COMPSCI 2C03 – Week 10 Exercises

Sample solutions and notes on sample solutions for this week's exercises.

Lecture 1: Graphs

1. Exercise 4.1.1: What is the maximum number of edges in a graph with V vertices and no parallel edges or self loops? What is the minimum number of edges in a graph with V vertices, none of which are isolated (isolated = degree 0).

   Max: V-1 + V-2 + … + 1 = (V-1)V/2 = (V^2-V)/2

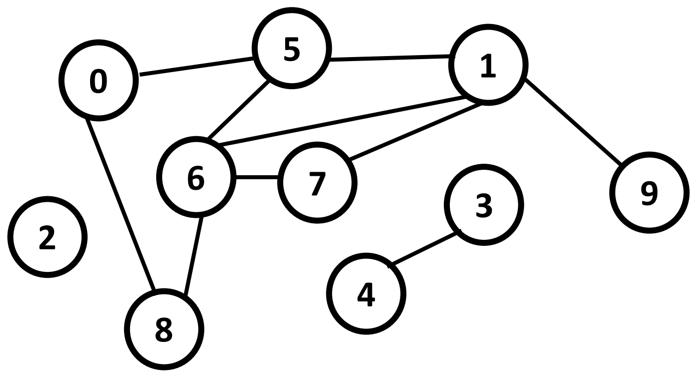
   Min: one edge for every pair of vertices, plus one edge if odd number of vertices. So ceiling(V/2).

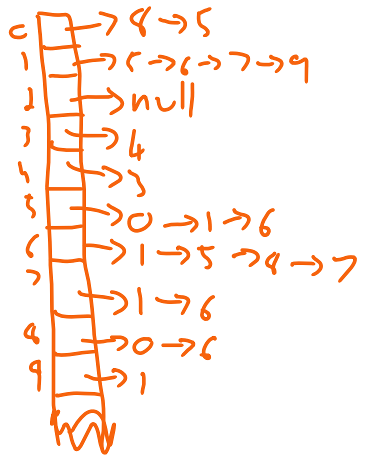
2. Exercise 4.1.4: Add an operation **hasEdge()** to the pseudocode Graph implementation. This operation takes a graph **g**, and two edge numbers **v** and **w** and returns **true** if **g** has the edge **v-w**, **false** otherwise.

```
hasEdge(g, v, w):
    for vertex in adj(g)
        if vertex == w:
            return true
    return false

# no need to check w's adjacency list assuming graph is consistent
```

3. Draw an adjacency list representation for the graph below.





Lecture 2: Graph Search

4. Use the adjacency list representation from question 3 to simulate a **depth-first search** from vertex 1 and draw the resulting **edgeTo** array and the corresponding output tree.

edge	5	Null	Null	Null	Null	1	8	6	0	1
marked	T	T				T	T	T	T	T
Index	0	1	2	3	4	5	6	7	8	9

Note your answer might differ if the order of vertices in your adjacency list differ.

Recursive calls to each vertex…

1

5

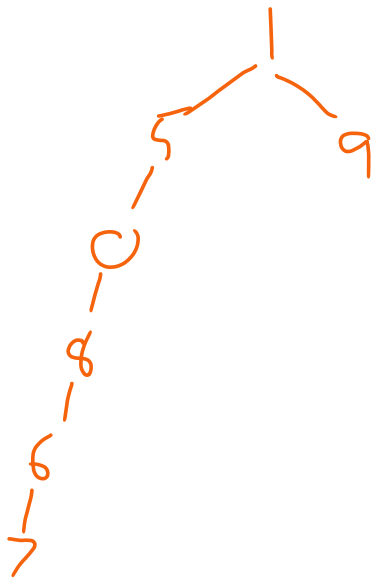
0

8

6

7

9



5. Exercise 4.1.10: Prove that every connected graph has a vertex whose removal (including all incident edges) will not disconnect the graph. Write a pseudocode function based on DFS that will find such a vertex. (Hint: Consider a vertex whose adjacent vertices are all marked.)

   In the previous solution, consider just the connected component containing the vertex 1. From the DFS trace above, deleting 7 will not disconnect the connected component because once we arrive at 7, we have already visited all its adjacent nodes (they're all marked). If we had visited 9 first, we would have found another eligible vertex. There will always be such a vertex.

   Another way to prove is to consider the longest simple path in the graph between two vertices u and v. If w is any other vertex of the graph, there must be a simple path from u to w that does not go through v. If there was such a path, then the longest simple path would be between u and w, contradicting the assumption that the longest simple path is through u and v. So, w will stay connected if we delete v.

```
marked ← array(num_vertices(g)) of False

find_safe_removal(g, v):
  marked[v] ← True
  for w in adj(g, v):
    if not marked[w]:
      return find_safe_removal (g, w)     # We only need to take 1 path
  return v                                # No paths to take. This node
                                          # is safe to delete.
```

6. **Tricky:** Write a **has_cycle** function in pseudocode. This function should take a (connected) graph and return **true** if the graph contains a cycle with no repeated edges (i.e., don't consider a single edge to form a cycle), **false** if it does not.

The key to solving this is keeping track of what's on the current path and at what position (depth). If a node is on the path when you process it, and it's more than one step behind you, you've found a cycle of more than 2 nodes.

```
marked ← array(num_vertices(g)) of False
onpath ← array(num_vertices(g)) of False
pathdepth ← array(num_vertices(g)) of 0

has_cycle(g, v, depth):
  depth ← depth + 1           # increment the current depth
  marked[v] ← True
  onpath [v] ← True           # this node is on the path!
  pathdepth [v] ← depth       # set the current depth

  for w in adj(g, v):
    if onpath[v]:
      if pathdepth[w] < depth - 1:  # found node 2 or more steps back
        return True
    if not marked[w]:
      result ← has_cycle (g, w, depth)
      if result == True:
            return True

  onpath [v] ← false          # this node is no longer on the path

  return False

Always call has_cycle with depth 0 to start.
```

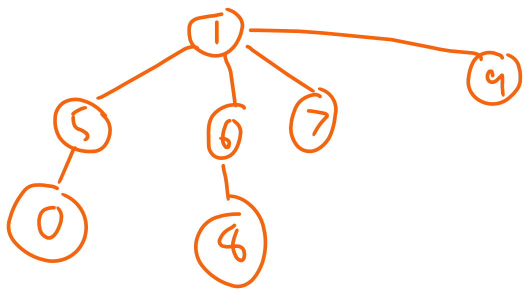
Lecture 3: Shortest Paths and Connectivity

7. Use the adjacency list representation from question 3 to simulate a **breadth-first search** from vertex 1 and draw the resulting **edgeTo** array and the corresponding output tree.

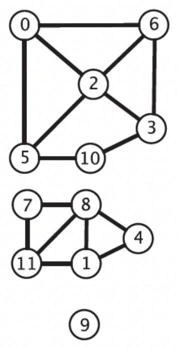
edge	5	Null	Null	Null	Null	1	1	1	6	1
marked	T	T				T	T	T	T	T
Index	0	1	2	3	4	5	6	7	8	9

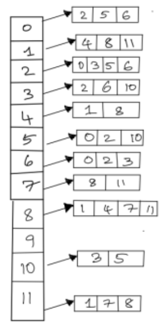
Note your answer might differ if the order of vertices in your adjacency list differ.

Queue: 1x 5x 6x 7x 9x 0x 8x



8. Simulate the Connected Components algorithm presented in class on the graph below.





id	0	1	0	0	1	0	0	1	1	2	0	1
marked	T	T	T	T	T	T	T	T	T	T	T	T
Index	0	1	2	3	4	5	6	7	8	9	10	11

```
Count: 0, start vertex 0
0
         2
                   3
                            6
                            10
                   5
Count 1, start vertex 1
1
         4
                   8
                            7
                                               11
Count 2, startvertex 9
9
```

9. Exercise 4.1.12: What does the BFS tree tell us about the distance from v to w when neither is at the root?

   It gets us an upper bound on the distance from v to w. BFS computes the shortest distances from source s to all vertices connected to s. (Page 541, proposition B) If s is connected to v and w then we have the shortest path from s to v and from s to w but not necessarily the shortest path from v to w. The distance between v and w is less than or equal to the sum of shortest distances from s to v and s to w.

10. How can the number of connected components of a graph change when a new edge is added?

    Adding an edge either keep the number of connected components unchanged if it is intra-component edge (between the nodes of one component), or decrease it if it is an inter-component edge (connecting a node in one component to a node in another component).

11. Modify the **bfs** pseudocode so that it computes an array of shortest paths lengths between **s** and all other vertices.

    Could do this by storing path lengths in the queue along with vertices, and incrementing each time.

    Or we could have an array of distances seen so far (instead of edge_to) and set the distance when we enqueue an unmarked vertex. The distance is one more than the distance of where we're coming from.

```
paths_bfs(g, s):
      marked ← array(num_vertices(g)) of False
      distances ← array(num_vertices(g)) of -1
      unvisited ← Queue()
      marked[s] ← True
      unvisited.put(s)
      distances[s] ← 0
      while not queue_empty(unvisited):
            v ← dequeue(unvisited)
            for w in adj(g, v):
                  if not marked[w]:
                        distances[w] ← distances[v]+1
                        marked[w] ← True
                        enqueue(unvisited, w)
      return distances
```

12. Modify or add to the **has_cycle** code from question 6 so that it works for a non-connected graph. Note that even if you did not complete that question, you can just assume that the function exists and create a new function that calls it to get the job done.

```
Reusing the array and function declarations from question 6.

has_cycle_non_connected(g):
      for s ← 0 to num_vertices(g) - 1:
            if not marked[s]:
                  if has_cycle (g, s, 0): return True
      return False
```