Final Project: Trip Planner

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, nor have I specified which data structures and algorithms to use. Instead, you may choose from among the data structures you have implemented for previous assignments as well as any algorithms that we either implemented or saw in class. Selecting appropriate data structures and algorithms is up to you, and your grade will reflect these choices.

In `planner.rkt`, I've supplied a definition of the interface you need to implement, as well as auxiliary definitions discussed in this document.

Preamble

Most of the learning value of this assignment comes from figuring out the overall structure, moving pieces, and how they fit together, and less from the implementation of that plan. To get the most out of this assignment, I therefore discourage discussing even a general approach with your colleagues.

Also, be warned that this assignment is significantly larger and longer than previous ones. **I recommend you start as early as you can.**

1 Problem Overview

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

1.1 Items

- 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). The name of a point-of-interest is unique across all points-of-interest, but a category may be shared by multiple points-of-interest. Each position can feature zero, one, or more POIs.

See figure 1 for an example of a map containing the three kinds of items.

We will make three assumptions about segments and positions:

1. All roads are two-way roads.
2. The length of a road segment is the standard Euclidian distance between its endpoints.
3. Points-of-interest can only be found at a road segment endpoint.

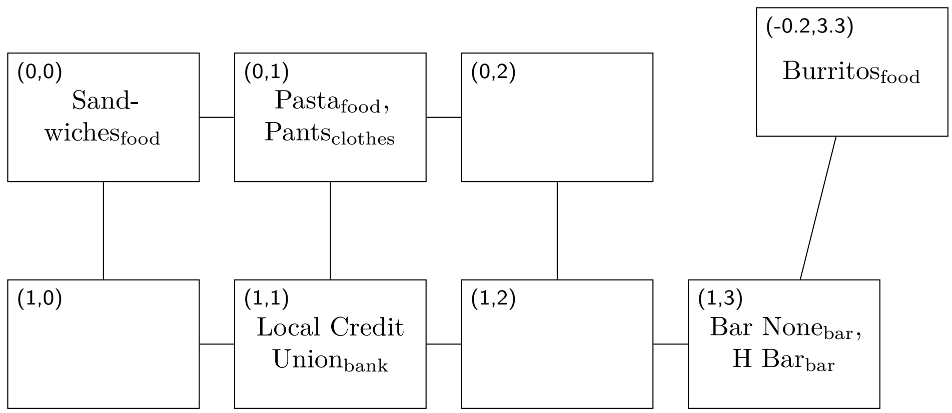

Figure 1: Example map with POIs labeled by latitude and longitude

Category	**Result positions**
food	(0,0), (0,1), (-0.2,3.3)
bank	(1,1)
bar	(1,3)
barber	*nothing*

Table 1: Example *locate-all* queries

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)–(-0.2,3.3)
(1,1)	Sushi	*nothing*

Table 2: Example *plan-route* queries

Start	**Category**	n	**Result POIs**
(1,3)	food	1	Burritos
(0,2)	food	1	Pasta
(0,2)	food	2	Pasta, Sandwiches (in any order)
(0,2)	food	3	Pasta, Sandwiches, Burritos (in any order)
(0,2)	food	4	Pasta, Sandwiches, Burritos (in any order)
(0,2)	bar	1	Bar None *or* H Bar
(0,2)	bar	2	Bar None, H Bar (in any order)
(0,2)	bar	3	Bar None, H Bar (in any order)
(0,2)	school	5	*Nothing*

Table 3: Example *find-nearby* queries
(your implementation should return the full POIs, not just the names)

Representation	**Contract**	**Purpose**
`num?`	`Lat?`	latitude
`num?`	`Lon?`	longitude
`str?`	`Cat?`	POI category
`str?`	`Name?`	POI name
`[Lat?, Lon?]`	`RawPos?`	raw position
`[Lat?, Lon?, Lat?, Lon?]`	`RawSeg?`	raw road segment
`[Lat?, Lon?, Cat?, Name?]`	`RawPOI?`	raw POI
vector of road segments	`VecC[RawSeg?]`	input raw road segments
vector of POIs	`VecC[RawPOI?]`	input raw POIs
linked list of positions	`ListC[RawPos?]`	*locate-all* and *plan-route* result
linked list of POIs	`ListC[RawPOI?]`	*find-nearby* result

Table 4: Summary of vocabulary types

1.2 Queries

A complete trip planner must support three forms of queries:

locate-all Takes a point-of-interest category; returns the positions of all points-of-interest in the given category. The positions can returned be in any order you want, but the result should not include duplicates.

plan-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. You can assume the starting position is at a road segment endpoint. Returns the empty list if the destination does not exist or is unreachable.

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. You can assume the starting position is at a road segment endpoint. Resolve ties however you want, and order points-of-interest within the list in any order you want.

For some example queries and results see tables 1, 2, and 3. For specifics on the interface your code should conform to (i.e., precise input and output types), read on.

2 API Specification

The trip planner API is specified as a DSSL2 interface named `TRIP_PLANNER` (provided in the starter code), which you must implement as a class named `TripPlanner`.

The `TRIP_PLANNER` interface refers to a variety of "vocabulary types" that represent the three kinds of items discussed in section 1.1, as well as other types used by `TRIP_PLANNER` operations. All these types are summarized in table 4.

2.1 Basic Types

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

2.2 Raw Item Types

In the external API that your `TripPlanner` class must support, items are represented in "raw" form, i.e., as vectors (not structs) of basic types. Specifically:

- A raw position is represented as a 2-element vector containing a latitude and a longitude.
- A raw 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.
- A raw 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.

These types are intended for communication between your `TripPlanner` class and the client; you will probably want to define richer representations for internal usage in your implementation.

2.3 Input Types

Your `TripPlanner` constructor must take two arguments:

- A vector of raw road segments.
- A vector or raw points-of-interest.

Both element types are as described in section 2.2.

In the starter code, I have provided a small example test that creates a `TripPlanner` instance with inputs in the correct format. You can use it to make sure your implementation conforms to the interface.

2.4 Output Types

The results for each of the three kinds of queries must be expressed in terms of the raw item types. Specifically:

- *locate-all* must return its result as a linked list (using the `cons` library) of raw positions. The elements of the list can be returned in any order, and each position may only appear once in the result.
- *plan-route* must also return a linked list (also using the `cons` library) of raw positions. The positions must be in the order in which they appear in the path from the source to the destination. For non-existent or unreachable destinations, *plan-route* must return the empty list.
- *find-nearby* must return a linked list (still using the `cons` library) of raw points-of-interest. The points-of-interest in the list can be in any order.

In the starter code, I have provided one small example test for each kind of query that checks that the output produced by your implementation are in the correct format. You can use them to make sure your implementation conforms to the interface.

3 Dependencies

Your solution is likely to rely on a number of different ADTs and data structures, including some which you've implemented as part of previous assignments.

As with homework 4, I'm providing you with compiled versions of solutions, this time for homeworks 2, 3, 4, and (eventually) 5. That way, everyone can start on firm footing regardless of the state of their previous homeworks.

As before, extract the `project-lib.zip` archive in the same directory as your `planner.rkt` file and import the files you need. The file names and interfaces are identical to the ones used for these assignments.

For example, `import 'project-lib/dictionaries.rkt'` would import a dictionary library whose interface is the one from homework 3. For reference, the graph representation used by our `graph.rkt` is an adjacency matrix.

Note: Be careful to recreate the exact directory structure described above and use the exact import syntax described above. Otherwise your code won't work, either when you try to run it yourself, or when I try to grade it.

Note: The compiled files I provide are specific to Racket version 8.7; if you use any other version (including a more recent one!), they will not work.

Alternatively, you can use your own homework solutions (e.g., if they are better suited to the task at hand). Because you will only turn in your `planner.rkt` file, you will need to copy over to it any homework code you wish to use.

You may also import modules from the DSSL2 standard library, such as the `cons` and `sbox_hash` libraries.

Any of the code I posted to Blackboard alongside lectures are also fair game. You'll just have to copy their implementation to your `planner.rkt` file. And be sure to acknowledge your sources.

4 Report

Iexpect you to put care and thought into your choices of ADTs, data structures, and algorithms. To assess this, you will need to write a brief report explaining your decisions.

For each use of an ADT in your program, list the following:

- What role does it play in solving the problem?
- What data structure did you pick for it?
- Why did you pick that data structure over other choices?

Each of these should be no more than a short paragraph. For conciseness and legibility, please format as a series of bullet points:

- *ADT (e.g., Stack)*
  - *role (1-2 sentences)*
  - *which data structure (e.g., linked list stack)*
  - *why this data structure (1 short paragraph)*
- *ADT (e.g., Queue)*
  - *role (1-2 sentences)*
  - *which data structure (e.g., ring buffer)*
  - *why this data structure (1 short paragraph)*
- ...

You don't need to describe intermediate structs, or intermediate arrays that are not meaningful sequences in their own right. But when in doubt, mention it.

Then, for each non-trivial algorithm in your program, explain the following:

- What role does it play in solving the problem?
- Why did you pick that algorithm over other choices?

Please follow the same format as for the ADT portion of the report.

By non-trivial algorithm, I mean an algorithm that is sophisticated enough to be worth discussing in its own right. For example, merge sort is non-trivial, but looping over an array to find the largest element is not. But when in doubt, mention it.

Note: Unlike self-evaluations for previous assignments, your report should describe your solution as it stands in your *final* submission.

Note: Even if you cannot get your program working, submit your report! If your design is sound, you can get credit for it with your report.

5 Advice

This project is a significant step up from earlier assignments both in terms of scale and in terms of challenge. To help you be successful, here are a few pieces of advice.

- **Start early!** Not only is there a lot to do, some of the design work can benefit from thinking about it on and off, while you do other things.
- Don't start writing code right away; planning out your design—as we did in the data design lectures—will save you time in the long run.
- You may want to implement *locate-all* (the easiest of the three queries) first, then try *plan-route*, and save *find-nearby* (the hardest of the three) for last. Of course, you'll also want to test each one thoroughly, to be confident in its correctness, before moving on to the next.
- When you get one of the queries working, *save a copy of your work*. That way, even if you end up breaking your program while working on another query, you'll at least be able to submit this older, partially working version and get feedback on it (and credit for it!) Better to submit a program that does less but does it successfully, than to submit a program that tries to do more but does not run at all.
- When designing your solution, you may be interested in using efficient, but somewhat complex algorithms or data structures. These can indeed be good choices, but getting them to work correctly may be difficult. If you find yourself struggling to get a complex algorithm or data structure to work, you may want to take a step back and (temporarily, at least) use a simpler equivalent instead. This simpler version can serve as a stopgap until you get the more complex solution to work, or as a fallback in case you don't get it working. Just be sure to explain your decision process in your report.
- The stakes on the project are quite high; see the syllabus. For this reason, if you find yourself falling behind on the project, you may want to prioritize it over studying for the second exam. Not only might it make more sense grades-wise, it likely makes more sense learning-wise as well: you would likely learn more by working on the project than by studying for the exam. Of course, doing both is ideal! So consider this a last resort.

6 Grading

Please submit your completed version of `planner.rkt`, containing:

- a definition of the `TripPlanner` class,
- and sufficient tests to be confident of your program's correctness.

Please also submit your report in `pdf` format to its (separate) Blackboard assignment.

Functional Correctness

We will use six separate test suites to test your submission:

- **Basic locate**: construction of the example trip planner from figure 1, locate-all with at most one POI per location.
- **Advanced locate**: locate-all multiple POIs per location, locate-all edge cases.
- **Basic route**: plan-route with one possible path.
- **Advanced route**: plan-route with multiple possible paths, plan-route edge cases, plan-route stress tests (to check for extreme inefficiencies).

- **Basic nearby**: find-nearby looking for the one closest relevant POI.
- **Advanced nearby**: find-nearby looking for multiple closest POIs, find-nearby edge cases, complex combinations of POIs and categories, find-nearby stress tests (to check for extreme inefficiencies).

To get credit for a test suite, your submission must pass *all* its tests.

The outcome your submission will earn will be determined as follows:

- **Got it**: passes all six test suites.
- **Almost there**: passes any four test suites.
- **On the way**: passes any two test suites.
- **Not yet**: does not achieve "on the way" requirements.
- **Cannot assess**: I could not successfully run our grading tests on your submission (usually due to a crash), which also means *we could not give you feedback*. **Please run your code before submitting!**

Non-Functional Correctness

There is no self-evaluation for the final project; I will therefore not be evaluating non-functional correctness for it. I nonetheless highly encourage you to continue applying the good habits you learned!

Report

When evaluating your report, I will be specifically looking for:

- **Justifications**: your choices of data structures and algorithms made must be well-motivated, with clear pros and cons and comparisons to alternatives. You may find it helpful to refer back to the first Data Design lecture for discussion of possible decision criteria.
- **Completeness**: your report must describe all the elements required for a correct trip planner. The precise number and makeup of these elements will necessarily differ between solutions, however, and I will take this into account.
- **Clarity**: your writing must be clear, precise, and concise. Technical terminology must be used accurately, and technical arguments your report makes must be correct.

Outstanding reports which excel at all of these criteria will earn a positive modifier. Reports which fall slightly short of these criteria, but are otherwise reasonable will earn a neutral modifier. Reports which are found lacking will earn a negative modifier.

Reference

The following functionality is provided by each library in `project-lib.zip`. The behavior of each export is as described in the relevant homework handout; this reference is merely a convenience.

`project-lib/stack-queue.rkt`

```
interface STACK[T]:
    def push(self, element: T) -> NoneC
    def pop(self) -> T
    def empty?(self) -> bool?

# The QUEUE interface is exported by the `ring_buffer` library
# interface QUEUE[T]:
#     def enqueue(self, element: T) -> NoneC
#     def dequeue(self) -> T
#     def empty?(self) -> bool?

class ListStack[T] (STACK):
    def __init__ (self)

class ListQueue[T] (QUEUE):
    def __init__ (self)
```

`project-lib/dictionaries.rkt`

```
interface DICT[K, V]:
    def len(self) -> nat?
    def mem?(self, key: K) -> bool?
    def get(self, key: K) -> V
    def put(self, key: K, value: V) -> NoneC
    def del(self, key: K) -> NoneC

class AssociationList[K, V] (DICT):
    def __init__(self)

class HashTable[K, V] (DICT):
    def __init__(self, nbuckets: nat?, hash: FunC[AnyC, nat?])

def first_char_hasher(s: str?) -> int?
```

project-lib/graph.rkt

```
let Vertex? = nat?
let VertexList? = Cons.ListC[Vertex?]

let Weight? = AndC(num?, NotC(OrC(inf, -inf, nan)))
let OptWeight? = OrC(Weight?, NoneC)

struct WEdge:
    let u: Vertex?
    let v: Vertex?
    let w: Weight?
let WEdgeList? = Cons.ListC[WEdge?]

interface WUGRAPH:
    def len(self) -> nat?
    def set_edge(self, u: Vertex?, v: Vertex?, w: OptWeight?) -> NoneC
    def get_edge(self, u: Vertex?, v: Vertex?) -> OptWeight?
    def get_adjacent(self, v: Vertex?) -> VertexList?
    def get_all_edges(self) -> WEdgeList?

class WuGraph (WUGRAPH):
    def __init__(self, size: nat?)

def sort_vertices(lst: Cons.list?) -> Cons.list?:
def sort_edges(lst: Cons.list?) -> Cons.list?:

def dfs(graph: WUGRAPH!, start: Vertex?, f: FunC[Vertex?, AnyC]) -> NoneC

def dfs_to_list(graph: WUGRAPH!, start: Vertex?) -> VertexList?
```

project-lib/binheap.rkt

```
interface PRIORITY_QUEUE[X]:
    def len(self) -> nat?
    def find_min(self) -> X
    def remove_min(self) -> NoneC
    def insert(self, element: X) -> NoneC

class BinHeap[X] (PRIORITY_QUEUE):
    def __init__(self, capacity, lt?)

def heap_sort[X](v: VecC[X], lt?: FunC[X, X, bool?]) -> NoneC
```