Bradley Pawlow
CS 214: Data Structures & Algorithms
Professor St-Amour
December 3, 2020

Homework 7 Report

ADT Section

- **Weighted, Undirected Graph:** _map_graph
    - This ADT allowed for easy storing of distances between different POIs or locations on the map that were connected via roads. Each vertex represents a location/POI, while each edge was the distance between these two points.
    - Adjacency matrix with weights
    - The adjacency matrix was easy to import and create since I had already used it for homework 4. Also, I am very familiar with indexing in arrays and wanted to take advantage of the constant time complexity of setting/getting edges between two vertices rather than iterating through an adjacency list as the alternative option.
- **Stack:** new_names
    - This ADT allowed me to easily add and remove POI names at each location for each category (*see _category_pos_hash for more details*). Used for easy inserting when a new name is found and easy removal for the find_nearby function.
    - Singly linked List
    - The time complexity is constant for push() and pop() so I can easily insert POI names when initializing the TripPlanner class and removing those names for find_nearby(). Also, rather than using an array and worrying about memory allocation, with a singly linked list I can add as many names to the data structure as I want. Plus, it was already implemented in homework 2 for easy access.
- **Stack:** temp_list/saved_data
    - This ADT allowed me to keep track of the POI names that are found when going through each location of the map within the find_nearby() implementation.

Saved_data restores the POI names back into the hash table while temp_list contributes to the final solution.

  - Singly linked List
  - The time complexity is constant for push() and pop() so I can easily insert POI names when iterating through each of the vertices in the graph and easily remove them from the respective hash table value. Also, rather than using an array and worrying about memory allocation, with a singly linked list I can add as many names to the data structure as I want. Plus, it was already implemented in homework 2 for easy access.

- **Dictionary**: _category_present_hash
  - This ADT allowed for easy checking of whether a given POI category (key) is present anywhere on the map.
  - Hash table with open addressing
  - Hash tables have constant lookup and insertion time complexities on average if it is within a proper load factor (buckets always less than or equal to the number of vertices in my graph ADT). This is much more efficient than just storing key-value pairs in a vector or association list; in addition, there is always a possibility of hash collisions and open addressing properly handles those cases. It was also easy to import an open addressing hash table interface and class from homework 3.
- **Dictionary**: _category_pos_hash
  - This ADT allowed me to hold information of each POI category and its latitude/longitude location, as a key, and assign all the POI names (value) for that key. I could ultimately use all the POI names from the specific category and position to return in the find_nearby function.
  - Hash table with open addressing *(key = struct, value = stack / linked list)*
  - Hash tables have constant lookup and insertion time complexities on average if it is within a proper load factor (buckets always less than or equal to the number of vertices in my graph ADT). This is much more efficient than just storing key-value pairs in a vector or association list; in addition, there is always a possibility of hash collisions and open addressing properly handles those cases. It was also

    - easy to import an open addressing hash table interface and class from homework 3 and use it to store a struct as key and a linked list as a value.

- **Dictionary**: _name_pos_hash
    - This ADT allowed me to hold information of each POI name (key) and its latitude/longitude location as a two-element vector (value). It is primarily used in find_route() to convert the destination name to a valid position.
    - Hash table with open addressing *(key = string, value = two-element vector)*
    - Hash tables have constant lookup and insertion time complexities on average if it is within a proper load factor (buckets always less than or equal to the number of vertices in my graph ADT). This is much more efficient than just storing key-value pairs in a vector or association list; in addition, there is always a possibility of hash collisions and open addressing properly handles those cases. It was also easy to import an open addressing hash table interface and class from homework 3 and use it to store a string as a key and a vector as a value.
- **Dictionary**: _pos_hash
    - This ADT allowed me to assign a vertex number (natural number) to each unique latitude/latitude position on the graph (initially a two-element array). This helped with utilizing the graph functions because the graph implementation works with vertex numbers from 0 to n-1 (n = number of vertices).
    - Hash Table with Open Addressing *(key = two-element array, value = natural number)*
    - Hash tables have constant lookup and insertion time complexities on average if it is within a proper load factor (buckets always less than or equal to the number of vertices in my graph ADT). This is much more efficient than just storing key-value pairs in a vector or association list; in addition, there is always a possibility of hash collisions and open addressing properly handles those cases. It was also easy to import an open addressing hash table interface and class from homework 3 and use it to store an array as a key and a natural number as a value.
- **Dictionary**: _name_POI_hash

    - This ADT allowed me to store the POI name (key) with all the information regarding that POI (value). This was ultimately used in find_nearby() when converting all the POI names to the proper 4-element vector that is returned.
    - Hash table with open addressing *(key = string, value = 4-element vector)*
    - Hash tables have constant lookup and insertion time complexities on average if it is within a proper load factor (buckets always less than or equal to the number of vertices in my graph ADT). This is much more efficient than just storing key-value pairs in a vector or association list; in addition, there is always a possibility of hash collisions and open addressing properly handles those cases. It was also easy to import an open addressing hash table interface and class from homework 3 and use it to store a string as a key and a vector as a value.
- **Priority Queue:** todo
    - This ADT allowed me to store information of the different vertices in the graph that still need to be visited and continually sorts the vertices based on priority (closest distances). This was used in my Dijkstra's algorithm to relax edges in a clever order.
    - Binary heap, storing elements in an array
    - Binary heaps have better time complexities (O(logn) for inserting and removing) than a general array or linked list and are already defined in homework 5. Also, I was able to use this ADT to access heapsort and efficiently sort all the shortest distances from a starting vertex to all other vertices in dijkstra's algorithm.

Algorithms Section

- **Dijkstra's Algorithm**
    - This algorithm allowed to me to visit each of the vertices in the graph connected to a given starting vertex and store the minimum distances to each vertex (dist array) and the specific path (pred array). It also loops through the nearby edges and vertices by proximity to determine the priority of the next vertices to visit. We can sort these distances to find the closest vertices from the starting point and use this to ultimately find the closest POIs in find_nearby. Also, by knowing each

of the shortest paths to each vertex from the given starting point, we can thus locate the shortest path to a specific POI in find_route by looping through the predecessors.
  - This algorithm is a more efficient way to avoid brute force relaxation by storing edges based on a clever order rather than having to visit all of them like in the Bellman-Ford algorithm.
- **Heap Sort**
  - This sorting algorithm is useful for taking the dist array, sorting the distances from smallest value to largest value, and returning the new array. This is helpful because allows for easy indexing and easy looping through each of the vertices with the shortest distance from a given starting point. This is ultimately used in find_nearby to check whether a given location has the POI category we are looking for and if the program should add it to the list of nearby POIs.
  - I picked this algorithm due to its relatively fast time complexity of O(nlogn) compared to slower sorting algorithms (worst case scenario) such as insertion or selection sort. It was also easy for me to access the algorithm since it is already defined in the BinHeap class from homework 5.