CS 214 Final Project Design Doc 3

Data structures

- Position-to-node ID dictionary
    - **Role**: Maps every unique position used to initialize `TripPlanner` (ie, every segment endpoint and POI location) to a node in the graph.
    - **Data structure**: Hash table with separate chaining
    - **Justification for data structure**: I picked hash tables over association lists because although both were available in the compiled code, association lists have linear worst- and average-case time complexity for i) lookup, ii) insertion, iii) deletion and iv) member-checking. On the other hand, hash tables have constant average-case amortized time complexity for all four functions. A well-designed hash function (which was available in an importable library) can prevent collisions, making amortized Θ(1) complexity realistic. Hash tables also beat sorted and unsorted arrays in average-case complexity for all four functions. Arrays have between Θ(log n) and Θ(n) time complexity for the four functions.

      I chose separate chaining over open addressing because of availability as a reliable implementation in the compiled code. However, both implementations have amortized Θ(1) time complexity for the four functions.
- Node ID-to-position dictionary
    - **Role**: Maps every node in the graph to a unique position.
    - **Data structure**: Hash table with separate chaining
    - **Justification for data structure**: Same as above.
- Name-to-POI dictionary
    - **Role**: Maps every POI's name to its POI `struct`.
    - **Data structure**: Hash table with separate chaining
    - **Justification for data structure**: Same as above.
- Category-to-POI dictionary
    - **Role**: Maps every unique POI category to a linked list containing the POI `struct`s in that category.
    - **Data structure**: Hash table with separate chaining
    - **Justification for data structure**: The number of categories in test cases is typically small, so a dictionary implemented with an association list wouldn't do too poorly. In fact, since it is a simpler data structure, the initialization time would be lower. However, from stress testing I found that initializing the `TripPlanner` with even 100 segments and POIs takes just ~600 ms, with the bulk of the runtime taken by `find_nearby` (~3300 ms).[1] So, it didn't seem worth shaving

[1] Stress tests in Appendix

the initialization time by a bit to potentially massively increase the pathfinding time for a use case with a lot of categories.

- Node ID-to-node ID graph
    - **Role**: Maps edges (with weights representing Euclidean distances) between different nodes
    - **Data structure**: Weighted undirected adjacency matrix
    - **Justification for data structure**:
        - I used a weighted graph to minimize the number of times the distance between two positions was calculated. If I used an unweighted graph, I would have to extract the positions represented by two nodes and recalculate the Euclidean distance between them. Since getting the distance between two positions is a very frequent operation in Dijkstra's, it was important to make it time-cheap.

          I paid for this in higher space requirements. However, although the *space* required by the program increased (storing a `num` per edge instead of a boolean value), the *space complexity* remained the same $O(n^2)$ because the space per edge increased by a constant factor. This tradeoff seems worth it.
        - Since roads are two-way in this problem, they are naturally represented as an undirected graph. If I used a directed graph, I would have to remember to set edges bidirectionally. This is easy to forget during coding and can lead to hard-to-trace bugs. It also makes code harder to maintain if other programmers use my codebase, since *they* have to notice and remember to use the unnatural directed representation as well.
        - I used an adjacency matrix (AM) instead of an adjacency list (AL) because:
            - AM has a lower `get_edge` complexity (O(1) against AL's O(e)). `get_edge` is the single most frequent graph operation in Dijkstra's, so a low time cost is crucial.
            - AM also has O(1) `set_edge` against AL's O(e), which is helpful during graph initialization.
            - A reliable AM implementation was available in the compiled code.
        - However, using an AM has downsides:
            - `get_adjacent`, the only graph operation used in Dijkstra's besides `get_edge`, is O(v) rather than AL's O(e). This is a problem if the graph is sparse.
            - AM has a higher space complexity ($O(v^2)$) than AL (O(v+e)) for sparse graphs.
        - To address these downsides:
            - `get_edge` is a more frequent operation than `get_adjacent`, so I prioritized it.

- Although test cases are sparse graphs for quickly testing `TripPlanner` methods, realistically graphs used by map apps are on the denser side; AL and AM have the same space complexity for dense graphs, since AL's $O(v+e) = O(v+v^2) = O(v^2)$. So, using an AM wouldn't take up much more space. Further, the time complexity of `get_adjacent` wouldn't be a problem.

Algorithms

- Dijkstra's algorithm
    - **Role**: Finding the shortest path from `src_node` to `dst_node` for `plan_route` and generating `dist`, a table of the distance of every node from the source node. `dist` is used by `find_nearby`.
    - **Justification**: According to lecture notes 13, Dijkstra's has better time complexity (O(e log e) bounded by $O(v^2 \log v)$) than Bellman-Ford (O(e * v) bounded by $O(v^3)$) since it relaxes nodes in a clever order. I collected empirical evidence which supports this: I implemented both algorithms and found that Bellman-Ford failed the 45-segment-and-POI stress test, while Dijkstra's came way under the 10 second-time limit. Bellman-Ford is useful when weights in a graph are negative, but since Euclidean distance is always nonnegative, Dijkstra's can be used.

      Simple implementations of depth-first-search and breadth-first-search also don't always find the shortest path, so they cannot be used either. They can be modified to compute all possible paths to the destination node, then return the lowest-cost path, but that would be far from Dijkstra's optimality.

      Minimum spanning tree algorithms would minimize the total weight on the graph while still keeping all nodes connected, but will not necessarily find the shortest path between two given nodes.
- Heap sort via priority queue
    - **Role**: Maintaining a heap `todo` of nodes not assessed during Dijkstra's
    - **Justification**: For greedy search, the closest node must be removed from `todo`. So, a stack wouldn't work as it would only remove the most recently added node, while a queue would only remove the oldest nodes.

      Heap sort is better than maintaining a sorted array, since removing the minimum is the same for both (O(1)) but insertion is O(log n) for a priority queue while a sorted array has O(n) and Θ(n) insertion due to the need to shift elements if a new element is added between the array. A sorted linked list has the same complexity: O(1) for removing the minimum, but O(n) for insertion since the entire list must be traversed.
- Following the `pred` table generated from Dijkstra's, starting from `pred[dst_node]`, until `None` is reached
    - **Role**: From the `dist` table created by Dijkstra's algorithm noting the distance from the source node to each of the other nodes, find the shortest path between the source and the destination.
    - **Justification**: Since `pred` contains the predecessor for each node in the shortest path from the source node to it, following it is a reliable way to find the

complete shortest path. Since `pred[src_node]` is `None` by definition, reaching it indicates that the path is complete. Notably, we don't stop when we reach `src_node`, since that would exclude `src_node` from the path object that is returned.

- Heap sort
    - **Role**: Finding n closest POIs in `find_nearby`
    - **Justification**:
        - While ordering the POIs in the relevant category, insertion is O(log n) compared to O(n) for a sorted linked list or array.
        - Looking up minimum is O(1) (same as for sorted linked list or array).
        - However, removing the minimum is O(log n) for priority queue (which is lower than O(1) for a well-implemented sorted linked list (where the head is changed) or sorted array (where only the start pointer of the array is moved, and a modulo with the array length is used to insert new elements).
        - However, for *total* time complexity of sorting operations in `find_nearby`, a priority queue performs much better: O(inserting n elements + looking up n minima + removing n minima) = O(n log n + n + n log n) = O(n log n) for a priority queue. Compare this to a sorted array or linked list's $O(n^2 + n + n) = O(n^2)$. So, a priority queue is chosen for `find_nearby`. (Note that this is not an argument about the total time complexity of `find_nearby` itself, which also has an O(e log e) call to Dijkstra's.)

Appendix

My stress test reimplementations with a hundred segments and POIs:

```
let N_ELEMS = 100
time 'initialization stress with N_ELEMS = %p'.format(N_ELEMS):
    let segs = [None for i in range(N_ELEMS)]
    let pois = [None for i in range(N_ELEMS)]
    for i in range(N_ELEMS):
        segs[i] = [i, i, i+1, i+1]
        pois[i] = [i, i, "coffee", "Starbucks #%p".format(i + 1)]

    let t = TripPlanner(segs, pois)

time 'full stress with N_ELEMS = %p'.format(N_ELEMS):
    let segs = [None for i in range(N_ELEMS)]
    let pois = [None for i in range(N_ELEMS)]
    for i in range(N_ELEMS):
        segs[i] = [i, i, i+1, i+1]
        pois[i] = [i, i, "coffee", "Starbucks #%p".format(i + 1)]

    let t = TripPlanner(segs, pois)
    let result = t.find_nearby(0, 0, "coffee", 1)
    result = Cons.to_vec(result)
```

I got the following output:

```
timing {label: 'initialization stress with N_ELEMS = 100', cpu: 562,
real: 589, gc: 78, result: None}
timing {label: 'full stress with N_ELEMS = 100', cpu: 3937, real:
3943, gc: 515, result: None}
```

I repeated the stress test many times to make sure the low runtime was not a one-off. But only one output is shown here for brevity's sake.