8.3 Depth-first search and Breadth-first search

- when we work with graphs, we may wish to do something to each node exactly once
  - e.g., distribute information to all computers in a network
  - do not want to send to any computer twice
- examine two techniques that accomplish this traversal
  - **depth-first search** - go as far as possible “down” a path before considering another
  - **breadth-first search** - go evenly in many directions
- for both:
  - first choose one node in graph as starting point
  - “visit” the node represents action to be done at each node
    - e.g., in search: visiting node = check for information we want
  - work for both directed and undirected graphs
- either method can be used to determine if a graph is connected.
  - create list of nodes visited during traversal
  - compare to set of all nodes in graph.
    - if same, graph is connected
    - if not, some nodes cannot be reached. (graph not connected)

8.3.1 Depth-first traversal

- visit the starting node then proceed to follow edges through graph until reach dead end
  - in undirected graph, node is dead end if all nodes adjacent to it have already been visited
  - in directed graph, there are no outgoing edges
  - when reach a dead end, we back up along the path until we find an unvisited adjacent node
    - then continue in that new direction
  - algorithm ends when we back up to starting node and all adjacent nodes have been visited.
  - in examples, we will select smallest adjacent node

- recursive DFS algorithm:
  - \( DFS(G, v) \):
    - \( G \) is graph
    - \( v \) is current node
    - visit(v)
    - mark(v)
    - for every edge \( vw \) in \( G \):
      - if \( w \) not visited:
        - \( DFS(G, w) \)
  - recursive algorithm relies on system stack to keep track of where it has been in the graph
  - so it can properly back up when it reaches a dead end.
  - can also create a non-recursive algorithm using a stack...

8.3.2 Breadth-first traversal

- visit starting node and then, on first pass, visit all nodes directly connected to it
  - in the second pass, we visit all nodes two edges “away” from starting node
  - with each new pass, we visit nodes that are one more edge away
  - there might be cycles in the graph
    - possible for node to be on two paths at different lengths from starting node
    - but since we visit node for the first time along shortest path,
      - will not need to consider it again
    - need to keep a list of visited nodes
    - or use a variable w/in the node to mark it as visited
breadth first traversal: [1, 2, 8, 3, 7, 4, 5, 9, 6]

breadth first uses a queue

- hence why 9 is visited before 6 in example.
- because node 4 is on the queue before node 5
- => we visit the nodes connected to node 4 before we visit nodes connected to node 5

algorithm:

- **BFS(G, v)**
  - G is graph
  - v is current node
  - `visit(v)`
  - `mark(v)`
  - `enqueue(v)`
  - while queue not empty:
    - `dequeue x`
    - for every edge xw in G do
      - if w not marked
        - `visit(w)`
        - `mark(w)`
        - `enqueue(w)`

algorithm will add the root of the breadth-first traversal tree to the queue but then immediately remove it

- as it looks at nodes adjacent to root, they will be added to end of queue
- once all nodes adjacent to root have been visited, we return to queue and get first of those nodes

8.3.3 traversal analysis

- assume that work done as we visit each node is most complex part
- work done to check to see if adjacent node has been visited and work to traverse edges is not significant in this case
  - order of the algorithm is the number of times a node is visited.
  - how many times? visit each node exactly 1 time.
  - \( O(|V|) = O(N) \)
    - at worst, we will examine all |E| edges (N - 1) => order N