Kyle Torres
CS 460-01

Final Project

Problem Statement:

Going to a live sporting event is always a great environment watching your favorite team play in person. Let’s say you get hungry and don’t want to miss a major event during the game. Although there is no possible way to predict when the next big thing happens in a game, you can try to optimize the path you take when getting food.

Core Algorithm:

This algorithm finds a path from seat to concession taking into account the combined wait and walk time and returns the best concession to go to. In other words it returns the concession that takes less time away from your seat. This algorithm uses a combination of Queue to determine the wait time at a concession stand and Dijstra’s to find the best path.

```cpp
RecommendedConcession::RecommendedConcession(const StadiumGraph& graph) : graph(graph) {}

std::string RecommendedConcession::recommendBest(const std::string& seat, int& totalTime) const {
    std::string bestConcession = "None";
    totalTime = std::numeric_limits<int>::max();

    int bestWalkTime = std::numeric_limits<int>::max();
    int bestWaitTime = std::numeric_limits<int>::max();

    // For specified seat and each concession calculate total wait time (walk time + wait time)
    for (const std::string& concession : graph.getAllConcessions()) {

        // Get walk total walk time from specified seat to current concession
        int walk = graph.dijkstra(seat, concession);


        // In the case a concession is not reachable
        if(walk == INT_MAX) continue;

        // Get wait time for current concession
        int wait = graph.getQueueTime(concession);

        // Calculate total time from specified seat to current concession
        int combined = walk + wait;

        // Output total time for specified seat and current concession
        std::cout << "\n";
        std::cout << seat << " → " << concession
                  << " | Walk: " << walk << ", Wait: " << wait
                  << " → Total: " << combined << "\n";

        // Determine best route from specified seat to a concession
        if (combined < totalTime || (combined == totalTime && walk < bestWalkTime)||
            (combined == totalTime && walk == bestWalkTime && wait < bestWaitTime)) {
            totalTime = combined;
            bestWalkTime = walk;
            bestWaitTime = wait;
            bestConcession = concession;
        }
    }

    // Return best concession to got to from specified seat
    return bestConcession;
}
```

```cpp
// Implementation of Dijkstra's algorithm to find the quickest path from a seat to a concession in a stadium
int StadiumGraph::dijkstra(const std::string& start, const std::string& end) const {

    // Map to store shortest path to each node
    std::unordered_map<std::string, int> dist;

    // Minimum priority queue to select the next closest node
    std::priority_queue<
        std::pair<int, std::string>,
        std::vector<std::pair<int, std::string>>,
        std::greater<>> priQueue;

    // Initialize all distances
    for (const auto& node : adjList) {
        dist[node.first] = std::numeric_limits<int>::max();
    }

    // Initialize start node to a distance of 0
    dist[start] = 0;
    priQueue.emplace(0, start);

    // Examine the stadium graph
    while (!priQueue.empty()) {
    std::pair<int, std::string> currentItem = priQueue.top();
    priQueue.pop();

    int cost = currentItem.first;
    std::string node = currentItem.second;

    if (node == end) return cost;

    for (const auto& neighborPair : adjList.at(node)) {
        const std::string& neighbor = neighborPair.first;
        int weight = neighborPair.second;

        int newCost = cost + weight;
```

```cpp
        // Check if new path is shorter
        if (newCost < dist[neighbor]) {

            // New path is shorter, update and push onto queue
            dist[neighbor] = newCost;
            priQueue.emplace(newCost, neighbor);
        }
    }
}

    return std::numeric_limits<int>::max(); // No path found
};
```

Test Plan and Results:

```
Test Case 1: Pick Shortest Wait Time

Seat1 → Concession1 | Walk: 5, Wait: 3 → Total: 8

Seat1 → Concession2 | Walk: 8, Wait: 2 → Total: 10

Seat1 → Concession3 | Walk: 5, Wait: 1 → Total: 6
Expected: Concession3 , Got: Concession3 (Total time: 6 mins)
Test Passed

Test Case 2: Disconnected concession

Seat2 → Concession1 | Walk: 7, Wait: 3 → Total: 10
Expected: Concession1 , Got: Concession1 (Total time: 10 mins)
Test Passed

Test Case 3: If tied total time return shortest distance time

Seat3 → Concession1 | Walk: 8, Wait: 2 → Total: 10

Seat3 → Concession2 | Walk: 6, Wait: 4 → Total: 10

Seat3 → Concession3 | Walk: 7, Wait: 3 → Total: 10
Expected: Concession2 , Got: Concession2 (Total time: 10 mins)
Test Passed
```

Discussion of Trade-offs, limitations, future work:

There was a choice to use Dijkstra's or A* and I decided to choose Dijkstra's. I would have liked to explore using A* but due to time constraints I was unable to further research. The limitation to the current state of the algorithm is that wait times are not being dynamically updated. And something to do for future work will be implementing a way to add randomness to the queue times and allow wait times to be dynamically updated.

GitHub repo link:

https://github.com/Kyray-00/CS460-01-Final-Project