Ethan Sterling

CS 214

Prof. Vincent St-Amour

March 14, 2023

Trip Planner Project Report

ADTs and Algorithms:

- WuGraph (_data)
    - It represents the positions as its vertices and the road segments as its edges with a weight equal to the standard Euclidean distance. By using dijkstra's algorithm on this graph, we can find the shortest path between two points and their distances. This is essential information for plan_route and find_nearby.
    - Adjacency Matrix
    - An adjacency matrix has a set_edge and get_edge of O(1) while an adjacency list is O(d). While an adjacency list is more space efficient for sparse graphs, the methods in TripPlanner will often be used in large cities with lots of connecting roads, such as in the Chicago grid system. By using a matrix over a list, we get a better time complexity for constructing and using the graph and we do not sacrifice too much space complexity in large cities, where the majority of people reside.
- Dictionary (_node_to_position)
    - Allows to go from a vertex on the graph _data to the position it represents. The plan_route method needs to have an output in terms of positions and this

    dictionary provides a way to switch from the nodes dijkstra's returns in its predecessor vector to positions.

  - Direct Addressing Vector
  - This is a great data structure to use as the vertices of _data are all natural numbers so the keys in the dictionary can be natural numbers as well. A direct addressing vector always has a time complexity of O(1) for get and set. This is better than the worst-case time complexity of a hash table, O(n), when there are collisions and better than other data structures for dictionaries where time complexity is worse (i.e. an association list). The downside of a direct addressing vector is that it has a fixed size; however, we can set it to the number of road segments multiplied by 2 because that is the maximum number of positions there can be in a vector of road segments.

- Dictionary (_position_to_node)
  - Allows to go from a position to the vertex on the graph _data that represents it. For plan_route and find_nearby the input is a starting latitude and longitude. This dictionary allows us to go from this input to a vertex in the graph _data so we can use dijkstra's algorithm.
  - Hash Table
  - A hash table allows us to go from a vector, [Lat?, Lon?] to an integer and set the value of a vector to an integer with an average time complexity of O(1). By setting the number of buckets to the number of segments, we have a load factor less than or equal to 2 because the number of vectors is bounded by 2 times the number of road segments (a very unlikely worst-case scenario too). This load

factor is desirable for a separate chaining hash table to ensure an average time complexity of O(1) and efficient space usage. This is better than alternative dictionaries such as an association list or sorted array because a time complexity of O(1) is more efficient than a time complexity of O(n) or O(log n).

- Dictionary (_cat_to_positions)
    - Keys are categories (string) and values are a linked list of positions that contain a POI with that category. This dictionary is essential for the locate_all method as it just calls get on the inputted category.
    - Hash Table
    - Similar to the benefits of _position_to_node, a hash table has a get and set with an average time complexity of O(1). This allows the dictionary to be populated in a time complexity O(n), the fastest we can possibly achieve for n inputs, and locate_all to return the desired output in a time complexity of O(1). By setting the number of buckets to the number of positions, we will have a load factor less than 2 when there is an average of less than 2 categories per position which is a reasonable assumption given some positions have no categories at all. An even better data structure would be a hash table that can resize because then we would not have to worry about load factor. This hash table has better time complexity than an association list because O(1) is faster than O(n) for get/set.
- Dictionary (_name_to_position)
    - Allows us to go from a POI name to its position. This is essential for plan_route because our input is a name and we need the position of that POI to find the shortest path from the starting position to that position.

  - Hash Table
  - Similar reasoning to the _cat_to_positions hash table, get and set will have an average time complexity of O(1) for get/set. The number of buckets is the number of positions and load factor will stay under two assuming an average of less than 2 POIs per position. An even better data structure would be a hash table that can resize because then we would not have to worry about load factor. This hash table has better time complexity than an association list because O(1) is faster than O(n) for get/set.
- Dictionary (_position_to_POIs)
  - Allows us to go from a position to a linked list of POIs at that position. Helpful for find_nearby as we must return a linked list of POIs based on how close the position of the POI is to the starting position.
  - Hash Table
  - Similar to the previously mentioned hash tables, this hash table has an average time complexity of O(1) for get/set. The load factor will always be 1 or less, which is less than 2, because the number of buckets is the number of positions and the keys are the positions with POIs. This hash table has better time complexity than an association list because O(1) is faster than O(n) for get/set.
- Priority Queue (pq in find_nearby)
  - Allows us to quickly sort all the positions containing the imputed category by distance from the starting position.
  - Binary Heap

  - A binary heap allows us to have a O(log n) time complexity for both insert and remove_min. This is better than having a O(n) time complexity for either insert or remove_min which a sorted or unsorted list provides, respectively. In addition, we can have a custom less than function which compares the distances of each position from the starting category.
- Dijkstra's Algorithm (dijkstra)
  - Dijkstra's algorithm provides us with the shortest distance from a starting vertex in _data to all the other vertices in _data as well as the predecessor of each vertex to traverse this shortest distance. We use the predecessors it returns to solve plan_route as we can find the sequence of positions to get from the starting position to the destination. We use the distances Dijkstr's returns to solve find_nearby as we can compare the distances of each position with the desired category to the starting position.
  - Dijkstra's is better than Bellman-Ford because while Bellman-Ford needs to relax every edge |V|-1times, Dijkstra's algorithm only relaxes each edge once. This is because Dijkstra's has a specific order in which it relaxes the edges where Bellman-Ford has a random order. Due to this, Dijkstra's algorithm will have a time complexity of $O(|V|^2 \log |V|)$ which is nicer than Bellman-Ford's complexity of $O(|V|^3)$.
- Priority Queue (pq in dijkstra)
  - Dijkstra's algorithm uses this priority queue to determine the order in which it relaxes edges.
  - Binary Heap

- A binary heap allows us to have a O(log n) time complexity for both insert and remove_min. This is better than having a O(n) time complexity for either insert or remove_min which a sorted or unsorted list provides, respectively. In addition, we can have a custom less than function which compares the distances of each vertex from the starting vertex.