Graphs

Author: Andrew Shallue, Fall 2024, for CS229 (Data Structures)

General Introduction

Graphs are important objects in computer science, because they allow us to model more complicated discrete relationships. The essence of a graph is that it has a set of vertices, V, and a set of edges E. A given edge in E connects one or more vertices.

This general definition gets complicated fast because there are a host of reasonable assumptions and restrictions we might make. A good starting point is wikipedia. For this lab, we will focus on the following terms:

- All our graphs will be simple. This means that an edge connects exactly two vertices (so it doesn't connect one vertex to itself), and each pair of vertices can have at most one edge connecting them.
- Our graphs will be either undirected or directed, and either weighted or unweighted. See notes or wikipedia for definitions of these terms.
- In code, an edge set can be represented either densely or sparsely. For this lab, the Graph class will utilize a sparse representation for simple, undirected, unweighted graphs. The DenseGraph class will utilize a dense representation for simple, directed, weighted graphs.

Overall tip: sometimes it helps to work with one row of a 2D vector. Here is an example, where graph is of type `vector<vector<int>>`:

```
for(int i = 0; i < graph.size(); ++i){
      vector<int>& row = graph.at(i);
      // do something with row
}
```

It is important to have `row` be a reference, so that changes to `row` will impact the overall 2D object, rather than modifying a copy of the row.

Part 1

Instructions

1. Create files `Lab10Part1.h` and `Lab10Part1.cpp`. The header file will have two functions: `positions` and `lab10_part1_main()`. Most of the work will be in `lab10_part1_main()`.
    a. `Done.`

2. Setup a 2D vector of `int` type. This object will have type `vector<vector<int>>`.
    a. Check.

3. I want the 2D vector to look like this:
```
1 2 3 4 5
2 3 4 5 0
3 4 5 0 1
4 5 0 1 2
5 0 1 2 3
0 1 2 3 4
```
    a. First try it through explicit assignment. Here's a small example:
    `graph = {{0, 1, 2}, {1, 2, 0}, {2, 1, 0}}.`
        Check.
    Try to expand that example to the larger 2D vector requested.
    b. Clear the vector and try it again, this time with a double for loop. Hint: row `i` takes the numbers `[1, 2, 3, 4, 5]`, adds `i`, then reduces modulo `6`, and excludes `i` itself from the list. The outer loop performs an action `6` times. The inner loop will create a `vector<int>` object called `row`, use a loop to create and append the appropriate integers, then append the row to the `vector<vector<int>>` object.
        i. Checkedy check.

4. Using a double loop, print the entries of your 2D vector to confirm it is constructed correctly.
    a. All done.

5. Write a function with signature `vector<vector<int>::iterator> positions(int n, vector<vector<int>>& g)`. For the given input integer `n`, return a vector of iterators, which has the same size as `g`. Each iterator points at a position in one of the vectors that holds `n`. I encourage you to use the `find` function from the `algorithm` package. Question: what does `find` return when `n` is not in the row?
    We did it.

6. Use your positions function to remove all the 3's from the 2D vector. I want you to use the `vector::erase` function that takes an iterator as a parameter. Then print the 2D vector again to ensure the removal worked correctly.
    a. Watch out! One of the rows doesn't have a 3. What will `find` return, and hence what iterator is in that position in the `positions` output? If you try to erase, it will cause undefined behavior, so only call `erase` on an iterator where a 3 was actually found.
        i. Complete.

7. In Part 2 we will use `pair<int, int>` to represent an edge, and in Part 3 `tuple<int, int, double>` to represent a weighted edge. One can access the elements of pair `p` with `p.first` and `p.second`. One can access the index `i` piece of a tuple `t` with `std::get<i>(t)`. As an example of creation, here is a tuple representing a directed edge from 0 to 1 with weight 21: `tuple<int, int, double> n1{0, 1, 21}`. Note: you need to add `pair` and `tuple` to your namespace, or else call them with `std::pair` and `std::tuple`.
    a. Create unweighted edges using `pair` that correspond to the path `0 - 5 - 2 - 0`
    b. Create weighted edges using `tuple` that correspond to the path `0 -> 5 -> 2 -> 0`, with weights of your own choosing.
    c. For each of those two paths, create a vector and store all the edges in that vector.
    d. For each of the two vectors, loop over the vector, accessing each edge and printing a message with its information. For example "edge from 0 to 1 with weight 21."
        i. All done.

Part 2 - Graph

Introduction

This class will use a sparse representation to instantiate unweighted, undirected graphs. The class will have two data members, `num_vertices_` and `edges_`. The `edges_` will have type `vector<vector<int>>`, with the integer in position (i, j) representing a vertex that is a neighbor to i. For example, if the graph has a row that looks like

```
0: 1 5
```

Then this means that vertex 0 is neighbors with vertex 1 and vertex 5. A vertex should never appear multiple times in one row, because there is at most one edge connecting two vertices. And a vertex should never appear as a neighbor in its own row.

Instructions

1. The `Graph` class will have two private data members: `int num_vertices_` and `vector<vector<int>> edges_.`
    a. `Done.`

2. The default constructor will set `num_vertices_` to 0 and will leave `edges_` empty.
    a. All done.

3. The non default constructor will set `num_vertices_` to the parameter `n.` For `edges_,` push `n` empty vectors onto the outer vector, so that `edges_` has size `n,` but each vector in `edges_` has size 0.
    a. Complete.

4. Write a `print()` function that prints the edge sets. Each line corresponds to one row of the graph. Print the vertex followed by a colon, followed by neighbors, space separated. For example, if `0` is neighbors with `2, 4, 5` print

    ```
    0: 2 4 5
    Check.
    ```

5. The `make_complete()` function will reset the graph to be a complete graph. Clear all the existing edges, then make sure to add every possible edge to every possible row.
    a. Completed.

6. The `make_cycle()` function will reset the graph to be a cycle. A cycle has edges (0, 1), (1, 2), (2, 3), ... (n-1, n), (n, 0). The graph should be undirected, so make sure that for an edge like (0, 1), you add 1 to the 0 row and add 0 to the 1 row.
    a. All complete.

7. The `add_edges(...)` function takes a parameter of type `vector<pair<int, int>>&.` This is a vector of pairs. Each pair represents an edge. Add all the edges in the parameter vector to the graph. Be careful - if the edge already exists we don't want to double up. I want you to use the find function to determine whether a given edge already exists or not.
    a. Check.

8. The `add_vertex(...)` function takes a single parameter of type `vector<int>&.` Add a new vertex to the graph. To do this, add one to `num_vertices_,` add a new row to `edges_,` and add edges so that the new vertex is connected to all the vertices given in the parameter list.
    a. Did it.

9. The `disconnect_vertex(int v)` function removes all edges connecting vertex v to the graph. The vertex v is still part of the graph, I just want you to remove edges. I want you to use the `find` function to find v in every row. If v is in a row, use the `erase` function to remove it.
    a. We did it.

10. The `delete_edges(...)` function takes a single parameter of type `vector<pair<int, int>>&`. The parameter represents a vector of edges. I want you to remove those edges from the graph. Once again, use `find` and `erase`. If a given edge is not in the graph, ignore it.
    a. Done.

11. The `max_degree()` function returns the maximum degree among all vertices.
    a. All done.

12. The `is_adjacent(int v1, int v2)` function returns bool. Return `true` if the edge
`(v1, v2)` is in the graph, `false` otherwise.
    Complete.

13. The `is_two_path(int v1, int v2)` function returns bool. Return true if there is a path of length 1 or 2 that connects `v1` to `v2`. I recommend looping over all neighbors of `v1` and using the `is_adjacent` function.
    a. Check.

Part 3 - DenseGraph

Introduction

The `DenseGraph` class will use a dense representation, and will implement directed, weighted graphs. The data members will be `int num_vertices_` and `vector<vector<double>> edges_`. This time the number in the `(i, j)` entry will be 0 if there is not an edge `i -> j`, and a non-zero value representing the weight if there is an edge `i -> j`. For example,

```
0: 0 0 2.4 0 0.7 1.2
```

means that 0 has out-degree 3, with `0 -> 2` of weight 2.4, `0 -> 4` of weight 0.7, and `0 -> 5` of weight 1.2. There is no edge from 0 to any of 0, 1, 3.

Instructions

For this class I will give you the header file. Implement all functions found therein. I think this is a little easier than `Graph`, because you always have a 2D vector for `edges_` which is of size n-by-n, so one can always do `edges_.at(i).at(j)`.

`Completed.`