HW3: Graphs

Due: Thursday, October 26, 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 functions that you'll need to write, along with a few suggested helpers and some code to help with testing.

Your task

Representation

First you will need to define your representation, the `WUGraph` data type. A `WUGraph` represents a weighted, unordered graph, where vertices are identified by consecutive natural numbers from 0, and weights are arbitrary numbers:

```
# A Vertex is a Natural
# A Weight is a Number
```

The API also uses a data type for weights that includes `False` to indicate the absence of an edge:

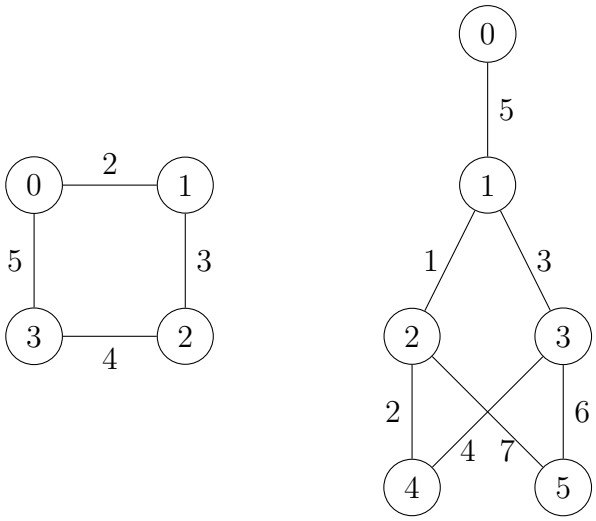
```
# A MaybeWeight is one of:
# - Weight
# - False
```

Your graph representation is up to you—you may use either adjacency lists or an adjacency matrix.

Graph examples

To ensure your representation is adequate and to facilitate testing, you must define two example graphs. Define `GRAPH1` to be the four-vertex graph on the left, and define `GRAPH2` to be the six-vertex graph on the right:

[1] `http://goo.gl/Yhy1G2`



Graph operations

Once you've defined your graph representation, you will have to implement five functions for working with graphs:

```
make_graph   : Natural -> WUGraph
set_edge!    : WUGraph Vertex Vertex MaybeWeight -> Void
graph_size   : WUGraph -> Natural
get_edge     : WUGraph Vertex Vertex -> MaybeWeight
get_adjacent : WUGraph Vertex -> ListOf[Vertex]
```

To construct a graph, we would start with `make_graph(n)`, which returns a new graph having `n` vertices and no edges. Then we add edges using `set_edge!(g, u, v, w)`, which connects vertices `u` and `v` by an edge having weight `w`. The weight `w` may be an actual numeric weight or it may be `False`, which effectively removes the edge.

`graph_size(g)` returns the number of vertices in `g`, which is the same as the number originally passed to `make_graph` to create the graph. `get_edge(g, u, v)` returns the weight of the edge from `u` to `v`, which will be `False` if there is no such edge. Note that because the graph is undirected, `get_edge(g, u, v)` should always be the same as `get_edge(g, v, u)`.[2] Finally `get_adjacent(g, v)`

[2]This probably means that `set_edge!` needs to maintain an invariant.

returns a list of all the vertices adjacent to `v` in graph `g`—note that an undirected graph does not distinguish predecessors from successors.

Depth-first search

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

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

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

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.

Deliverables

The provided file `graph.rkt`, containing

- working definitions of your `WUGraph` data type and its operations,
- definitions of the example graphs `GRAPH1` and `GRAPH2`,
- a working definition of the `dfs` function, and
- sufficient tests to be confident of your code's correctness.