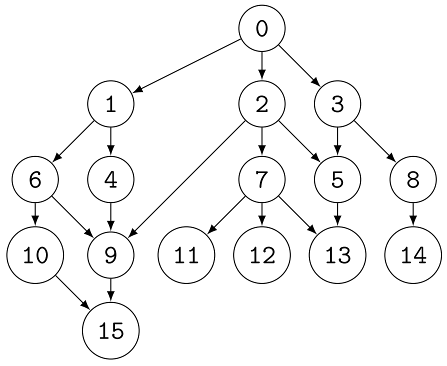
I pledge that I have neither given nor received any unauthorized aid on this assignment.

Input Data

Graph



Adjacency Matrix

```
0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0
0,0,0,0,0,1,0,1,0,1,0,0,0,0,0,0
0,0,0,0,0,1,0,0,1,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0
0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

Data Structures

Graph

Python:

```python
class Graph:
    def __init__(self, num_nodes = 0):
        self.p = [[0 for x in xrange(num_nodes)] for x in xrange(num_nodes)];
        self.num_nodes = num_nodes;

    def get_adjacent(self, i):
        return [x for x in xrange(self.num_nodes()) if self.p[i][x] != 0];

    def is_edge(self, i, j):
        return self.p[i][j] != 0;

    def num_nodes(self):
        return self.num_nodes;

    def set_edge(self, i, j, value):
        self.p[i][j] = value;
```

Discussion

For all of the homework problems, I implemented and made use of the simple graph specification (CS 319 Lecture 7, Slide 5). I also added a convenience method, `get_adjacent`, which queries the internal edge matrix for a particular node to find adjacent node indices. For stack and queue structures, the Python `list` contains the functions necessary to avoid deep implementation.

Problem 1 [COMPLETE]

Implementation

Python:

```python
def DFS(graph, vertex = 0, visited = None):

    if visited is None:
        visited = [False for x in xrange(graph.num_nodes())];

    if not visited[vertex]:
        visited[vertex] = True;
        sys.stdout.write(str(vertex) + " ");

    for adjacent in graph.get_adjacent(vertex):
        if adjacent is vertex:
            continue;
        if not visited[adjacent]:
            DFS(graph, adjacent, visited);
```

Output

```
0 1 4 9 15 6 10 2 5 13 7 11 12 3 8 14
```

Discussion

My DFS implementation is recursive, and selectively visits nodes based on their visited status. Python passes variable by reference, so a simple `list` was used to monitor the boolean status of the graph's nodes beginning at the scope of the first function call. It was during this implementation that I created the `get_adjacent` method on the graph imlpementation, instead of creating a range sequence and iterating through all nodes. A ”self-check” is still required, because a node may loop back to itself (causing an infinite recursion in the code). By making the implementation recursive, it is treating each node as the ”parent,” or entry point, and traversing, depth-first, only the unvisited nodes that it has directional paths that connect the two.

Problem 2 [COMPLETE]

Implementation

Python:

```python
def DFS(graph, vertex = 0, visited = None, depth = -1):

    if visited is None:
        visited = [False for x in xrange(graph.num_nodes())];

    if not visited[vertex]:
        visited[vertex] = True;
        sys.stdout.write(str(vertex) + " ");

    if depth is 0:
        return;

    for adjacent in graph.get_adjacent(vertex):
        if adjacent is vertex:
            continue;
        DFS(graph, adjacent, visited, depth - 1);

def IDS(graph):

    visited = [False for x in xrange(graph.num_nodes())];

    for depth in xrange(graph.num_nodes()):
        if all(visited):
            break;
        DFS(graph, 0, visited, depth);
```

Output

```
0 1 2 3 4 6 5 7 9 8 10 13 11 12 15 14
```

Discussion

I implemented iterative depth by creating a ”control” function to repeatedly call the depth-first (depth-limited) search with increasing depth until all nodes have been visited. The visitation status is maintained at the scope of the controlling function, so that multiple (seperate) call to the depth-first search would not lose context on visitation status. I also adjusted the recursion check to force recursive calls to nodes that have already been visited, so that the iterative nature of this solution would function without short-circuiting after the first iteration.

Problem 3 [COMPLETE]

Implementation

Python:

```python
def improved_DFS(graph, vertex = 0, visited = None):

    if visited is None:
        visited = [False for x in range(graph.num_nodes())];

    if not visited[vertex]:
        visited[vertex] = True;
        sys.stdout.write(str(vertex) + " ");

    memory = [];

    for adjacent in graph.get_adjacent(vertex):
        if adjacent is vertex:
            continue;
        if not visited[adjacent]:
            memory.append(adjacent);

    return memory;

def improved_IDS(graph, vertex = 0):

    visited = [False for x in range(graph.num_nodes())];
    memory = [vertex];

    if not all(visited):
        while len(memory) > 0:
            memory.extend(improved_DFS(graph, memory.pop(0), visited));
            memory = unique(memory);

def unique(seq):
    return list(_unique(seq));

def _unique(seq):
    seen = set();
    for x in seq:
        if x in seen:
            continue;
        seen.add(x);
        yield x;
```

Output

```
0 1 2 3 4 6 5 7 9 8 10 13 11 12 15 14
```

Discussion

For this solution, I altered the depth-limited depth-first search to return a list of nodes that were one-level deeper than the depth allows. Each iteration ”pumps” the queue with ”next” nodes, so that the queue will increase in length until all nodes have been queued for visitation while concurrently being ”drained” by the control function. No more than two levels will be in the queue at any given time. To further optimize this implementation, each time the ”master” queue is modified, an order-preserving uniqueness filter is applied to the queue to prevent the duplicate iterationof nodes that have already been visited. This occurs when there are mutliple edges to a node.