**Technical Design: Personalized Layover Planner**

**1. Introduction**

This document outlines the technical design of the Personalized Layover Planning Tool. The system's primary goal is to generate an optimal and time-feasible itinerary for a passenger during a layover. It takes the total layover duration, arrival/departure gates, and passenger interests as input, and produces a step-by-step route of locations to visit within the airport terminal.

The design prioritizes **speed** for user-facing requests by using a **pre-computation** strategy for pathfinding and a **fast, heuristic-based algorithm** for itinerary generation.

**2. System Architecture & Flow**

The system operates in two main phases: a one-time **Pre-computation Phase** and a real-time **Request Phase**.

* **Pre-computation Phase**: The airport terminal map is static. Therefore, we can calculate the shortest travel time between all pairs of locations (nodes) beforehand. The result of this phase is a complete shortest-paths matrix. This is computationally intensive but only needs to be run once or whenever the airport layout changes.
* **Request Phase**: When a user requests a plan, the system uses the pre-computed path matrix to quickly generate a custom itinerary. This phase is optimized for speed.

**2.1. Data Flowchart**

Here is a flowchart illustrating the process during the Request Phase:

Code snippet

graph TD

A[User Input: Layover Time, Gates, Interests] --> B{Find Matching Locations};

B --> C{Filter out non-matching POIs using Hash Set Intersection};

C --> D{Get List of Interesting Location IDs};

D --> E{Build Itinerary using Greedy Algorithm};

subgraph "Greedy Itinerary Builder"

E --> F[Start at Arrival Gate];

F --> G{Find closest unvisited POI};

G --> H{Time Check: Can we visit it and still reach the departure gate?};

H -- Yes --> I[Add POI to Itinerary & Update Time/Location];

I --> G;

H -- No --> J[Finalize Path to Departure Gate];

end

J --> K[Format & Display Final Itinerary];

subgraph "Pre-computation (Done Once)"

X[Airport Map Data] --> Y{Run Floyd-Warshall};

Y --> Z[All-Pairs Shortest Path Matrix];

end

Z --> E;

**3. Data Structures**

The choice of data structures is critical for the system's performance.

**3.1. Airport Graph Representation**

The airport is represented by two primary structures:

1. **Nodes Dictionary**: nodes = {node\_id: {details}}
   * A Python dictionary (hash map) maps integer IDs to location details (name, type, tags).
   * **Trade-offs**: A hash map provides O(1) average-case time complexity for accessing any node's data by its ID. This is superior to using an array, where we might need to search for a specific node if IDs are not contiguous. The memory overhead is minimal for the expected number of locations in an airport.
2. **Adjacency Matrix**: dist\_matrix = [[...]]
   * An NtimesN matrix (list of lists in Python) where matrix[i][j] stores the direct travel time between node i and node j.
   * **Trade-offs & Justification**:
     + **Adjacency Matrix vs. Adjacency List**: For this problem, an **adjacency matrix** was chosen.
     + **Reasoning**: The Floyd-Warshall algorithm works naturally with an adjacency matrix. More importantly, airport terminals can be considered relatively **dense graphs** (many locations are connected to many others, especially through central hubs). While an adjacency list is more space-efficient for sparse graphs (O(N+E) vs. O(N2)), the O(N2) space complexity of a matrix is acceptable for a few hundred nodes. The key benefit is the **O(1) time complexity to check the weight of an edge** between any two nodes, which is used repeatedly in our pathfinding and itinerary-building algorithms. An adjacency list would require O(textdegree) to find an edge.

**3.2. User Interests and Location Tags**

* **Structure**: Python set() is used for both the tags of each location and the user\_interests.
* **Trade-offs & Justification**: Using sets is a deliberate choice for performance. The core operation is matching user interests to location tags. The set.intersection() operation is highly optimized, with an average time complexity of O(min(∣S\_1∣,∣S\_2∣)), where S\_1 and S\_2 are the two sets. This is significantly faster than iterating through two lists (O(∣L\_1∣cdot∣L\_2∣)). This makes the filtering step (find\_interesting\_locations) very efficient.

**4. Algorithms**

**4.1. floyd\_warshall**

This function pre-computes the shortest path between all pairs of nodes in the graph.

* **Pseudocode**:
* function FloydWarshall(graph):
* let dist be a copy of the graph's adjacency matrix
* let n = number of nodes in graph
* for k from 0 to n-1:
* for i from 0 to n-1:
* for j from 0 to n-1:
* if dist[i][k] + dist[k][j] < dist[i][j]:
* dist[i][j] = dist[i][k] + dist[k][j]
* return dist
* **Complexity**:
  + **Time**: O(N3), where N is the number of nodes. The three nested loops dictate this complexity.
  + **Space**: O(N2) to store the resulting shortest-paths matrix.
* **Justification**:
  + **vs. Running Dijkstra from every node**: An alternative is to run Dijkstra's algorithm from each of the N nodes. With a binary heap, Dijkstra is O(E+NlogN), making the total time O(Ncdot(E+NlogN)). For a dense graph where E approaches O(N2), this becomes O(N3), similar to Floyd-Warshall. Given that Floyd-Warshall has a much simpler implementation and performs well for the expected scale of an airport graph (N < 500), it is the more practical choice for this pre-computation task.

**4.2. find\_interesting\_locations**

This function filters locations based on user preferences.

* **Complexity**:
  + **Time**: O(NcdotL), where N is the total number of nodes and L is the average size of the tag sets. This is because we must iterate through all nodes, and for each, the set intersection takes time proportional to the set sizes.
  + **Space**: O(K), where K is the number of interesting locations found. This is for storing the resulting list of IDs.
* **Justification**: This is a straightforward filtering task. A linear scan through all nodes is unavoidable as any node could be a potential match. The use of hash sets for the matching operation itself is the most performant method available.

**4.3. build\_greedy\_itinerary**

This function constructs the final itinerary using a greedy "nearest neighbor" heuristic.

* **Pseudocode**:
* function BuildItinerary(start, end, totalTime, pois, paths):
* itinerary = [start]
* availableTime = totalTime - safetyBuffer
* currentLoc = start
* unvisitedPois = set(pois)
* while availableTime > 0 and unvisitedPois is not empty:
* find bestNextPoi in unvisitedPois with minimum travel\_time from currentLoc
* if no bestNextPoi found:
* break
* timeToReturn = paths[bestNextPoi][end]
* requiredTime = travel\_time(currentLoc, bestNextPoi) + visitDuration + timeToReturn
* if requiredTime <= availableTime:
* add bestNextPoi to itinerary
* availableTime -= (travel\_time + visitDuration)
* currentLoc = bestNextPoi
* remove bestNextPoi from unvisitedPois
* else:
* break // No more time for any other stops
* add end to itinerary
* return itinerary
* **Complexity**:
  + **Time**: O(K2), where K is the number of *interesting* locations. The while loop runs at most K times. Inside the loop, we iterate through the remaining unvisited points of interest (at most K) to find the nearest one.
  + **Space**: O(K) to store the unvisited\_poi\_ids set and the final itinerary list.
* **Justification**:
  + **vs. Optimal Solution (Traveling Salesperson Problem - TSP)**: Finding the absolute shortest path that visits a set of locations is an NP-hard problem. An optimal solver (e.g., using dynamic programming) would have a time complexity of roughly O(K2cdot2K), which is computationally infeasible for anything more than a handful of locations.
  + The **greedy heuristic** provides a "good enough" solution very quickly. For a passenger, a fast, practical plan is far more valuable than a mathematically perfect plan that takes too long to compute. The nearest-neighbor approach is intuitive, simple to implement, and produces sensible routes for this use case.

**5. Benchmarks & Performance**

Performance is split between pre-computation and real-time requests.

| Operation | Complexity | N=50 Nodes | N=200 Nodes | Notes |
| --- | --- | --- | --- | --- |
| **Pre-computation** |  |  |  |  |
| floyd\_warshall | O(N3) | ~125k ops | ~8M ops | Intensive, but performed only once. A 200-node airport is very large. |
| **Real-time Request** |  |  |  |  |
| find\_interesting\_locations | O(N) | ~50 ops | ~200 ops | Extremely fast. Assumes small tag sets. |
| build\_greedy\_itinerary | O(K2) (K <= N) | ~225 ops | ~225 ops | Very fast. Assumes K=15 matching POIs. Depends on K, not N. |

**Conclusion**: The system is designed to be highly responsive to user requests. The expensive O(N3) calculation is done offline. The real-time performance depends on K (the number of places the user is interested in), not N (the total number of places in the airport), making the user experience very fast.