COSC 320 - Advanced Data Structures and Algorithm Analysis
Lab 10

Dr. Joe Anderson

Due: 21 November 2019

1 Objectives

In this lab you will focus on the following objectives:

1. Review basic graph representations and operations
2. Develop familiarity with the `c++` standard library tools.
3. Solve the problems of implementing abstract algorithms with programming languages.

2 Preliminaries

2.1 Graph Representations

A *Graph* is a pair of sets, $G = (V, E)$ where V is a finite set of objects which we call the "vertices", and $E \subset V \times V$, which we refer to as "edges". If we consider the edges to be symmetric, i.e. we equate the meaning of $(u, v) \in E$ and $(v, u) \in E$, we call the graph "undirected" and otherwise it is "directed." Consider the following example of an undirected graph:

$$V = \{a, b, c, d, e, f\}$$
$$E = \{(a, c), (b, d), (a, d), (c, e), (a, f), (f, b), (c, b), (e, a)\}$$

We can illustrate this configuration of vertices and edges in Figure 1;

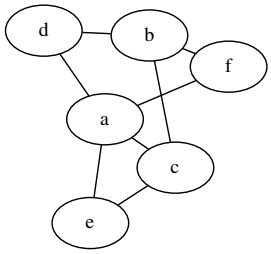

Figure 1: Graph Example

Naturally, we must ask ourselves the question of how to represent this structure in a way that is amenable to an algorithm. For instance, what interface would be needed with the graph object to run an algorithm

to find a path between two vertices? It generally does not make sense to keep the graph as a purely string representation, as in the definition. Instead, we typically opt for either the **adjacency matrix** A_G for graph G, which is a matrix of dimensions $|V| \times |V|$, defined by

$$(A_G)_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

This has many applications, and many linear algebra tools lend themselves to graph algorithms.

However, a slightly more natural way is with an **adjacency list**. This can be viewed as an array, with one element per vertex; at each array cell is the head to a linked list of all the neighbors of that vertex. So in the above example, in the array cell corresponding to "a" would be the list of vertices "d, f, c, e". In this way, we can see that vertices can appear many times in the list, but in general this is a little more space efficient than storing the entire matrix. However, to test of two vertices are neighbors, we would have to perform a linear search, meaning we pay a price in computational demand, where looking at a single matrix cell would be done in constant time. For this lab, you will work with the adjacency list representation of graphs, using tools built into the `c++` standard library.

2.2 Using the standard library structures

In this lab, you will use aspects of the `c++` standard library. To do this effectively, you will need to frequently refer to the documentation, easily found at:

- `vector` - `http://en.cppreference.com/w/cpp/container/vector`
- `queue` - `http://en.cppreference.com/w/cpp/container/queue`
- `map` - `http://en.cppreference.com/w/cpp/container/map`

Each of these behaves very closely to the structures studied during the course, with the possible exception of `map`. For this structure, you can think of it as a generalization of an array, where the indices are no longer necessarily integers, but any data type. An implication of this is that in your map of vertices, `vertices[3]` may have a value, but `vertices[2]` may not. Moreover, the use of `vertices[-1]` makes sense in the context of the map, but not in the context of our graphs.

A second note is that the `vector` structure takes the place of linked lists. Although there is a `std::list`, `vector` is more commonly used due to its performance characteristics.

2.3 Breadth-first search (BFS)

- Main idea:
  - Start at some source vertex s and visit,
  - All vertices at distance 1,
  - Followed by all vertices at distance 2,
  - Followed by all vertices at distance 3,
  - $\vdots$
- BFS corresponds to computing *shortest path* distance (number of edges) from s to all other vertices.
- To control progress of our BFS algorithm, we think about *coloring* each vertex
  - *White* before we start,
  - *Gray* after we visit the vertex but before we have visited all its adjacent vertices,

  - *Black* after we have visited the vertex and all its adjacent vertices (all adjacent vertices are gray).

- We use a queue Q to hold all gray vertices—vertices we have seen but are still not done with.
- We remember from which vertex a given vertex v is colored gray – i.e. the node that discovered v first; this is called parent$[v]$.
- Algorithm:

```
BFS(s)
    color[s] = gray
    d[s] = 0
    ENQUEUE(Q, s)
    WHILE Q not empty DO
        DEQUEUE(Q, u)
        FOR (u, v) ∈ E DO
            IF color[v] = white THEN
                color[v] = gray
                d[v] = d[u] + 1
                parent[v] = u
                ENQUEUE(Q, v)
            END IF
            color[u] = black
        END FOR
    END WHILE
```

- Algorithm runs in $O(|V| + |E|)$ time

3 Tasks

1. Put your code in a folder to be zipped and turned in at the end.
2. Write a class called `Graph` that will be used to represent an undirected graph in code. Use the following basic structure:

```
class Graph{
  private:
    /* will map an int to a list of its neighbors */
    std::map<int, std::vector<int>> vetices;

  public:
    addVertex(int);   // add a vertex to the graph
    addEdge(int,int); // add an undirected edge to the graph
    print();          // prints the adjacency list of each vertex, to show the structure
    printBfs(int);    // prints the vertices discovered by a BFS, starting at a given vertex
};
```

   (a) Nodes in the graph will be identified as integers.

(b) The `addVertex` should check to make sure a duplicate node isn't entered, and then add it to the map, `vertices`, with an empty vector of neighbors.

(c) The `addEdge` should add an edge between two vertices (if it doesn't already exist), and since this is an undirected graph, will need to add it to *both* adjacency lists.

(d) The `printBfs` should execute a version of the BFS algorithm discussed in class.

i. Instead of assigning a color to each node (possible but cumbersome, with modification to the graph representation), use a `std::map` to store the colors, parents, and distances of each vertex (as in the algorithm above). For example, the map for colors might be defined as: `std::map<int, color_t>` where `color_t` is an enum to hold values for white, black and gray.

ii. Use your Heap Queue from an earlier lab or the `std::queue<int>` data structure to control the exploration of the graph.

3. Write a test program to demonstrate (clearly) the correctness of each of the above functions.

4. Include a `Makefile` to build your code.

5. Include a `README` file to document your code, any interesting design choices you made, and **answer the following questions completely and thoroughly**:

(a) Summarize your approach to the problem, and how your code addresses the abstractions needed.

(b) What is the theoretical time complexity of your algorithms (best and worst case), in terms of the size of the tree? Be sure to vary the parameters enough to use the observations to answer the next questions!

(c) How could the code be improved in terms of usability, efficiency, and robustness?

4 Submission

All submitted labs must compile with your provided `Makefile` and run on the COSC Linux environment.

Upload your project files to MyClasses in a single `.zip` file.

Turn in (stapled) printouts of your source code, properly commented and formatted with your name, course number, and complete description of the code.

Also turn in printouts reflecting several different runs of your program (you can copy/past from the terminal output window). Be sure to test different situations, show how the program handles erroneous input and different edge cases.