Homework 4: Graph

For this assignment you will implement an API for weighted, undirected graphs; then you will use this API to build some small graphs and write a depth-first search.

In `graph.rkt` I've supplied stubs for the code you'll need to write, along with a few suggested helpers and some code to help you with testing.

Orientation

The kinds of graphs we are interested in for this assignment are weighted, undirected graphs whose vertices are natural numbers. In particular, a graph of n vertices will have vertices numbered 0, 1, …, $n-1$. This makes it straightforward to associate information with each vertex in a vector of size n via direct addressing.

Before defining our signature for weighted, undirected graphs, we define contracts for describing several of the arguments and results involved.

- A vertex is represented as a natural number:

  ```
  let Vertex? = nat?
  ```

- We use singly-linked lists of vertices, made out of a `cons` struct with `data` and `next` fields as provided by the `cons` library (`import cons`):

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

  The `Cons.ListC` contract optionally takes a contract for the list elements, which lets us express that a `VertexList?` is indeed a linked list of `Vertex?`es.

- A weight is a real number (i.e., not an infinity or "Not a Number"), and an optional weight is either a weight or `None`:

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

  Note: this differs slightly from the interfaces we have seen in class.

- A weighted edge is represented by a struct containing two vertices and a weight; we use lists of those as well:

  ```
  struct WEdge:
      let u: Vertex?
      let v: Vertex?
      let w: Weight?

  let WEdgeList? = Cons.ListC[WEdge?]
  ```

> Note that `WEdge` is used in the result of one of the graph methods (below), but you don't have to use it internally in your graph representation.

Now we can give our signature for weighted, undirected graphs as a DSSL2 interface with five operations:

```
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?
```

The operations behave as follows:

- The `len` method returns the number of vertices in the graph, that is, n.
- The `set_edge` method adds an edge of weight `w` between vertices `v` and `u` when `w` is a number; if the edge already exists, its weight is updated to `w`. If `w` is `None` then the edge, if it exists, is removed, and if absent remains absent.

  Note that because the edges of undirected graphs are symmetric, the order of `u` and `v` mustn't matter; this implies that `set_edge` must maintain an invariant.
- The `get_edge` method returns the weight of the edge between vertices `u` and `v` if it exists, or `None` if it does not.
- The `get_adjacent` method returns a list of all vertices that are directly connected to vertex `v`. The order of the list is unspecified.[1]
- The `get_all_edges` method returns a list of all edges in the graph, in unspecified order. For each edge in the graph, it includes only one direction in the list. For example, if a graph has an edge of weight 10 between vertices 1 and 3, then the resulting list will contain either `WEdge(1, 3, 10)` or `WEdge(3, 1, 10)`, but not both.

Overall, this interface differs slightly from the ones we have seen in lecture for UU, UD, and WD graphs, both in terms of supported operations and operation behavior. This is intentional: this way, you get to work with multiple graph variants. You are likely to encounter still more variants in practice!

[1]This means any order you like.

Your task

Representation

Your job is to implement the `WUGraph` class, which must satisfy the `WUGRAPH` interface. To do so, you must choose a representation, as either an adjacency matrix or adjacency lists. Whichever you choose, you will need to add some field(s) to the `WUGraph` class and fill in the `__init__` method to initialize them.

1. Define the field(s) for your representation at the top of the `WUGraph` class.
2. Complete the definition of the `__init__` method. The `WUGraph` constructor takes one natural number argument, which is the number of vertices desired in the new graph.

Note: the representation you use internally doesn't need to correspond to the return types of the ADT operations! You can use whatever kinds of boxes you want, so long as what you create is either an adjacency matrix or adjacency lists. So long as your operations translate from your internal representation to the types expected by the interface, it's all good.

Graph operations

Once you've defined your graph representation, you will have to implement the five graph API methods as specified by the `WUGRAPH` interface. Their required time complexities depend on your choice of representation.

Adjacency matrix representation

3. Implement the `len` method, which must be $\mathcal{O}(1)$ time.
4. Implement the `set_edge` method, which must be $\mathcal{O}(1)$ time.
5. Implement the `get_edge` method, which must be $\mathcal{O}(1)$ time.
6. Implement the `get_adjacent` method, which must be $\mathcal{O}(V)$ time.
7. Implement the `get_all_edges` method, which must be $\mathcal{O}(V^2)$ time.

Adjacency lists representation

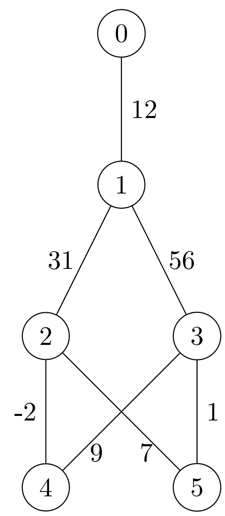
The running times of several adjacency list operations depend on d, the degree of the graph.

3. Implement the `len` method, which must be $\mathcal{O}(1)$ time.
4. Implement the `set_edge` method, which must be $\mathcal{O}(d)$ time.
5. Implement the `get_edge` method, which must be $\mathcal{O}(d)$ time.
6. Implement the `get_adjacent` method, which must be $\mathcal{O}(d)$ time.
7. Implement the `get_all_edges` method, which must be $\mathcal{O}(V + E)$ time.

Building graphs

The next part of your task is to use the graph class that you built to construct two example graphs: one fixed, and one of your choice. You must construct both these graphs using only methods from the `WUGRAPH` interface. Of course, you're welcome to build more graphs for testing as well!

First, complete the `example_graph` function so it builds the following graph:



8. Complete the `example_graph` function.

Second, complete the `my_neck_of_the_woods` function to build a graph representing your hometown and neighboring towns/cities (nodes), road connections between them (edges), and distances between connected cities (weights). Include at least five cities.

Then to associate nodes to cities, build a bidirectional mapping consisting of two dictionaries, one for each direction. Please use the most efficient dictionary possible for each direction.

Dependencies

To make sure everyone has access to working dictionary implementations, we have provided you with a compiled solution for homework 3 in `hw4-lib.zip`.

Extract `hw4-lib.zip` *in the same directory* as your `graph.rkt` file, and the `import` statement at the top will import all the definitions from homework 3, with the interface described in the homework 3 handout.

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

Note: The compiled files we provide are specific to Racket version 8.12; if you use any other version, they will not work.

You are welcome to use these dictionaries, or use some of your own creation. If you use one of your own, though, be sure to include its code in your submission; your file must be self-contained (aside from `hw4-lib` and the DSSL2 standard library, of course).

Your `my_neck_of_the_woods` function must return a `CityMap` struct which combines the graph and the two dictionaries. With this bidirectional mapping, we can now do operations in terms of cities, as opposed to having to use nodes directly! Write (at least!) one test that uses the graph and both directions of the bidirectional mapping together to ask a cities-related question of your choice.

In case you would prefer not to share your own neck of the woods, you can use my hometown (and related cities) instead:

- **Bellaire**: where I'm from
- **Houston**: where my sister lives
- **San Antonio**: where my cousin lives
- **Huntsville**: where one of my aunts lives
- **Los Angeles**: where my grandparents used to live
- **Baton Rouge**: where one of my grandfathers used to work

For connections and distances, you can consult a map.

9. Complete the `my_neck_of_the_woods` function.
10. Write a test case that uses all the parts of `my_neck_of_the_woods`.

Depth-first search

Once you have your graph implementation working, there's one more thing to implement, a pre-order depth-first search function to traverse a graph:

```
dfs : WUGRAPH Vertex [Vertex -> None] -> None
```

This function takes a graph `g`, a vertex `u`, and a visitor function `f`. It performs a depth-first search starting at `u`. As it encounters each vertex `v` for the first time, it calls `f(v)`. The visitor function is called on each reachable vertex exactly once, in a valid depth-first order.

10. Implement the `dfs` function, which must have the optimal asymptotic time complexity: $\mathcal{O}(V + E)$ if using adjacency lists, or $\mathcal{O}(V^2)$ if using an adjacency matrix.

Keep in mind: your `dfs` function must operate correctly on *any* conforming implementation of the `WUGRAPH` interface; not just yours. So be sure to use only methods which are part of the interface.

In order to help you test `dfs`, we have provided a function `dfs_to_list` that uses it to construct a list of vertices in DFS-order. It should be relatively easy to write `assert` tests for `dfs_to_list` once you know in what order your `dfs` function visits vertices. You're welcome to use the graphs you built earlier to test your DFS and/or create new ones.

The starter code also includes functions `sort_vertices` and `sort_edges`, which sort lists of vertices and `WEdge`s, respectively. This is useful for testing because several methods produce lists in an unspecified order.

Grading

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

- a definition for the `WUGraph` class,
- definitions for the functions `example_graph`, `my_neck_of_the_woods`, and `dfs`,
- sufficient tests to be confident of your code's correctness,
- and the honor code.

Be sure to remove any leftover debugging printing code, comment out any code that would cause infinite loops, and make sure that your submission can run successfully. If your submission produces excessive output, loops infinitely, or crashes, we will not be able to give either you feedback on it or credit for it. **Please run your code one last time before submitting!**

Functional Correctness

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

- **Basic WU graph**: construction, `get_edge`, `get_adjacent`, and `set_edge` in the non-update and non-delete case.
- **Advanced WU graph**: `set_edge` update and delete cases, `get_all_edges`.
- **Basic search**: correct DFS traversal order on connected graphs.
- **Advanced search**: DFS on disconnected graphs, and stress tests on large random graphs (to check for computational complexity issues).

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 four test suites.
- **Almost there**: passes both basic test suites, and fails a single advanced test suite.
- **On the way**: either passes both basic test suites, or passes both the basic and advanced test suite for one of the two topics.
- **Not yet**: does not achieve "on the way" requirements.
- **Cannot assess**: we could not successfully grade your submission.
- **Missing honor code**: your submission did not include the honor code.

Non-Functional Correctness

For this assignment, the self-evaluation will be specifically looking for:

- Thorough testing, including edge cases
- Efficiency from the use of the correct representation and operations
- Good data representation choices