

HW4: Graph

Due: Thursday, November 8, at 11:59 PM, via GSC

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

For this assignment you will implement an API for weighted, undirected graphs; then you will use this API to implement a simple depth-first search.

In `graph.rkt`¹ I've supplied headers for the methods and function that you'll need to write, along with a few suggested helpers and some code to help 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 $0, 1, \dots, n - 1$. This makes it easy to store information about vertices in a vector of size n .

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

- A vertex is represented as a natural number:

```
let Vertex? = nat?
```
- We use singly-linked lists of vertices:

```
let VertexList? = Cons.ListC[Vertex?]
```
- A weight is a number, and an optional weight is either a weight or `False`:

```
let Weight? = num?  
let OptWeight? = OrC(Weight?, False)
```
- A weighted edge is represented by a struct containing two vertices and a weight; we use lists of those as well:

¹<https://goo.gl/9n42HB>

```

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 is not intended for use 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?) -> VoidC
  def get_edge(self, u: Vertex?, v: Vertex?) -> OptWeight?
  def get_adjacent(self, v: Vertex?) -> VertexList?
  def get_all_edges(self) -> WEdgeList?

```

The behavior of the operations is as follows:

- The `len` method returns the number of vertices in the graph, that is, n .
- The `set_edge` method adds an edge between vertices `v` and `u` of weight `w`, when `w` is a number; if the edge already exists, it updates the weight to `w`. If `w` is `False` then the edge, if it exists, is removed.

Note that because the edges of undirected graphs are symmetric, the order of `u` and `v` cannot 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 `False` 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.
- 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.

One easy way to avoid redundant edges is to only add an edge `e` to the list when `e.u <= e.v`.

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.
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 $\mathcal{O}(E)$ time.

Depth-first search

Once you have your graph implementation working, you must implement a depth-first search function:

```
dfs : WU_GRAPH Vertex [Vertex -> Void] -> Void
```

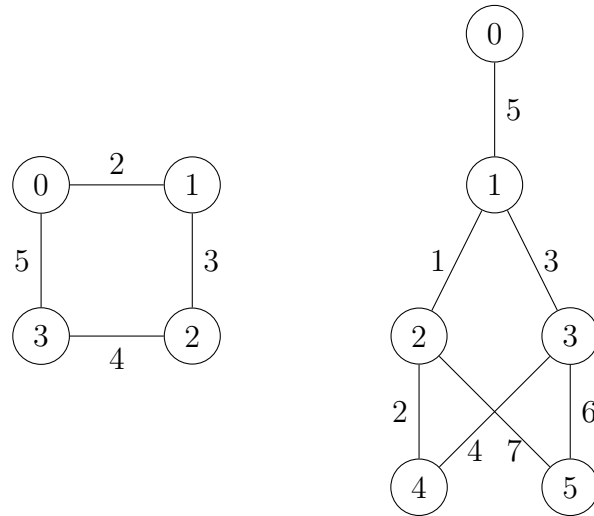
This function takes a graph g , a vertex v , and a visitor function f . It performs a depth-first search starting at v . As it encounters each vertex u for the first time, it calls $f(u)$. The visitor function is called on each reachable vertex exactly once, in a valid depth-first order.

8. 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.

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_eq` tests for `dfs_to_list` once you know in what order your `dfs` function visits vertices.

Testing

To facilitate testing, we have provided you two example graphs. Function `EX_GRAPH1` returns the four-vertex graph on the left, function `EX_GRAPH2` returns the six-vertex graph on the right:



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

Deliverables

The provided file `graph.rkt`, containing

- working definition of the `WuGraph` class,
- a working definition of the `dfs` function, and
- sufficient tests to be confident of your code's correctness.