Project 2
Applications of Graph Search*

Due: Midnight, Monday, 12 Nov

In this homework, you will adapt the breadth- and depth-first search to determine properties of graphs, implementing two or three applications of them. Start from the Lua files in `http://web.cs.wpi.edu/~cs2223/b12/proj/proj2.zip`. Run your first two procedures on the undirected graphs in the file `undirected.lua` and run the third procedure on the directed graphs in `directed.lua`. Include the results in a block comment in your `turnin` submission.

For full points (100 pts.), do either of the BFS problems (Connected Components or Bipartite Graphs) and also the DFS one (Finding a Cycle). For 20 points extra credit, do all three. Doing a second BFS problem will be easier once you have the first one working.

The files `bfs.lua` and `dfs.lua` contain the breadth-first search and depth-first search code shown in class. Files `util.lua`, `stacks.lua`, and `queues.lua` contain utility functions and the implementations of the *stacks* and *queues* datatype. The file `graphs.lua` defines the graph datatype. Reference their contents using the `module.function` notation, after doing `require "module"`.

There are a few directed and undirected graphs in `directed_graphs.lua` and `undirected_graphs.lua`. Use them to test all your code thoroughly. You can also build randomly generated graphs using the functions at the end of `graphs.lua`.

Write your code in a file named `my_proj2.lua`, and do not change the files in the starter package. Use comments to say which of your functions is the answer to which problem below, and to explain how all your code works. I expect that you will copy and modify the code from `bfs.lua` and `dfs.lua`. Try to modify the key search procedures cleanly, so that it's very easy to see what you're adding for each of the tasks. Test all your code.

Connected Components. An undirected graph $G = (V, E)$ with vertices V and edges E divides naturally into *connected components*. A set of vertices $C \subseteq V$ is a connected component if:

1. for all $x, y \in C$, there is a path using edges in E between x and y; and

*Joshua Guttman, FL 137, `mailto:guttman@wpi.edu`. Include [cs2223] in the Subject: header of email messages. Due midnight at the end of Monday, 12 Nov.

2. for all $x \in C$ and $y \notin C$, there is no path using edges in E from x to y, and no path from y to x.

Breadth-first search, starting from a vertex s, will discover the vertices in the same connected component as s. Program a routine that will use BFS repeatedly to identify all of the connected components of a graph. Return an array a with one entry for each component C. The entry for each component C should be an array containing the vertices in C. Thus, if the graph has three components, one containing nodes 1, 3, 5, a second containing nodes 2, 4, 6, and the third containing 7, 8, you want to return an array such as:

```
{  { 1, 3, 5 },
   { 2, 4, 6 },
   { 7, 8 }   }
```

The order of the components doesn't matter, and the order of the members within any component doesn't matter. Print out the members of each component separately.

Bipartite Graphs. An undirected graph G is *bipartite* if we can split its vertices into two disjoint sets A, B ("partition" the vertices) such that every edge crosses between the two parts. No edge should connect two vertices in A to each other, or connect two vertices in B to each other. Each one should connect a vertex in A to one in B.

For instance, consider an undirected graph where the vertices are either people or corporations, and there's an edge between v_1 and v_2 if one is a person and the other is a corporation, and the corporation employs the person. This graph is bipartite, because persons and corporations are disjoint. The graph may be quite complex, though. The same person can work for more than one corporation, and the same corporation will employ many persons. So paths in this graph may traverse many different corporations.

The obvious approach would be to generate all the partitions and check each one to see if it satisfies the bipartite condition. But, since the number of partitions of a set of k vertices is 2^k, that's not feasible: It would take time exponential in the number of vertices.

But BFS helps. We will visit the vertices, dividing the vertices into *even* layers and *odd* layers. If an edge in the BFS leads to a vertex we have already seen, and the original path placed it in an even layer, then the new edge should also place it in an even layer. If the original path placed it in an odd layer, then the new edge should also place it in an odd layer.

You will adapt the BFS code so that every time it finds an edge to a vertex v_1 to a vertex v_2, if v_2 was already discovered, then the new path agrees with the old path about whether v_2 belongs in the odd layers or the even layers.

Actually, we can make the test simpler. A property of BFS is that every edge connects two vertices that are either (i) both in the same layer, or else (ii) in adjacent layers. The reason is that if there's an edge between v_1 and v_2, and

the first of them is discovered in layer ℓ, then the other must be discovered no later than the next layer $\ell + 1$. It could also be discovered in layer ℓ.

If v_1 and v_2 are both discovered in layer ℓ, and there's an edge between them, then this edge will cause an even-odd conflict.

So your code only needs to check that every edge crosses layers.

Since the graph may not be connected, you will have to run BFS repeatedly to make sure that the whole graph has the bipartite property.

If the test succeeds, emit a partition of the vertices. If it fails, emit your starting point for the current BFS run, and also an edge between vertices in the same layer.

Finding a Cycle. Suppose that G is a *directed* graph. We would like to find a cycle if there is one, or else report that G has no cycles. The recursive depth-first search is a good starting point.

Suppose that every time you make a recursive call to explore from a new vertex v, you also record that v is *active*. When you return out of the recursive call, record that it is *no longer* active. For instance, you could use a new field `dfs_results[v].active`, setting it to `true` on the call, and to `false` on the return. If, when exploring u, you ever encounter an edge leading to an active vertex v, then you have discovered a cycle. Use the predecessors `dfs_results[u].pi` to follow the path from u back to v and output the cycle.

A single cycle is enough to show; your code can stop if it finds one. Or you may show more than one.

There's pseudocode in the book (*CLRS*, p. 601) to print a path.