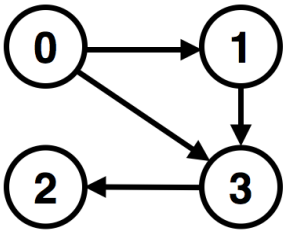
Graph Traversal Pre-lab

1. Introduction

In this lab, we will be traversing graphs, which are similar to trees in that they have connections between elements (called *vertices* with *edges*). This pre-lab will review graphs and describe the method we will be using to store graphs (adjacency matrices). We will also describe traversing graphs, which you will have to code during lab.

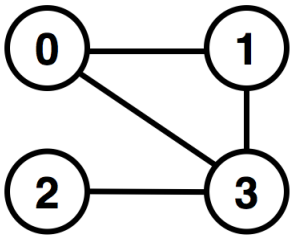
2. The Adjacency Matrix representation of a graph.

An "adjacency matrix" is one of the most simple ways to store a graph. An adjacency matrix is a 2-dimensional representation where each table entry is a boolean value that represents whether two vertices have an edge between them. The edges are directed — if a particular vertex at (row,col) is **`true`**, then the vertex at **`row`** has a directed edge to the vertex at **`col`**. For example, the following graph has the adjacency matrix shown below:



	0	1	2	3
0	false	true	false	true
1	false	false	false	true
2	false	false	false	false
3	false	false	true	false

For this lab, the graphs will be undirected, meaning that for every (row,col) that is marked true in the adjacency matrix, the corresponding (col,row) is also marked. E.g., the table below represents the following undirected graph:



	0	1	2	3
0	false	true	false	true
1	true	false	false	true
2	false	false	false	true
3	true	true	true	false

In a 2-dimensional array with the adjacency matrix variable **`adjacencyMatrix`**, a particular (row,col) entry is accessed by **`adjacencyMatrix[row][col]`**.

For this lab, we have created functions that create abstractions into **adjacencyMatrix** for you. The **graph.h** header file has the following functions:

```
bool isAdjacent(int u, int v);

bool markVertex(int u);

bool isMarked(int u);

void unmarkAll();
```

The functions take either two (**isAdjacent()**), one (**markVertex()** and **isMarked()**), or zero (**unmarkAll()**) vertices (ranging from **0** to the global variable **NODE_COUNT**), and the functions handle all adjacency matrix calculations. For example, to determine if two vertices 0 and 1 are adjacent, simply check the return value of **isAdjacent(0,1)**.

3. Graph Traversal Algorithms (DFS and BFS)

Recall that there were four types of traversals for binary search trees: *in-order*, *post-order*, *pre-order*, and *level-order*. The first three traversals are done via a “depth first search,” (DFS) meaning that they traverse “deep” into one branch of the tree, then deep into another branch of the tree, until all branches have been traversed. Level-order traversal (whereby each level is traversed from left-to-right, then the next level is traversed) is done via “breadth first search” (BFS). In a BFS, the “breadth” (or width) of the tree is searched first.

Graphs have analogous strategies for traversal: you can perform a DFS or a BFS on a graph. You cannot traverse a graph in any particular order, because graphs do not have an inherent ordering like binary search trees do. So, we are limited to either BFS in general, or DFS in general.

Because graphs are less structured than trees, we may end up visiting a graph vertex multiple times before we can consider the graph completely traversed. Therefore, we must mark the graph vertices as we “complete” them (meaning that they have been visited and completed).

BFS

A graph BFS algorithm searches locally around a particular vertex before traversing to vertices further away in the graph. Because of the nature of a BFS, it is not possible (without incurring extra time cost) to perform the algorithm recursively. We can, however, use a queue to perform the search. The algorithm is as follows:

```
unmark all,
choose x
mark and process (print) x add x to the Q
while Q not empty
      remove vertex u from Q
      for all unmarked neighbors w of u
            mark and process w
            insert w into Q
```

DFS

A graph DFS algorithm searches deep into the graph instead of locally. You only come back to the original vertex once a particular adjacent vertex has been completely searched. The nature of a DFS is such that you can perform the algorithm

recursively (or with a stack). Interestingly, the other adjacent vertices of a particular vertex can get searched and marked as complete before the recursive algorithm returns (unlike in a tree traversal). The algorithm is as follows:

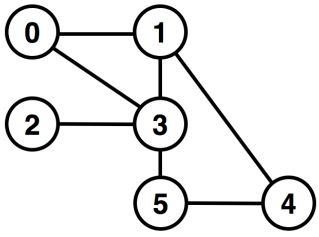
```
unmark all vertices

choose x
mark and process (print) x

for each unmarked neighbor of x, w
      recurse on w
```

4. Questions

1. Why is it necessary to unmark all vertices before starting both algorithms?

2. How would you mark vertex 5 using one of the functions provided in graph.h?

3. Perform a by-hand BFS on the following graph, and report the output, starting from vertex 0 (there are different answers that are correct!)



4. Perform a by-hand DFS on the same graph from question (3). There are different answers that are correct!

5. What kind of symmetry would the adjacency matrix of the graph from questions (3) and (4) have?