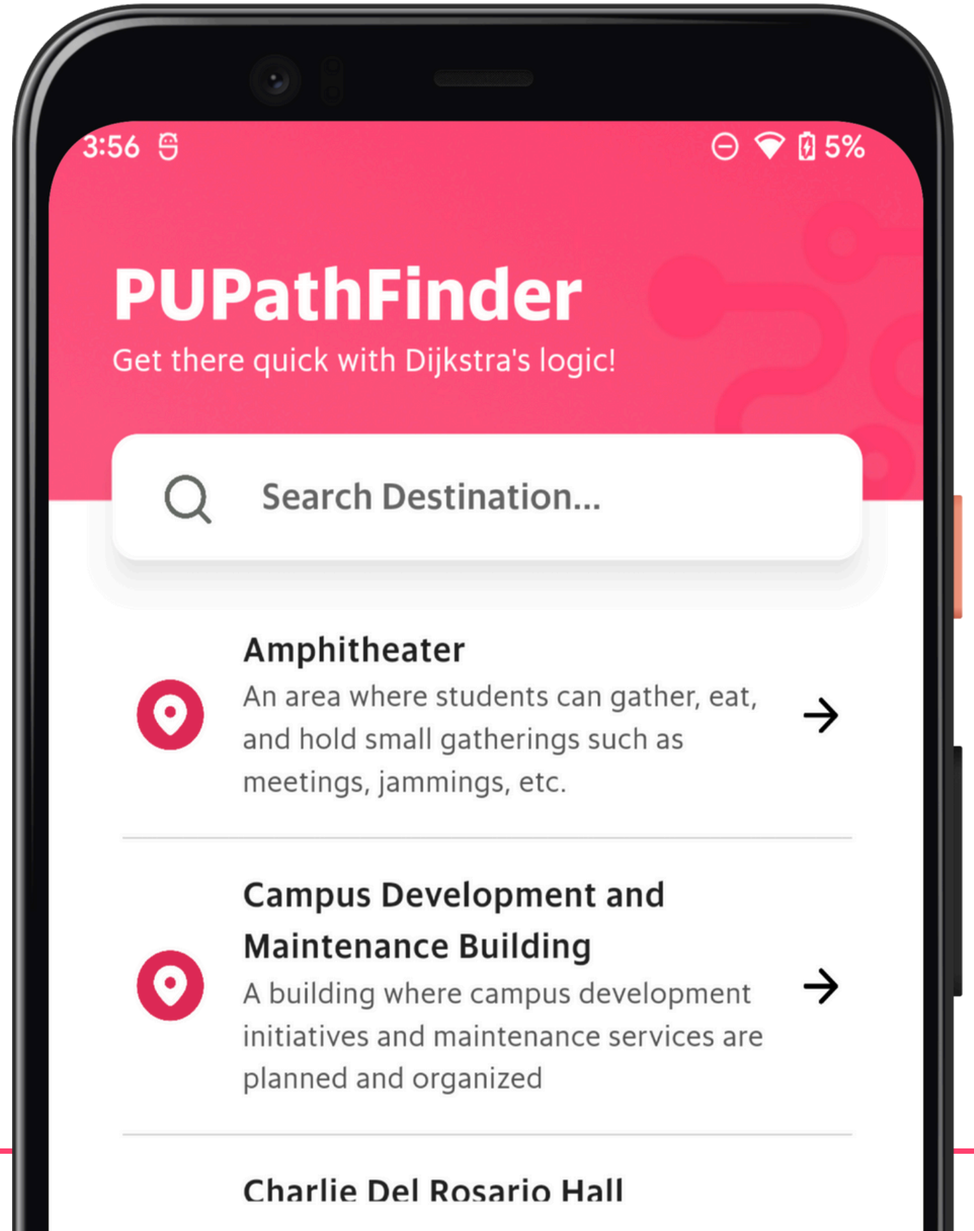


Methodology

Frontend

Home Page

- **Search Bar** - This allows users to search for a specific destination within the capus, while also filtering the facilities list based on the user input.

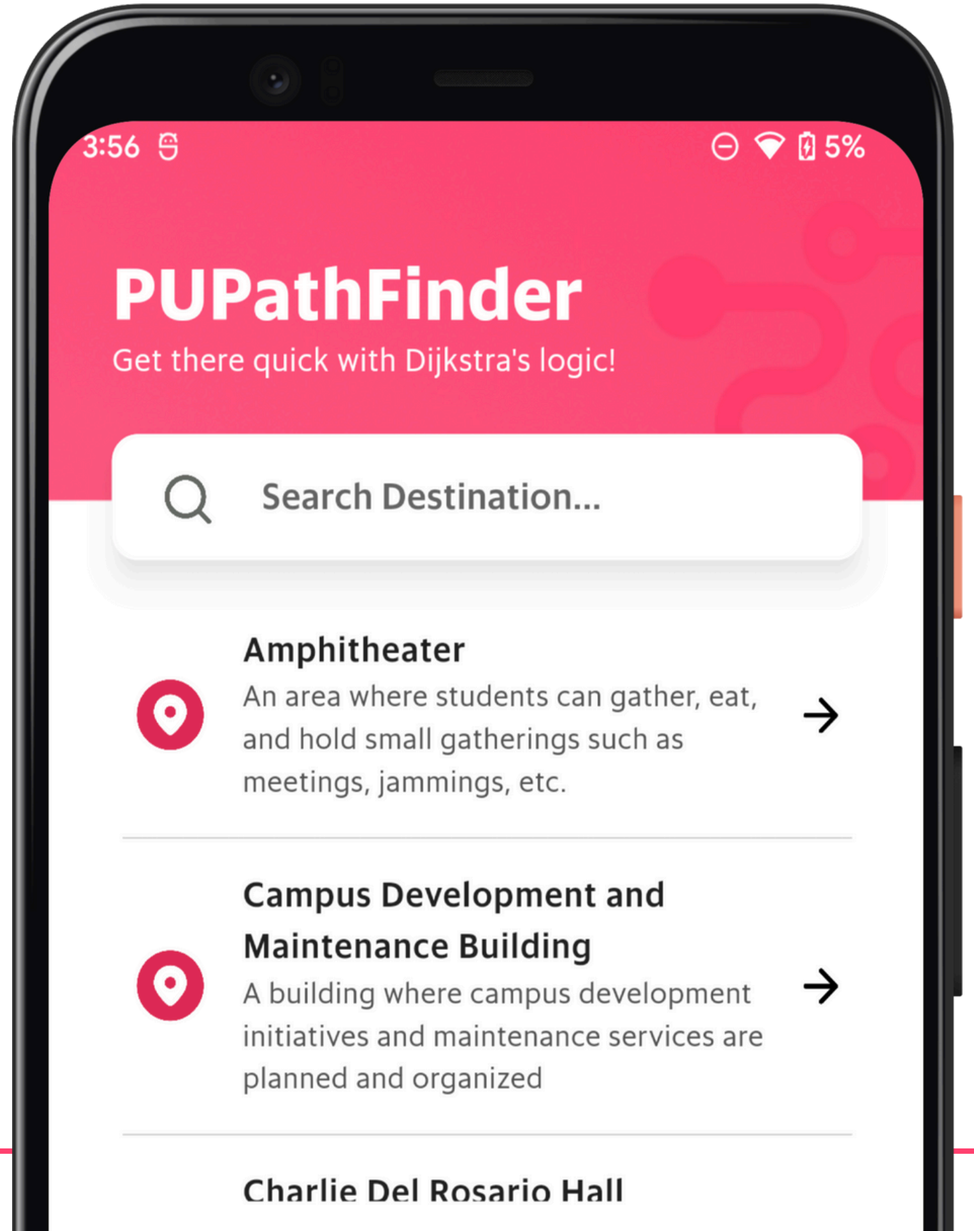


Methodology

Frontend

Home Page

- **Facilities List Section** - Located below the search bar, which dynamically displays the facilities based on user input in the search bar. All facilities are clickable, allowing users to quickly set their point of destination.



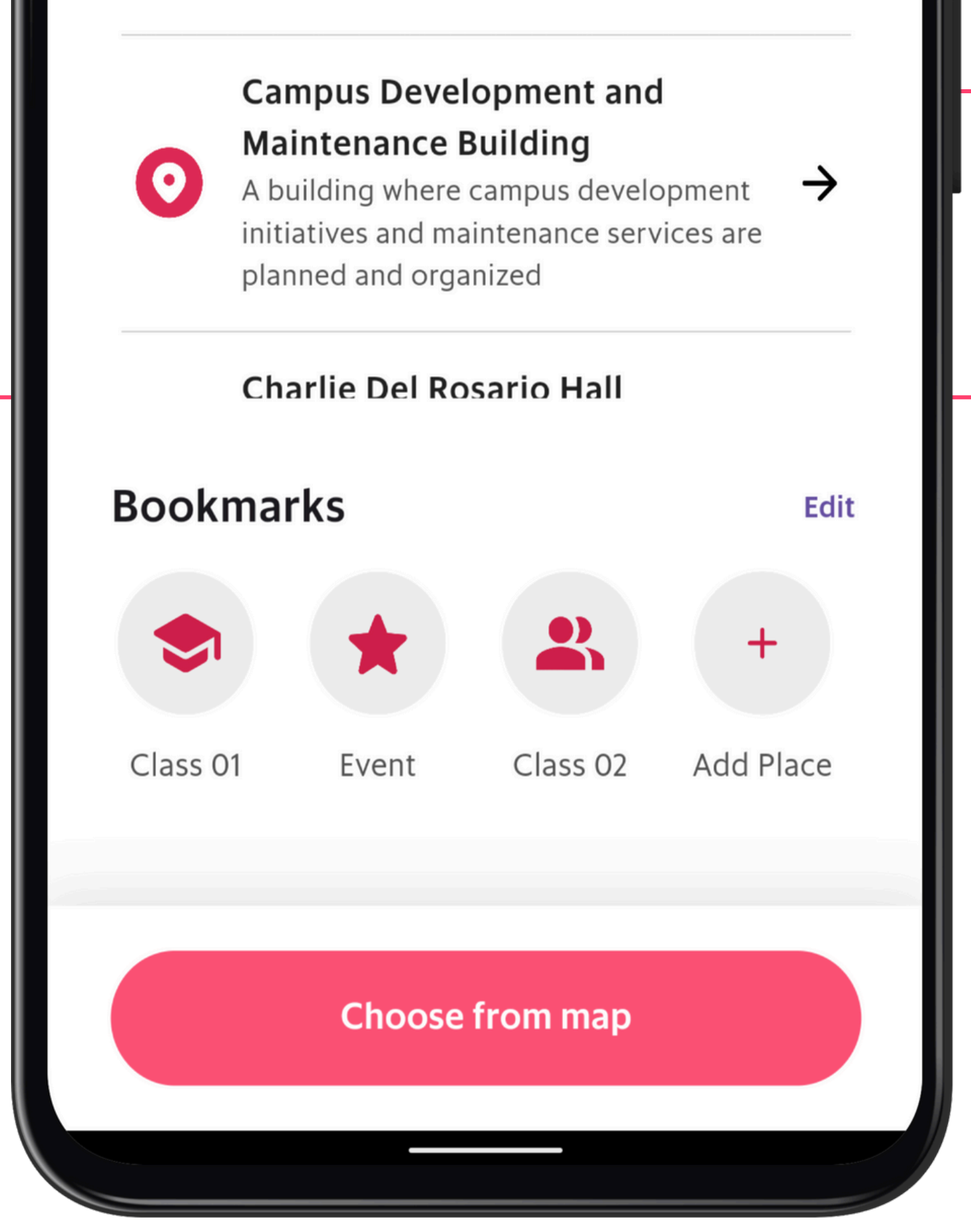
PUPATHFINDER

Methodology

Frontend

Home Page

- Bookmarks Section - This enables users to pin or bookmark frequently used facilties for quick access.

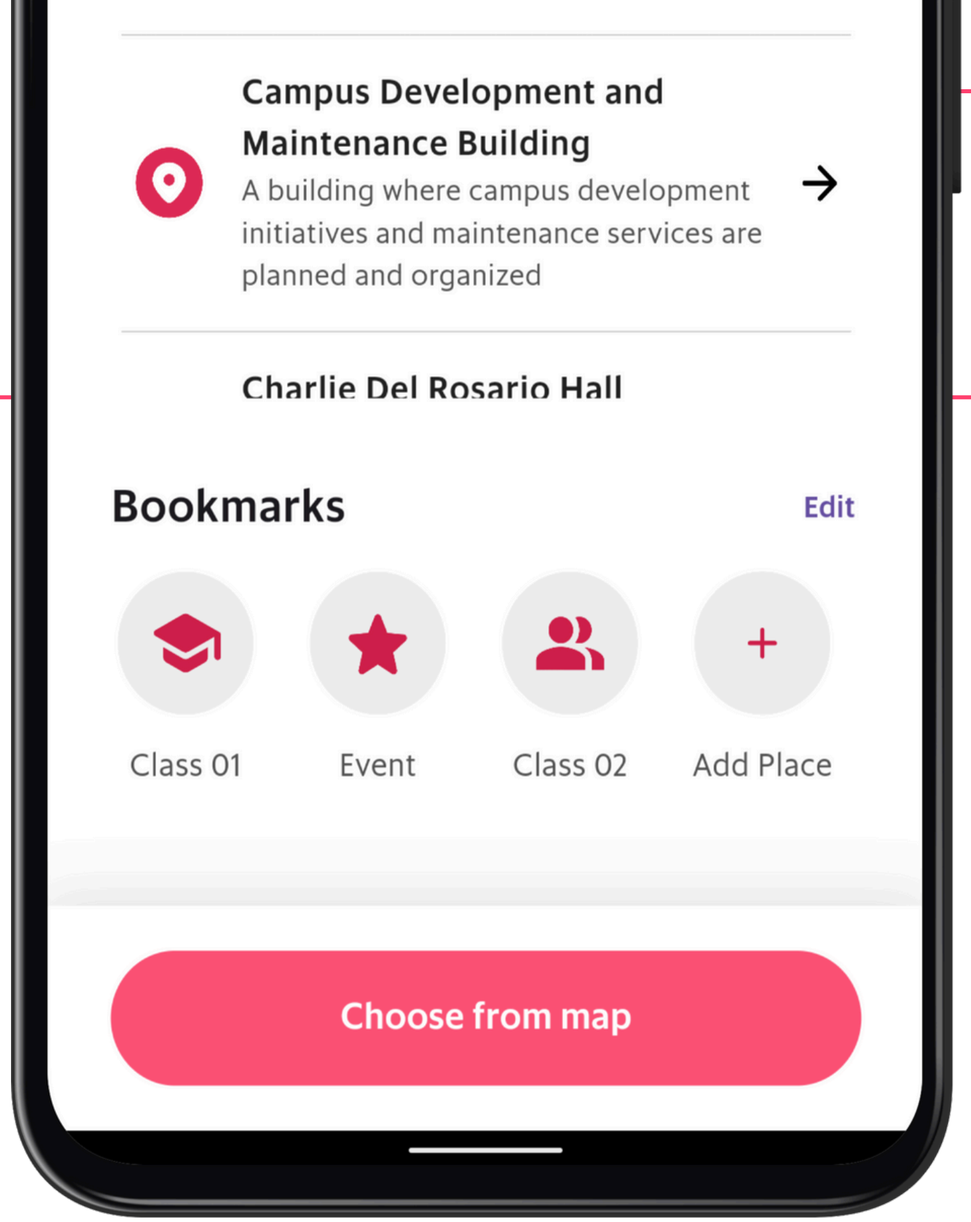


Methodology

Frontend

Home Page

- Primary Button - Labeled as "Choose from map," positioned at the bottom, which offers a direct link and access to the map page where users can navigate and select origin and destination points on the interactive map.



PUPATHFINDER

Methodology

Frontend

Map Page

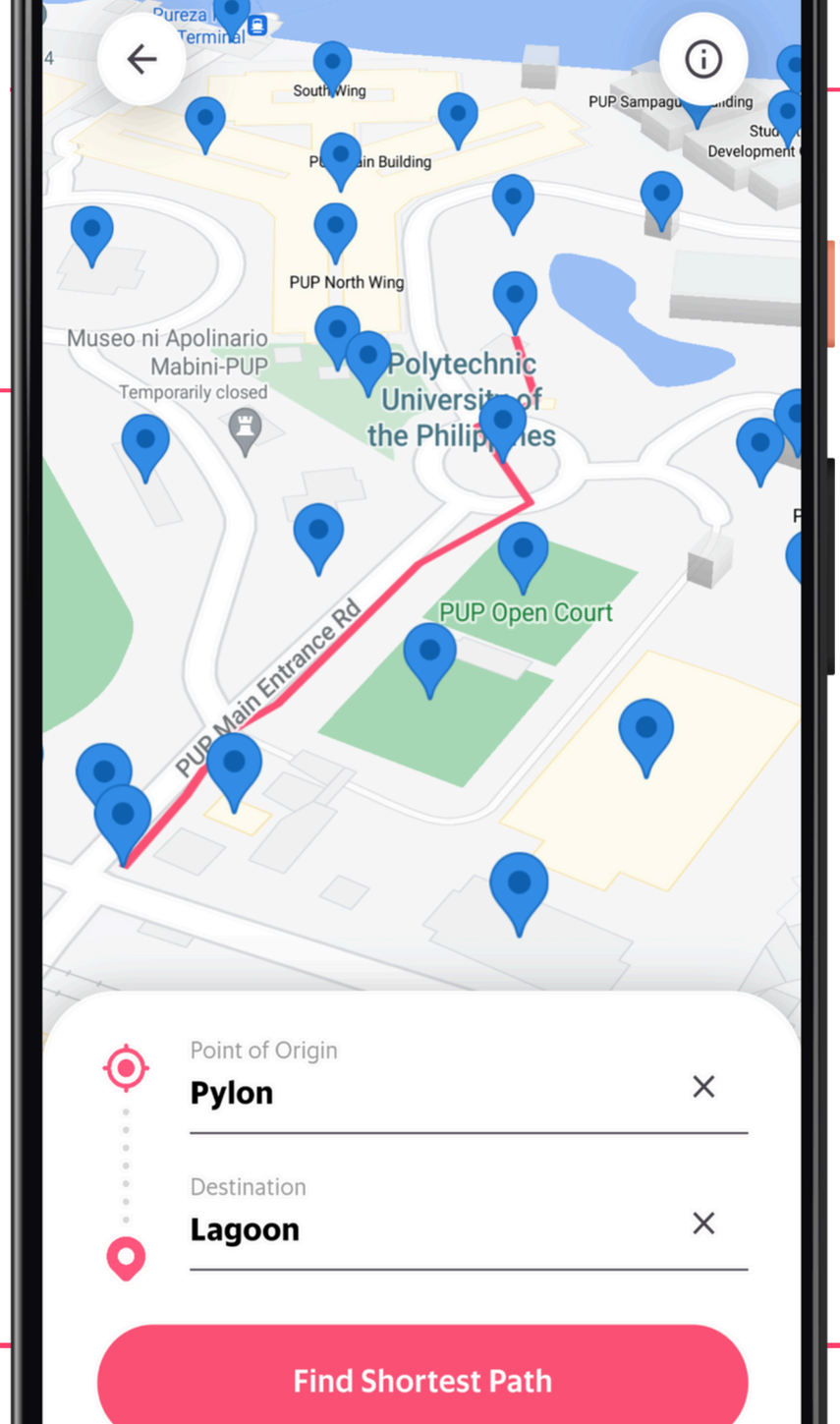
```dart
import 'package:flutter/material.dart';
import 'package:flutter_map/flutter_map.dart';
import 'package:latlong2/latlong.dart';
import '../../model/facilities_model.dart';
import '../../model/facilities_list.dart';
```

```dart
class MapPage extends StatefulWidget {
  final String origin;
  final String destination;
  final List<String>? facilityRooms;

  const MapPage({
    Key? key,
    required this.origin,
    required this.destination,
    this.facilityRooms,
  }) : super(key: key);

  @override
  State<MapPage> createState() => _MapPageState();
}
```

```dart
class _MapPageState extends State<MapPage>
```



PUPATHFINDER

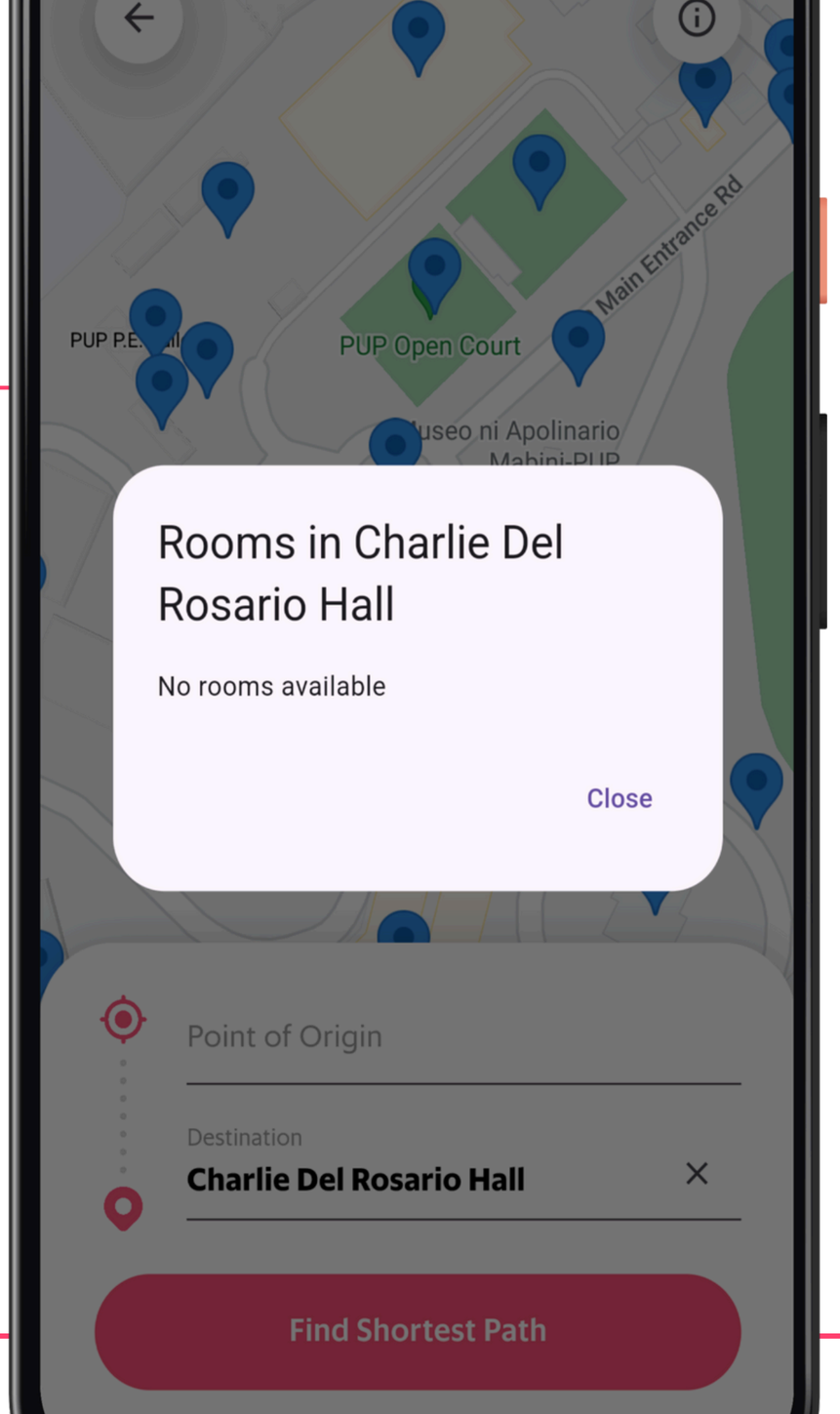
Methodology

Frontend

Map Page

```
void _showRoomsDialog(BuildContext context)
```

```
Widget build(BuildContext context)
```



PUPATHFINDER

Methodology

Frontend

Map Page

```
class MapsContent extends StatefulWidget
```

Methodology

Backend

Establishment of Algorithm Logic

Dijkstra's algorithm is chosen as the approach for finding the shortest path between vertices. The Pair class is defined to store the vertex and the weight so far (wsf) for each step in the algorithm. In the initial establishment of algorithm logic, the vertices and their connections are defined and stored using an adjacency list, implemented as an array of ArrayList<Edge>. Additionally, a HashMap<Integer, String> is used to map integer vertex identifiers to their corresponding location names.

Methodology

Backend

Establishment of Algorithm Logic

```java
public static void main(String[] args) {

        ArrayList<Edge> [] graph =  new ArrayList[200];
        for(int i = 0; i < graph.length; i++){
            graph[i] = new ArrayList<>();
        }

        HashMap<Integer, String> map= new HashMap<>();

        map.put(0, "Amphitheater");
        map.put(1, "Dome");
        map.put(2, "Grandstand");
        // ...

        graph[0].add(new Edge(0, 1, 14));
        graph[0].add(new Edge(0, 3, 5));

        graph[1].add(new Edge(1, 0, 14));
        graph[1].add(new Edge(1, 4, 2));
        graph[1].add(new Edge(1, 2, 6));

        graph[2].add(new Edge(2, 1, 6));
        graph[2].add(new Edge(2, 5, 1));
        graph[2].add(new Edge(2, 6, 7));
        // ...
```

Methodology

Backend

Establishment of Algorithm Logic

A priority queue is utilized to manage the vertices based on their current path weights, ensuring that the algorithm always processes the vertex with the smallest weight first. This priority queue is managed using the compareTo method in the Pair class, which compares the weights to maintain the correct order.

During the execution of Dijkstra's algorithm, the shortest path from the source vertex to all other vertices is determined by repeatedly selecting the unvisited vertex with the smallest weight and updating the weights of its neighboring vertices. This process continues until all vertices have been visited and the shortest paths have been established.

Methodology

Backend

Establishment of Algorithm Logic

```java
public static class Pair implements Comparable<Pair> {
        int vertex;
        int wsf; //weight so far

        Pair(int vertex, int wsf) {
          this.vertex = vertex;
          this.wsf = wsf;
        }

        public int compareTo(Pair other) {
            return Double.compare(this.wsf, other.wsf);
        }
      }
```

```java
int src = 0;
        boolean[] visited = new boolean[graph.length];

        PriorityQueue< Pair> prioQue = new PriorityQueue<>();
        prioQue.add(new Pair(src, 0));

        while (prioQue.size() > 0) {
          Pair topEle = prioQue.remove();

          if (visited[topEle.vertex] == true) {
            continue;
          }

          visited[topEle.vertex] = true;

          System.out.println("Vertex:" + " " +map.get(topEle.vertex) + ", " + "Shortest Path:" + " " +topEle.wsf);

          for (Edge edge : graph[topEle.vertex]) {
            if (visited[edge.nbr] == false) {
              prioQue.add(new Pair(edge.nbr, topEle.wsf + edge.weight ));
            }
          }
        }
```

Methodology

Backend

Graph file in Dart

Once the data is collected, the classification and labeling process involves organizing the vertices and edges. In the actual code, the edges are defined through a List class, where each source and destination nodes connections along with the edge weights are defined. The nodes themselves are defined using the Map class with their respective coordinates, which are used in the dispalying of polylines for the routes.

Methodology

Backend

Graph file in Dart

```dart
class Edge {
  final String source;
  final String destination;
  final double weight;

  Edge(this.source, this.destination, this.weight);
}

class Node {
  final String name;
  final double lat;
  final double lon;

  Node(this.name, this.lat, this.lon);
}
```

Methodology

Backend

Graph file in Dart

```
List<Edge> edges = [

  Edge("Amphitheater", "Amphitheater Entrance 1", 1.0),
  Edge("Amphitheater", "Amphitheater Entrance 2", 1.0),

  Edge("Amphitheater Entrance 1", "Amphitheater", 1.0),
  Edge("Amphitheater Entrance 1", "Amphitheater Entrance 2", 17.2),
  Edge("Amphitheater Entrance 1", "Lagoon Sidewalk - E", 38.0),
  Edge("Amphitheater Entrance 1", "Lagoon Sidewalk - S", 21.0),

  Edge("Amphitheater Entrance 2", "Amphitheater", 1.0),
  Edge("Amphitheater Entrance 2", "Amphitheater Entrance 1", 17.2),
  Edge("Amphitheater Entrance 2", "Food Stalls C", 9.5),
  Edge("Amphitheater Entrance 2", "Lagoon", 34.3),
```

Methodology

Backend

Graph file in Dart

```
Map<String, Node> nodes = {
  "Amphitheater": Node("Amphitheater", 14.597315774401595, 121.01042679308465),
  "Amphitheater Entrance 1": Node("Amphitheater Entrance 1", 14.597411959689742, 121.01036387507604),
  "Amphitheater Entrance 2": Node("Amphitheater Entrance 2", 14.597401156098861, 121.01052017046753),
  "Campus Development and Maintenance Building": Node("Campus Development and Maintenance Building", 14.596330048069238, 121.01118273009989),
  "Canteen - Pathwalk L": Node("Canteen - Pathwalk L", 14.596914993951678, 121.00995399834156),
  "Canteen - Pathwalk R": Node("Canteen - Pathwalk R", 14.597055440902018, 121.00991572191914),
  "Charlie Del Rosario Hall": Node("Charlie Del Rosario Hall", 14.597109560457717, 121.00961913582897),
  "Charlie Del Rosario Hall - Entrance": Node("Charlie Del Rosario Hall - Entrance", 14.597168107072093, 121.00987904034767),
```

Methodology

Backend

Dijkstra’s Algorithm in Dart

The dijkstra.dart file implements Dijkstra's algorithm to find the shortest path between nodes in a graph. It defines a Graph class to represent a graph using an adjacency list. The Graph constructor initializes the adjacency list with edges and their reverse to ensure source-to-destination and vice-versa connectivity. The dijkstra method calculates the shortest path from a start node to an end node using a priority queue to explore nodes based on their distances. It maintains maps to track distances and previous nodes for path reconstruction. The algorithm records the start time, processes nodes, updates distances, and reconstructs the shortest path upon reaching the end node, also printing the total weight and duration of the operation. If no path is found, it returns an empty list.

Methodology

Backend

Dijkstra's Algorithm in Dart

```dart
class Graph {
  final Map<String, List<Edge>> adjList = {};

  // constructor for the graph with the adjacency list
  Graph(List<Edge> edges) {
    for (var edge in edges) {
      // adds source and destination nodes sa adjacency list
      adjList.putIfAbsent(edge.source, () => []).add(edge); // adds the source node to the adjacency list
      adjList.putIfAbsent(edge.destination, () => []).add( // adds the destination node to the adjacency list
        Edge(edge.destination, edge.source, edge.weight), // adds the reverse edge
      );
    }
  }

```

Methodology

Backend

Dijkstra's Algorithm in Dart

```dart
List<String> dijkstra(String start, String end) {
  final distances = <String, double>{}; // for distance from a single node
  final previous = <String, String?>{}; // for tracking the paths
  final priorityQueue = PriorityQueue<MapEntry<String, double>>( // for the priority queue
        (a, b) => a.value.compareTo(b.value), // compares the values of the nodes
  );

  for (var node in adjList.keys) {
    distances[node] = double.infinity; // sets initial distances to infinite
    previous[node] = null; // sets initial previous node to null
  }
  distances[start] = 0; // sets the distance of the start node to 0

  // start time ni algo
  final startTime = DateTime.now(); // gets the current time
  print('Dijkstra algorithm started at: $startTime');

  priorityQueue.add(MapEntry(start, 0)); // adds the start node to the priority queue
```

Methodology

Backend

Dijkstra's Algorithm in Dart

```dart
while (priorityQueue.isNotEmpty) { // while may laman pa si priority queue
  final current = priorityQueue.removeFirst().key; // removes the first node from the priority queue

  if (current == end) { // if 'yung current node is the end node
    final path = <String>[]; // creates a list for the path
    double totalWeight = 0; // variable for total weight
    String? step = end; // sets the step to the end node
    while (step != null) { // while 'yung 'step is not null
      path.insert(0, step); // inserts the step to the path list
      if (previous[step] != null) { // if 'yung previous step is not null
        totalWeight += adjList[previous[step]]! // adds the weight of the edge to the total weight
            .firstWhere((edge) => edge.destination == step) // gets the edge with the destination of the step
            .weight; // gets the weight of the edge
      }
      step = previous[step]; // sets the step to the previous step
    }
```

Methodology

Backend

Dijkstra's Algorithm in Dart

```
    // end time ni algo
    final endTime = DateTime.now();
    final duration = endTime.microsecondsSinceEpoch - startTime.microsecondsSinceEpoch;
    final durationInMilliseconds = duration / 1000; // with decimal places para mas accurate

    print('Dijkstra algorithm ended at: $endTime');
    print('Dijkstra algorithm response time: ${durationInMilliseconds.toStringAsFixed(3)} ms');
    print('Total weight of the shortest path: $totalWeight');

    return path; // returns the path
}
```

Methodology

Backend

Dijkstra's Algorithm in Dart

```dart
    if (distances[current]! == double.infinity) { // if 'yung distance of the current node is infinite
      break; // Break the loop
    }

    for (var edge in adjList[current]!) { // for each edge in the adjacency list of the current node
      final alt = distances[current]! + edge.weight; // calculates the distance
      if (alt < distances[edge.destination]!) { // if 'yung distance is less than the distance of the destination node
        distances[edge.destination] = alt; // sets the distance of the destination node to the calculated distance
        previous[edge.destination] = current; // sets  the previous node of the destination node to the current node
        priorityQueue.add(MapEntry(edge.destination, alt)); // adds the destination node to the priority queue
      }
    }
  }
```

Methodology

Backend

Dijkstra's Algorithm in Dart

```dart
    // end time in case walang path
    final endTime = DateTime.now();
    final duration = endTime.microsecondsSinceEpoch - startTime.microsecondsSinceEpoch;
    final durationInMilliseconds = duration / 1000;

    print('Dijkstra algorithm ended at: $endTime');
    print('Dijkstra algorithm response time: ${durationInMilliseconds.toStringAsFixed(3)} ms (no path found)');

    return []; // returns an empty list if no path is found
  }
}
```

Results and Discussion

Response Time and Accuracy Rate

TestID	Origin	Destination	Response Time (ms)			
			Trial 1	Trial 2	Trial 3	Average
1	Pylon	PUP Sta Mesa Ferry Station	1.719	0.926	0.513	1.0527
2	Lagoon	Ninoy Aquino Library and Learning Resources Center	0.692	0.638	0.751	0.6937
3	Track and Field Oval	West Wing	0.674	0.493	0.383	0.5167
4	Grandstand	Gymnasium and Sports Center	0.975	2.348	0.473	1.2653
5	Pylon	East Wing	0.896	0.569	0.481	0.6487
6	Charlie Del Rosario Building	Nutrition and Food Science Building	1.371	0.487	0.522	0.7933
7	Pylon	Visitors Information Center	0.798	0.307	0.258	0.4543
8	Engineering Research Science Building	Property and Supply Building	1.229	0.685	0.675	0.8630
9	Interfaith Chapel	Lagoon	0.572	0.533	0.503	0.5360
10	Tahanan ng Atleta	Guard House	1.15	0.284	0.355	0.5963
						0.7420

TestID	Origin	Destination	Accuracy Rate (%)		
			Manual Calculation Total Weight (m)	Algorithm Computation Total Weight (m)	
1	Pylon	PUP Sta Mesa Ferry Station	411.4	394.8	104.2047
2	Lagoon	Ninoy Aquino Library and Learning Resources Center	158.5	142.5	111.2281
3	Track and Field Oval	West Wing	244.7	244.7	100.0000
4	Grandstand	Gymnasium and Sports Center	300.9	296.2	101.5868
5	Pylon	East Wing	362.4	361.6	100.2212
6	Charlie Del Rosario Building	Nutrition and Food Science Building	204.2	204.2	100.0000
7	Pylon	Visitors Information Center	32.4	32.4	100.0000
8	Engineering Research Science Building	Property and Supply Building	531.4	465.3	114.2059
9	Interfaith Chapel	Lagoon	183.9	166.5	110.4505
10	Tahanan ng Atleta	Guard House	162.6	161.4	100.7435
					104.2641

Discussion

The results demonstrate how well Dijkstra's algorithm works in the campus navigation system. The algorithm's practical application in real-world scenarios is shown by its ability to consistently locate the shortest pathways between facilities. The measured time complexity aligns with the theory, confirming the effectiveness of the method for the given graph size. Additionally, the successful validation of the calculated paths against manual measurements indicates that PUPathfinder is reliable in providing accurate navigation routes. These results support the decision to implement Dijkstra's algorithm in the mobile application, ensuring users can quickly and efficiently navigate the campus. In order to further enhance the system's usefulness and user experience, future improvements may include incorporating real-time data to deal with brief route changes, such as construction or restrictions.

Conclusion

Learning Outcomes

- Discuss relevant studies revolving around definition of Dijkstra's algorithm, existing campus navigation systems, and Flutter as a mobile development framework.
- Implement Dijkstra's algorithm as a single-source shortest path graph traversal through a campus navigation app.
- Demonstrate knowledge in data structures, particularly graphs, queues, and lists, as tools for implementation of Dijkstra's algorithm.

Conclusion

Learning Outcomes

- Develop a mobile application with Flutter framework and in a collaborative workspace environment such as Github.

- Analyze the performance of Dijkstra's algorithm in a campus navigation system with 100 and more nodes and vertices.

Findings

The PUPathfinder application was able to indetify the shortest paths to any destination facilities from any point of origin on campus, and has demonstrated efficiency, reliability, and accuracy in terms of the suggest routes.

Efficient

The overall average response time of the response time tests is **0.742 milliseconds**.

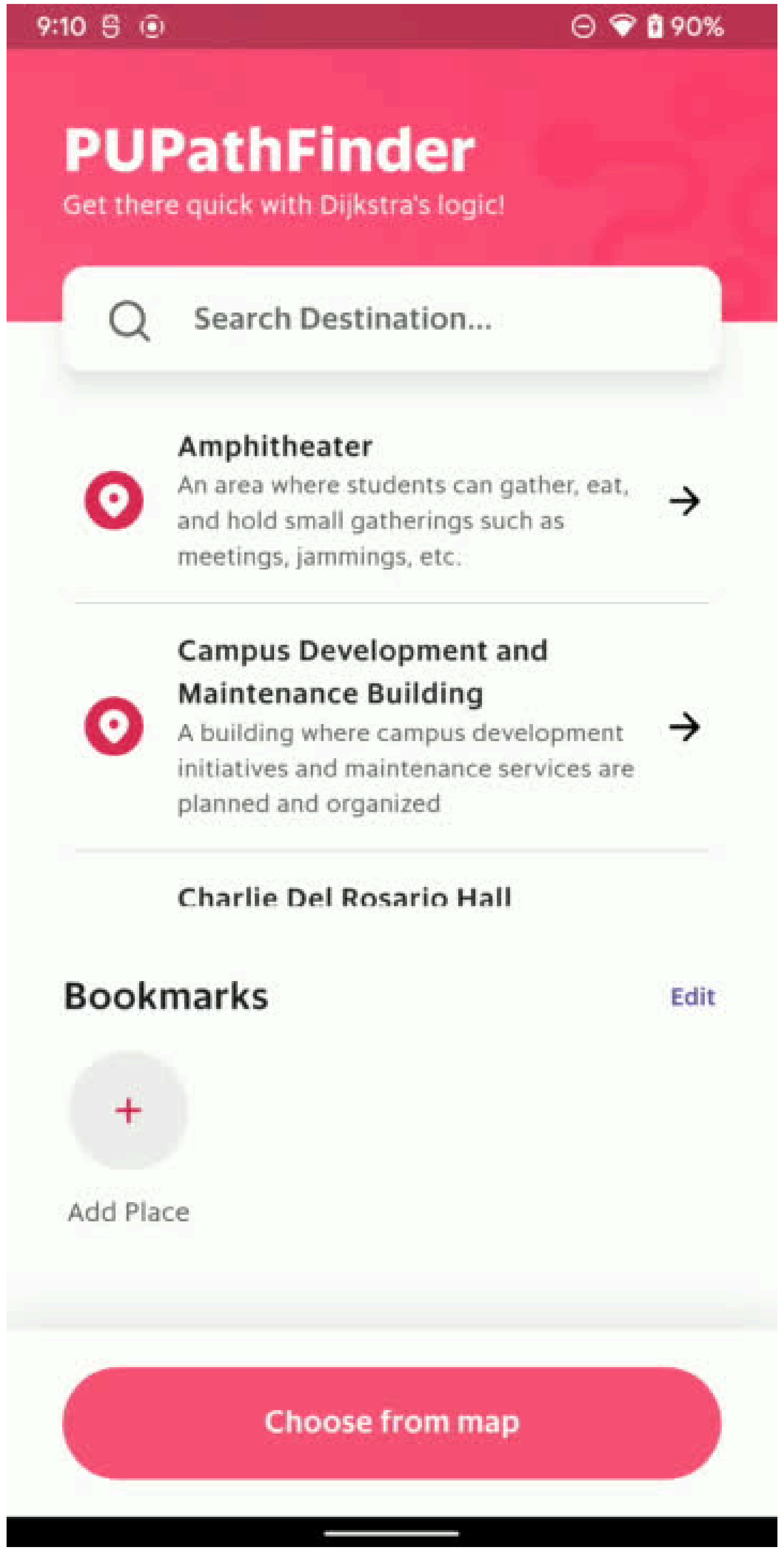
Reliable

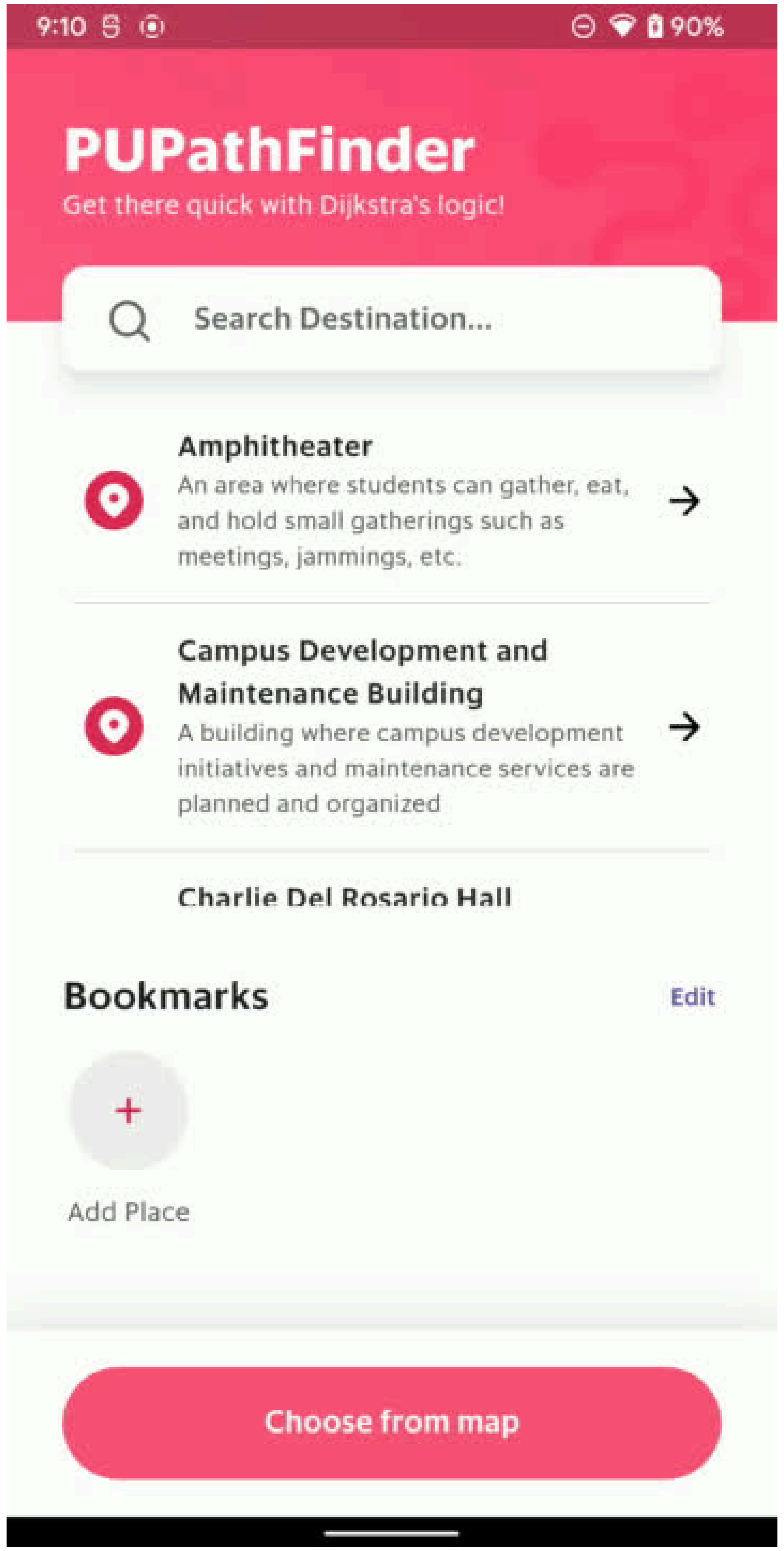
Peak of average response time were only at **1.2653 milliseconds**

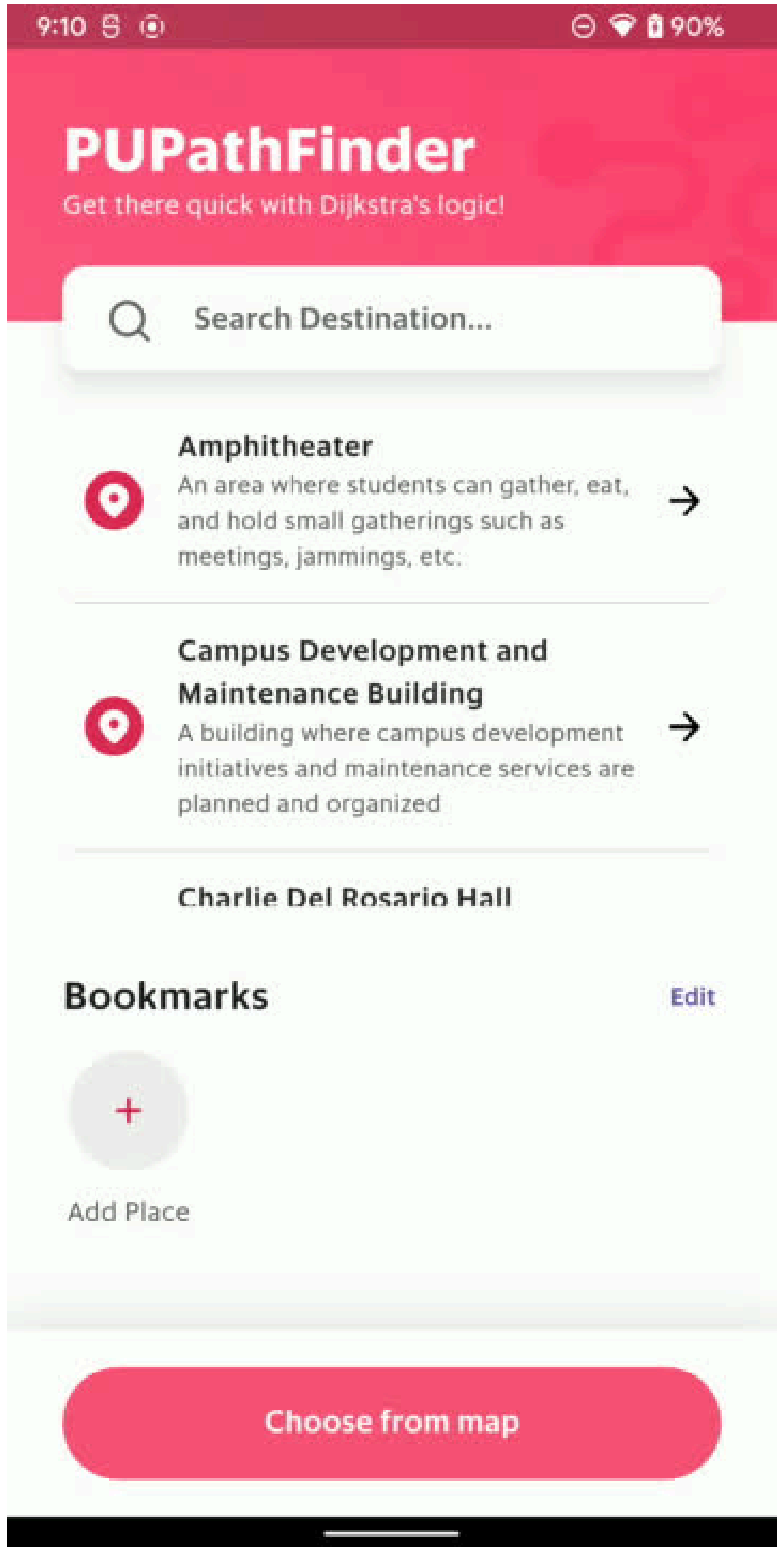
Accurate

The overall average accuracy rate across the 10 scenarios resulted in **104.2641%** or **1.04.**

PUPATHFINDER







Conclusion

Recommendations

- Regular application updates to reflect any changes in campus infrastructure, such as new buildings or construction zones, to ensure the accuracy and reliability of the navigation system.

- Integration of additional features like real-time location tracking and alternative route suggestions could further optimize travel time within the campus.

Conclusion

Recommendations

- Expansion of the coverage of the navigation system to include interior routes within major buildings could significantly help users locate their desired location, particularly in large and complex structures.

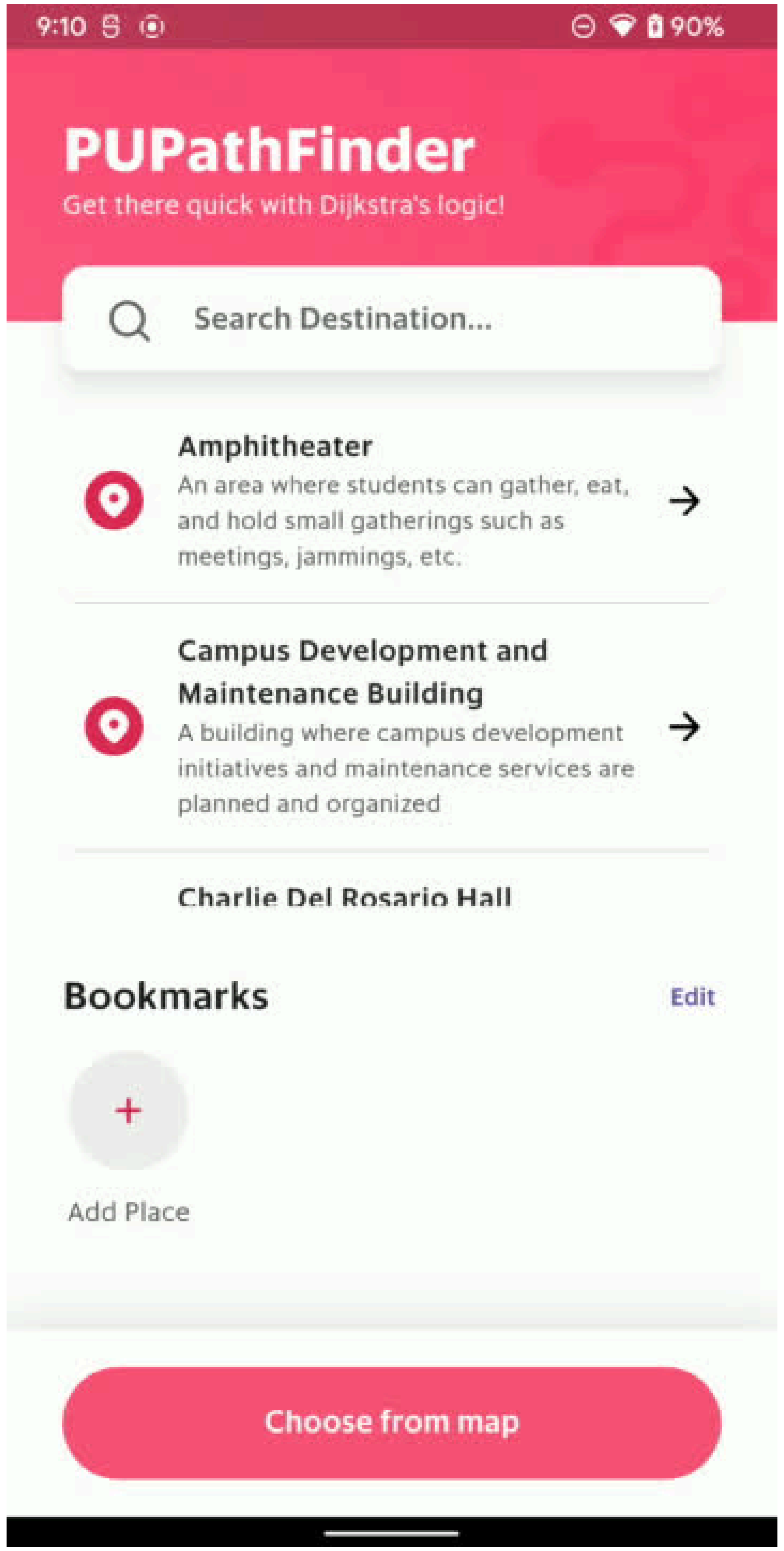
- Collaboration with campus administration to promote the use of the app and incorporate user feedback will be essential for continuous improvement of the navigation system to meet the needs of its users better.

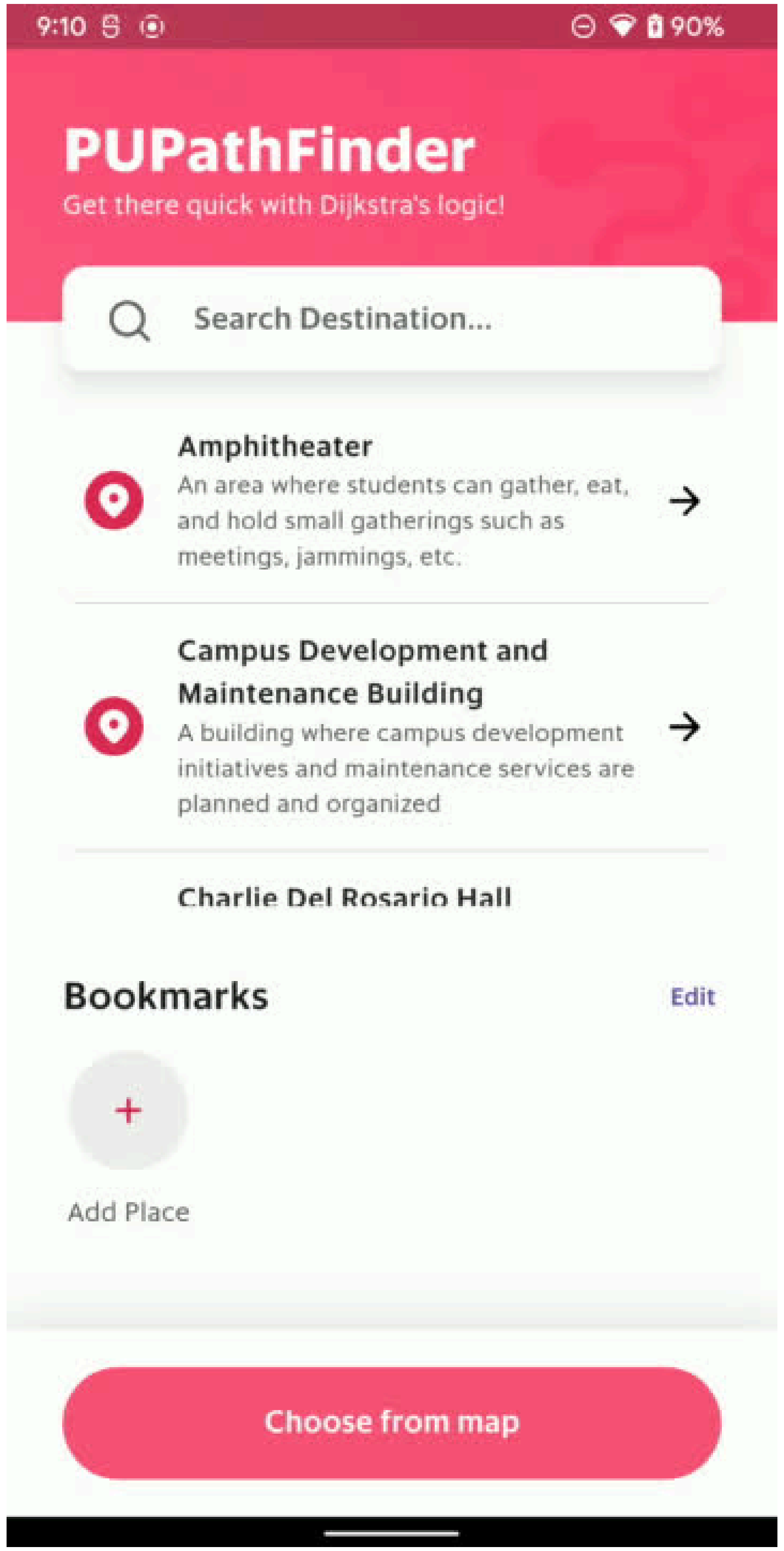
Conclusion

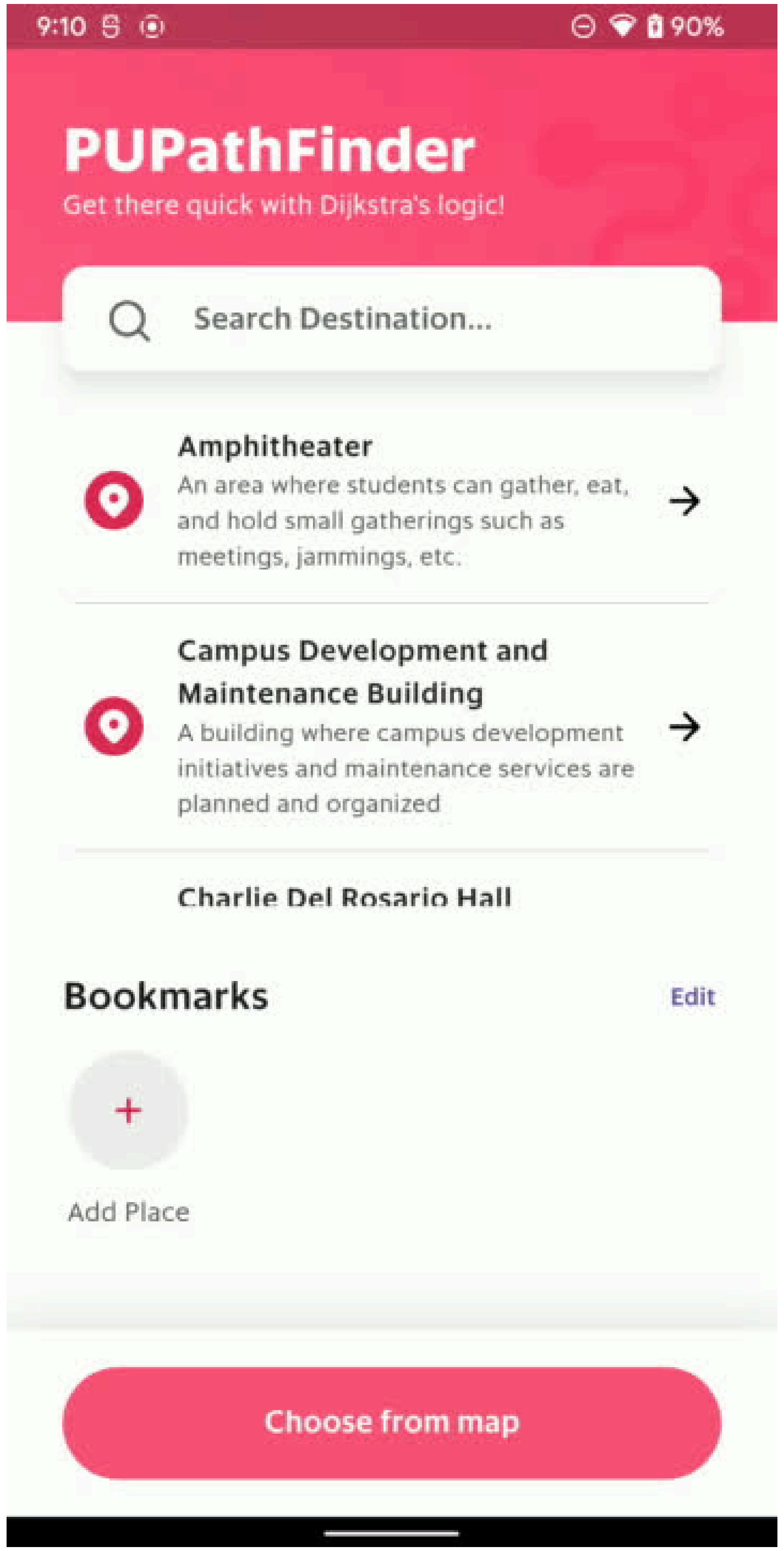
Open Problems

- It lacks a real-time location tracker.
- It excludes the rooms within the facilities to minimize the project's complexity and execution.
- The method used in collecting the data for the edges is not exactly accurate enough as they are manually collected and calculated using Google Maps.
- Users must put the exact name of the facility for both origin and destination. The names must match the set name in the program.

PUPATHFINDER







PUPATHFINDER

Meet the Team

Syruz Ken C. Domingo

Julia Kyla Rustia

Fervicmar D. Lagman

Henry James R. Carlos

Thank You

A Campus Navigation System for Shortest Routes between Facilities in PUP Mabini Campus using Dijkstra's Algorithm

Design and Analysis of Algorithms

Members:

Syruz Ken Domingo
Henry James Carlos
Fervicmar Lagman
Julia Kyla Rustia