Analysis of Alternatives

“position” entity

The TRIP_PLANNER.plan_route method requires finding a shortest path in a graph where vertices are positions (RawPos?). The provided graph implementation WUGraph was chosen. In this graph implementation, vertices are natural numbers completely filling an interval. Since natural numbers are more compact than RawPos?s, identification of positions by natural numbers was chosen as the main method of identification, and the contract PosI? was introduced for such identifiers.

- PosAttr
  - Represents a position. Contains position attributes. All hash tables below that need to map to positions map to PosAttr.
  - struct
  - struct is ideal for storing a fixed number of values (attributes) of various types.
- TripPlanner._pos_to_pos_attr
  - Finds a position by its RawPos?. Converts from input RawPos? to internal PosI?.
  - HashTableAll?[RawPos?, PosAttr?]
  - An array would be more efficient. However, in order to use an array, we need to convert RawPos? to a natural number, and for that we need to know bounds on coordinates of RawPos?. A search tree is less efficient. Search in a search tree requires $O(\log n)$ time, but search in a hash table requires $O(1)$ time on average. Linear search is even less efficient because it requires $O(n)$ time.
- TripPlanner._pos_i_to_pos_attr.
  - Finds a position by its PosI?. Converts from internal PosI? to output RawPos?.
  - Array VecC[OrC(NoneC, PosAttr?)]
  - Because PosI?s completely fill an interval, it is possible to use an array. A hash table is less efficient than an array because every search involves computing a hash when using a hash table. A search tree or linear search is less efficient than an array (see TripPlanner._pos_to_pos_attr to know why).

“POI” entity

- TripPlanner._poi_to_pos_attr
  - Finds a position of a POI identified by its name.
  - HashTableAll?[Name?, PosAttr?]

  - The only attribute of a POI is its name, and this attribute is unique. Thus a POI is identified by its name and does not require a struct. A hash table was chosen for the same reason as for TripPlanner._pos_to_pos_attr.

“category” entity

- TripPlanner._cat_to_set_pos
  - Finds positions that contain at least one POI in the given category (for TripPlanner.locate_all). Finds POIs in the given category at the given position (for TripPlanner.plan_route).
  - HashTableAll?[Cat?, HashTableAll?[PosI?, ListC[RawPOI?]]]
  - The only attribute of a category is its name, and this attribute is unique. Thus a category is identified by its name and does not require a struct. Hash tables were chosen for the same reason as for TripPlanner._pos_to_pos_attr.

Dijkstra’s algorithm

- TripPlanner._graph
  - A graph where vertices are positions and edges are road segments. Used to traverse paths between positions and POIs. In input data, road segments are represented by RawSeg?, but internally, they are stored as graph edges.
  - WUGraph
  - Graph implementations are complex, thus the provided implementation was used to save effort.
- DijkstraIterator
  - An implementation of Dijkstra’s algorithm which returns vertices one-by-one.
  - Class
  - The methods TripPlanner.plan_route and TripPlanner.find_nearby use Dijkstra’s algorithm in different ways. TripPlanner.plan_route terminates the algorithm when it obtains the destination vertex (dst). TripPlanner.find_nearby terminates the algorithm when it obtains enough vertices (positions) with POIs in the given category. DijkstraIterator is used in both methods.
- DijkstraIterator._v_attr.
  - Stores vertex attributes needed in Dijkstra’s algorithm.
  - Array VecC[DijkstraVertex?]
  - An array was chosen for the same reason as for TripPlanner._pos_i_to_pos_attr.
- DijkstraIterator._sorted_v.
  - Finds a vertex at the least distance in Dijkstra’s algorithm.

- Minimum binary heap BinHeap?[DijkstraSortedVertex?] stored in an array
- The following operations are used: extracting a minimal element from the data structure; inserting an element into the data structure. Both operations on a binary heap require $O(\log n)$ time. If a list or an array is used, inserting requires $O(1)$ time, but extracting requires $O(n)$ time; this is less efficient. If a search tree is used, both operations require $O(\log n)$ time. A binary heap stored in an array was chosen because it does not use pointers or heap allocation like a search tree, so it is more efficient. If a Fibonacci heap is used, inserting requires $O(1)$ time, and extracting requires $O(\log n)$ time; this is more efficient asymptotically than a binary heap. However, the real time depends on the size of input data which is not known. Another reason a binary heap was chosen is that a Fibonacci heap is difficult to implement and a binary heap implementation is provided, but a Fibonacci heap is not. A deficiency of the provided implementation is that it has no operation for decreasing the distance of a heap element. Instead, we add an element to a heap with the same vertex and the new distance, and the old element is considered *outdated*. Another option is to reimplement binary heap; it was not chosen to save effort.