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 implement a depth-first search.

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

Orientation

The graph for this assignment is a weighted, undirected graph 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 `car` and `cdr` 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 element, which lets us express that a `VertexList?` is indeed a linked list of `Vertex?`es.

- A weight is a real 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)
  ```

- 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 WU_GRAPH:
    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.

Your task

Representation

Your job is to implement the `WuGraph` class, which must satisfy the `WU_GRAPH` 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.

[1] This means any order you like.

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.

Graph operations

Once you've defined your graph representation, you will have to implement the five graph API methods as specified by the `WU_GRAPH` 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

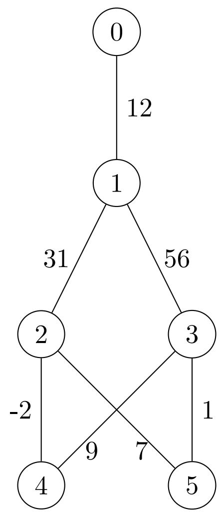
The running times of several adjacency list operations depend on d, the degree of the graph.

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. 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.

To associate nodes to cities, build a direct-addressing-based dictionary mapping node numbers to city names. The `my_neck_of_the_woods` function must return a `CityMap` struct which combines the graph and the dictionary.

In case you would prefer not to share, you can use my hometown instead:

- **Montreal**: where I'm from
- **Laval**: where my sister lives
- **Repentigny**: where my brother-in-law is from
- **Terrebonne**: where one of my aunts lives
- **Potton**: where another one of my aunts lives
- **Saint-Charles-sur-Richelieu**: where my grandparents used to live

For connections and distances, you can consult a map.

9. Complete the `my_neck_of_the_woods` function.

You must construct both these graphs using only methods from the `WU_GRAPH` interface.

Depth-first search

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

```
dfs : WU_GRAPH 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 `WU_GRAPH` 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.

Honor code

Every homework assignment you hand in must begin with the following definition (taken from the Provost's website[2]; see that for a more detailed explanation of these points):

```
let eight_principles = ["Know your rights.",
     "Acknowledge your sources.",
     "Protect your work.",
     "Avoid suspicion.",
     "Do your own work.",
     "Never falsify a record or permit another person to do so.",
     "Never fabricate data, citations, or experimental results.",
     "Always tell the truth when discussing your work with your instructor."]
```

If the definition is not present, you will receive no credit for the assignment.

Note: Be careful about formatting the above in your source code! Depending on your pdf reader, directly copy-pasting may not yield valid DSSL2 formatting. To avoid surprises, be sure to test your code *after* copying the above definition.

[2] http://www.northwestern.edu/provost/students/integrity/rules.html

Deliverables

Your completed `graph.rkt`, containing

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

Your code will be evaluated for correctness, resource efficiency, thoroughness, code reuse, and style.

Submission

Your homework must be submitted via Canvas.