Homework 7: Trip Planner

Code due: Tue., Dec. 3 at 11:59 PM (via GSC)
Self-eval due: Fri., Dec. 6 at 11:59 PM (on GSC)

You may work on your own or with one (1) partner.

For this assignment you will implement a trip planning API that provides routing and searching services. Unlike previous homework assignments, the representation is not defined for you, or nor have we specified which data structures and algorithms to use. Instead, you may choose from among the data structures you have implemented for previous assignments—hash table, graph, binary heap, and union-find—as well as associated algorithms that you either implemented or saw in class. Selecting appropriate data structures and algorithms is up to you, and your grade will reflect these choices.

Contents

1 Problem Overview **2**
1.1 Entities 2
1.2 Queries 2

2 API Specification **4**
2.1 Vocabulary Types 4
2.1.1 Basic Types 5
2.1.2 Entity Types 5
2.1.3 Collection Types 5
2.2 The `TripPlanner` Class and `TRIP_PLANNER` Interface 6
2.2.1 Creating a `TripPlanner` Instance 6
2.2.2 Using a `TRIP_PLANNER` Instance 6

3 Getting Started **7**
3.1 Dependencies 9

4 Deliverables **10**

1 Problem Overview

Your trip planner will store map data representing three kinds of entities and answer two kinds of queries about them. The following two subsections describe the entities and the queries in turn.

1.1 Entities

- A *position* has a latitude and a longitude, both numbers.
- A *road segment* has two endpoints, both positions.
- A *point-of-interest* (*POI*) has a position, a category (a string), and a name (a string).

See figure 1 on the following page for an example of a map containing the three kinds of entities. Note that each position can contain zero, one, or more POIs. While this example map is aligned to an integer grid, you should not assume this in general.

We will make two assumptions about the segments and their positions:

1. The length of a road segment is the standard Euclidian distance between its endpoints.
2. Points-of-interest only appear at positions that also occur as the endpoint of some road segment.

1.2 Queries

Your trip planner must support two forms of queries:

find-route Takes a starting position (latitude and longitude) and the name of a point-of-interest; returns a shortest path from the starting position to the named point-of-interest.

find-nearby Takes a starting position (latitude and longitude), a point-of-interest category, and a limit n; returns the (up to) n points-of-interest in the given category nearest the starting position.

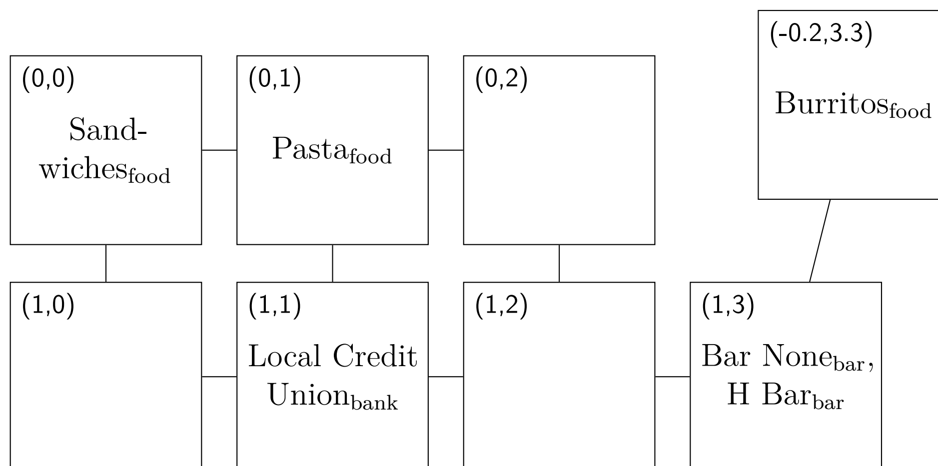

Figure 1: Example map with POIs labeled by latitude and longitude

Start	**Name**	**Result path**
(0,0)	Sandwiches	(0,0)
(0,1)	Sandwiches	(0,1)–(0,0)
(1,1)	Sandwiches	(1,1)–(1,0)–(0,0) *or* (1,1)–(0,1)–(0,0)
(1,1)	Burritos	(1,1)–(1,2)–(1,3)–(1,4)
(1,1)	Sushi	*nothing*

Table 1: Example *find-route* queries

Start	**Category**	n	**Result set**
(1,3)	food	1	{Burritos}
(0,2)	food	1	{Pasta}
(0,2)	food	2	{Pasta, Sandwiches}
(0,2)	food	3	{Burritos, Pasta, Sandwiches}
(0,2)	food	4	{Burritos, Pasta, Sandwiches}
(0,2)	bar	1	{Bar None} *or* {H Bar}
(0,2)	bar	2	{Bar None, H Bar}
(0,2)	bar	3	{Bar None, H Bar}
(0,2)	school	5	{}

Table 2: Example *find-nearby* queries

Representation	**Alias**	**Purpose**
`num?`	`Lat?`	latitude
`num?`	`Lon?`	longitude
`str?`	`Cat?`	POI category
`str?`	`Name?`	POI name
`[Lat?, Lon?]`		position
`[Lat?, Lon?, Lat?, Lon?]`		road segment
`[Lat?, Lon?, Cat?, Name?]`		POI
vector of road segment		input road segments
vector of POI		input POIs
linked list of position		*find-route* result
linked list of POI		*find-nearby* result

Table 3: Summary of vocabulary types

For some example queries and results see table 1 and table 2 on the previous page.

2 API Specification

The trip planner API is specified as a DSSL2 interface named `TRIP_PLANNER`, which you must implement as a class named `TripPlanner` in a file named `planner.rkt`.

The `TRIP_PLANNER` interface refers to a variety of "vocabulary types" in order to represent the three kinds of entities discussed in section 1.1 on page 2, as well as collections thereof. Thus, before describing the interface itself, we must define these types.

2.1 Vocabulary Types

The types described in this section are defined in three layers in the next three subsections: basic types, entities, and collections of entities. All the types are summarized in table 3.

2.1.1 Basic Types

The basic vocabulary types include latitude and longitude (represented as numbers) and POI categories and names (represented as strings):

```
let Lat?  = num?
let Lon?  = num?
let Cat?  = str?
let Name? = str?
```

2.1.2 Entity Types

The three entity types are represented as vectors (not structs) of basic types. In particular:

- A position is represented as a 2-element vector containing the latitude and longitude: `[Lat?, Lon?]`.
- A road segment is represented as a 4-element vector containing the latitude and longitude of one end position followed by the latitude and longitude of the other: `[Lat?, Lon?, Lat?, Lon?]`.
- A point-of-interest is represented as a 4-element vector containing the latitude and longitude of its position, then its category, and then its name: `[Lat?, Lon?, Cat?, Name?]`.

These types are intended for communication between the API and the client, but you will probably want to define richer representations for internal usage in your implementation.

There are no contracts provided for recognizing these types.

2.1.3 Collection Types

The API uses collections of entities in two places:

- The constructor for your `TripPlanner` class needs to take a collection of road segments and a collection of points-of-interest from which to build its map. These will be passed as a vector of road segments and a vector of POIs, respectively.

```
let roads = [ [0,0, 1,0], [0,1, 1,1], [0,2, 1,2],
              [0,0, 0,1], [1,0, 1,1],
              [0,1, 0,2], [1,1, 1,2], [1,2, 1,3],
              [1,3, -0.2,3.3] ]

let pois  = [ [0,0,      'food', 'Sandwiches'],
              [0,1,      'food', 'Pasta'],
              [1,1,      'bank', 'Local␣Credit␣Union'],
              [1,3,      'bar',  'Bar␣None'],
              [1,3,      'bar',  'H␣Bar'],
              [-0.2,3.3, 'food', 'Burritos'] ]

let tp    = TripPlanner(roads, pois)
```

Figure 2: Instantiating a `TripPlanner` for the map in figure 1 on page 3

- The queries defined by the `TRIP_PLANNER` interface (and thus implemented by your `TripPlanner` class) need to return sequences of positions (paths) and sets of nearby POIs. These results must be represented as linked lists (using `cons`) of positions and POIs, respectively.

2.2 The `TripPlanner` Class and `TRIP_PLANNER` Interface

2.2.1 Creating a `TripPlanner` Instance

Your `TripPlanner` class must define a constructor that takes two arguments: a vector of road segments and a vector of points-of-interest, each as described in section 2.1.3 on the preceding page. For an example of how to instantiate a `TripPlanner` instance see figure 2.

2.2.2 Using a `TRIP_PLANNER` Instance

Your `TripPlanner` class must implement the `TRIP_PLANNER` interface (figure 3 on the following page), which defines a method corresponding to each of the two trip planner query operations described in section 1.2 on page 2. For an example of how these operations are used, see figure 4 on page 8.

```
interface TRIP_PLANNER:
    def find_nearby(
            self,
            src_lat:  Lat?,     # starting latitude
            src_lon:  Lon?,     # starting longitude
            dst_cat:  Cat?,     # POI category
            n:        nat?      # maximum number of results
        )   ->        List?     # linked list of POIs

    def find_route(
            self,
            src_lat:  Lat?,     # starting latitude
            src_lon:  Lon?,     # starting longitude
            dst_name: Name?     # POI name
        )   ->        List?     # linked list of positions
```

Figure 3: The `TRIP_PLANNER` interface

3 Getting Started

There's no starter planner.rkt file provided, but there is a starter library that you should download[1]. It unzips into a directory named hw7-lib that contains two items:

- a file interfaces.rkt that provides type and interface definitions that you will need, and
- a subdirectory compiled that contains compiled versions of our solutions to Homeworks 3 through 6.

Once you've unzipped hw7-lib.zip, you should create your planner.rkt file in the resulting hw7-lib directory alongside interfaces.rkt. The first line of planner.rkt must be `#lang dssl2` to tell DrRacket what language you are using. After that, you should `import 'interfaces.rkt'` in order to get the `TRIP_PLANNER` interface. Additional imports are will be needed for your code to depend on prior homework assigments. The next subsection explains how to set that up.

[1] `https://bit.ly/33aM6Q5`

```
assert tp.find_route(1,2, 'Sushi')    is None

assert tp.find_route(1,2, 'Burritos') \
       == cons([1,2], cons([1,3], cons([-0.2,3.3], None)))
assert tp.find_route(1,3, 'Burritos') \
       == cons([1,3], cons([-0.2,3.3], None))
assert tp.find_route(-0.2,3.3, 'Burritos') \
       == cons([-0.2,3.3], None)

assert tp.find_nearby(1,3, 'food', 1) \
       == cons([-0.2,3.3, 'food', 'Burritos'], None)
assert tp.find_nearby(0,2, 'food', 1) \
       == cons([0,1, 'food', 'Pasta'], None)
assert tp.find_nearby(0,2, 'food', 2) \
       == cons([0,1, 'food', 'Pasta'],
               cons([0,0, 'food', 'Sandwiches'], None))

assert tp.find_nearby(0,2, 'bar', 1) in [
    cons([1,3, 'bar', 'Bar␣None'], None),
    cons([1,3, 'bar', 'H␣Bar'], None)
]

assert tp.find_nearby(0,2, 'bar', 2) in [
    cons([1,3, 'bar', 'Bar␣None'],
         cons([1,3, 'bar', 'H␣Bar'], None)),
    cons([1,3, 'bar', 'H␣Bar'],
         cons([1,3, 'bar', 'Bar␣None'], None))
]
```

Figure 4: Using a TRIP_PLANNER of the map in figure 1 on page 3

3.1 Dependencies

Your trip planner will most likely depend on some parts of Homeworks 3 through 6. You don't need to resubmit those old homeworks with this one, however, nor do you need to test with your old code, since `hw7-lib.zip` contains our compiled solutions—though you may use your own solutions if you wish.

Even if your own solutions work, our solutions include a few enhancements that you may want to take advantage of (or replicate):

- The provided hash table provides three significant enhancements over the HW3 hash table:
  - It grows automatically to maintain a reasonable load factor.
  - Looking up an absent key with `get` is no longer an error; instead, it returns `None` when the key isn't found.[2]
  - A **`for`** loop can be used to iterate over all the key–value associations in the table; see `interfaces.rkt` for details.

  We also provide a function `hash_table()` that takes 0 arguments and returns a small hash table with a good hash function.

- Instead of one graph class `WuGraph`, we provide two, `AdjMatWUG` and `AdjListWUG`, with the same interface. The predicate `WuGraph?` returns true for either class, and there's a function `WuGraph` that constructs one or the other (unspecified) for backward compatibility.

For each homework, 3 through 6, that you wish you use, you may choose between your implementation and ours, as follows:

Ours. To depend on one of our solutions, don't copy your solution into the `hw7-lib` directory. Instead, just import the corresponding `.rkt` file in `planner.rkt`, even though it doesn't exist.[3]

For example, to use our HW5 solution, just write **`import`** `'binheap.rkt'`, and DSSL2 will find our compiled code inside the `compiled` subdirectory.

[2] This makes some algorithms easier to write, though it means that if the value might be `None` then you need to check explicitly using the `mem?` method.

[3] If you get an error saying that the file isn't found, try upgrading DSSL2.

Yours. To depend on one of your own solutions, copy its .rkt file into the hw7-lib directory alongside your planner.rkt. Modify your solution to **import** 'interfaces.rkt' and remove any definitions—usually interfaces—that it will get from the new import instead. Then import your solution from planner.rkt.

For example, to use your own HW5 solution, you need to:

1. Copy your binheap.rkt into the hw7-lib directory.
2. Edit binheap.rkt to remove the definition of the PRIORITY_QUEUE interface and **import** 'interfaces.rkt' instead.
3. Add **import** 'binheap.rkt' to your planner.rkt.

For grading, we will use any dependencies that you submit and provide our own solutions for those that you don't. If you have changed partners since Homeworks 3–6, you may submit either partner's old solutions. Whatever you choose, it is **strongly recommended** that you test your code against the same dependencies that we will test it against for grading.

4 Deliverables

Your submission may contain multiple files, but it **must** contain a file named planner.rkt that provides the TripPlanner class as specified in this document.

You must include tests demonstrating that your implementation works; as usual, you will be graded for coverage. We recommend that you write your tests in a separate .rkt file that imports planner.rkt, rather than in planner.rkt directly. Name it anything you like, but don't forget to upload it.

Additional submitted files may include dependencies such as your code from prior homework assignments or new definitions for this homework that you wish to organize. You may name them anything you like, bearing in mind that any file you submit with a given name will take priority over our solution file having the same name.