Amusement Park

By Jabril Adan

The problem-Task A

Amusement parks are known for their thrilling rides and beautiful attractions. Everyone from children to senior citizens enjoy them but there is a common problem as well. The long waits, overcrowding, and not getting enough fun! I have decided to use my project to try and solve this problem. I will be implementing A* search to search through the possible paths within a template amusement park. For now the output will be a simple path between nodes but the projects end goal is a full app that uses gps to find your exact location at all times and current wait times at all the rides and attractions to maximize efficiency.

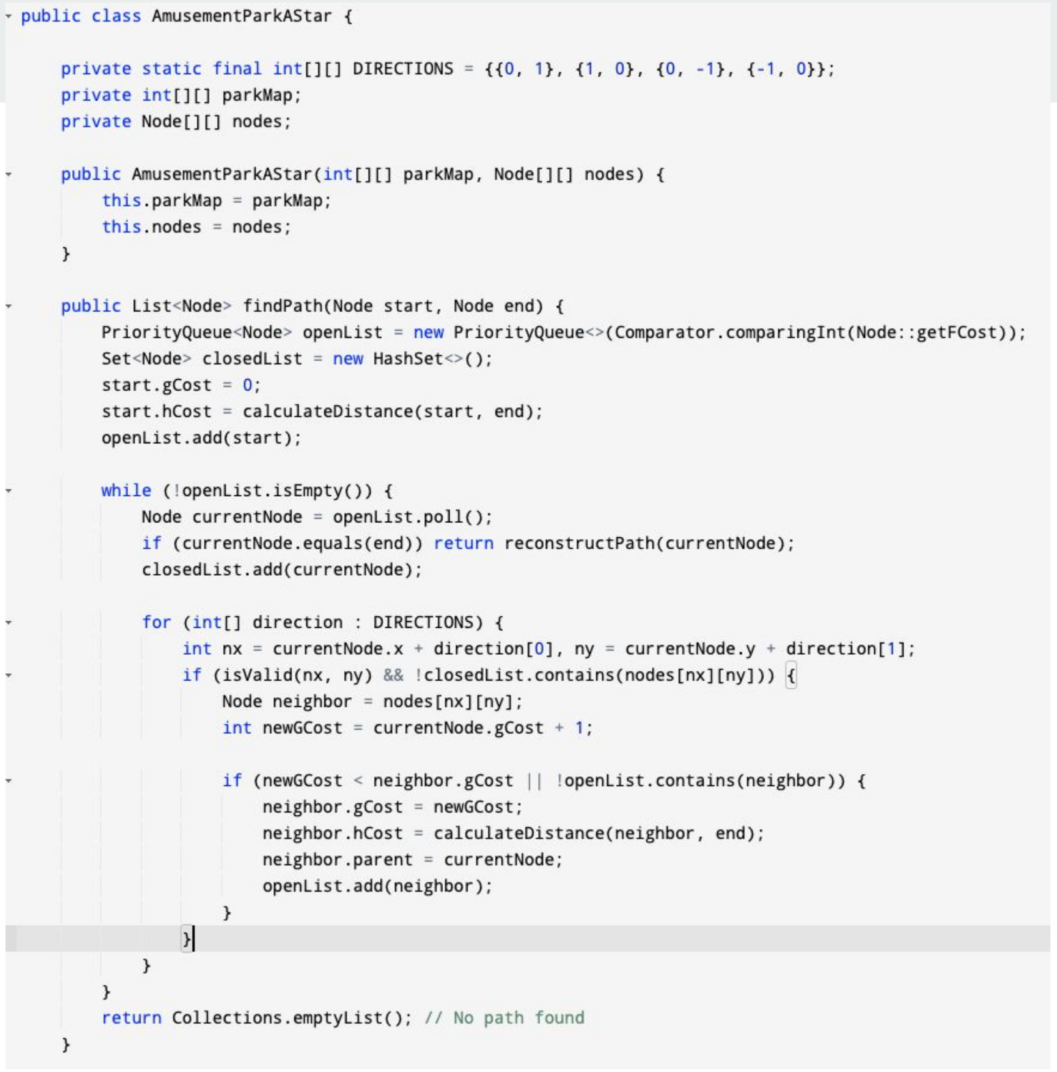
A* Search

I chose A* search because it combines the strengths of both dijkstra's optimality and greedy's efficiency. Guided by a heuristic A* search is used to find the shortest path in large and complex graphs, or in this case maps. Since our problem is searching through an amusement park A* search is the best option as the use of a Heuristic function to guide the user through the most optimal route of fun is the main goal.

- **Pros:**
  - Guaranteed optimal path
  - Efficient
  - Handles complex environments
    - As the environment changes so can the search by changing the cost and heuristic function as needed.
- **Cons:**
  - High memory consumption
  - Assumes static environment
    - This is especially hard because as the user moves around and wait times increase or decrease the path will eventually become nonoptimal.

MY A* SEARCH

Worst case time complexity: O(b^d)
Space Complexity:O(|V|)



```java
public class AmusementParkAStar {

    private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    private int[][] parkMap;
    private Node[][] nodes;

    public AmusementParkAStar(int[][] parkMap, Node[][] nodes) {
        this.parkMap = parkMap;
        this.nodes = nodes;
    }

    public List<Node> findPath(Node start, Node end) {
        PriorityQueue<Node> openList = new PriorityQueue<>(Comparator.comparingInt(Node::getFCost));
        Set<Node> closedList = new HashSet<>();
        start.gCost = 0;
        start.hCost = calculateDistance(start, end);
        openList.add(start);

        while (!openList.isEmpty()) {
            Node currentNode = openList.poll();
            if (currentNode.equals(end)) return reconstructPath(currentNode);
            closedList.add(currentNode);

            for (int[] direction : DIRECTIONS) {
                int nx = currentNode.x + direction[0], ny = currentNode.y + direction[1];
                if (isValid(nx, ny) && !closedList.contains(nodes[nx][ny])) {
                    Node neighbor = nodes[nx][ny];
                    int newGCost = currentNode.gCost + 1;

                    if (newGCost < neighbor.gCost || !openList.contains(neighbor)) {
                        neighbor.gCost = newGCost;
                        neighbor.hCost = calculateDistance(neighbor, end);
                        neighbor.parent = currentNode;
                        openList.add(neighbor);
                    }
                }
            }
        }
        return Collections.emptyList(); // No path found
    }
```

Explanation

- First the Amusementpark class is initialized with a park map and a nodes array which represent different rides and attractions as well as paths and walls. Each node has a cost associated with it along with coordinates within the map.
- The findpath method uses the openlist as a priority queue to store nodes to be evaluated, in order of their cost. The closedlist are nodes that have already been evaluated.
- The main loop continues as long as their are nodes within the open list, if the end node is pulled from the list then the path to get there is returned. If not the node is added to the closed list
- For each direction that the path takes the program also calculates the cost to reach that node. If the cost is lower or if that neighbor is not in the openlist, it is returned to the open list.
- When the end node is reached the reconstructPath method is then called upon to reconstruct the path that was used to reach the end.
- If no path is found then an empty list is returned which signifies that there is no path.

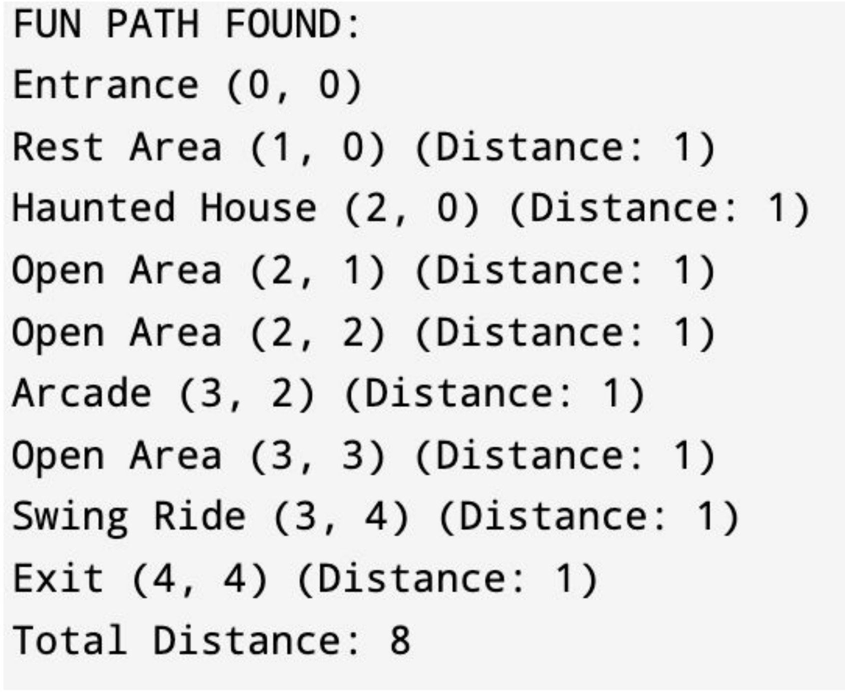
Entrance	Wall	Gift Shop	Food	Ferris Wheel
Rest Area	Wall	Carousel	Wall	Roller Coaster
Haunted House	Open Area	Open Area	Wall	Bumper Cars
Restroom	Wall	Arcade	Open Area	Swing Ride
Water Park	Open Area	Open Area	Wall	Exit

Open queue: Open path Ride Exit

Closed queue: Entrance Food

Output&Future

This sample output shows a small snippet of what is possible using a* search. On a 5x5 array with each node being either an attraction, path, or wall. We find that the program reaches the end while hitting as many rides as possible. Ideally the output should return a map for the user to follow to each point. In the future the program will be using gps to use the location of the individual while incorporating real time changes within the park to optimize fun.



```
FUN PATH FOUND:
Entrance (0, 0)
Rest Area (1, 0) (Distance: 1)
Haunted House (2, 0) (Distance: 1)
Open Area (2, 1) (Distance: 1)
Open Area (2, 2) (Distance: 1)
Arcade (3, 2) (Distance: 1)
Open Area (3, 3) (Distance: 1)
Swing Ride (3, 4) (Distance: 1)
Exit (4, 4) (Distance: 1)
Total Distance: 8
```