Graph traversals

Definition

> A Graph consists of a finite set of vertices or nodes and set of Edges which connect a pair of nodes.

Representation

Adjacency matrix

> Adjacency Matrix is a 2D array of size $|V| \times |V|$ where V is the number of vertices in a graph. Let the 2D array be `adj[][]`, a slot `adj[i][j] = 1` indicates that there is an edge from vertex i to vertex j and 0 means no edge.

Matrix

```
[
  [0, 0, 0, 1],
  [0, 0, 0, 1],
  [0, 0, 0, 1],
  [1, 1, 1, 0]
]
```

Graph

```
     4
   / | \
  /  |  \
 /   |   \
1    2    3
```

Pros vs. Cons

Pros:

- Representation is easier to implement and follow
- Removing an edge takes $O(1)$ time
- Queries like whether there is an edge from vertex `u` to vertex `v` are efficient and can be done $O(1)$

Cons:

- Consumes more space $O(V^2)$. Even if the graph is sparse contains less number of edges, it consumes the same space
- Adding a vertex is $O(V^2)$ time

Miscellaneous

We can sacrifice the $O(1)$ to get a better space efficiency by using "Adjacency List".

Source

Warm up

How to get neighbors of a particular node using an Adjacency Matrix?

```
List<Integer> neighbors(int[][] matrix, int i) {
  // TODO: note that we are given adjacency matrix or some 2D array of integer
  // and we want to find list of integer where they are connected with node `i`
  //
  //
}
```

Solution

```
List<Integer> neighbors(int[][] matrix, int i) {
  List<Integer> neighbors = new LinkedList<>();
  for (int node : matrix[i]) {
    if (node == 1) {
      neighbors.add(node);
    }
  }

  return neighbors;
}
```

Breadth-first search (BFS)

> Algorithm for traversing or searching tree or graph data structures. It starts at the tree root, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

BFS implementation notes

- Do we need a `Queue` or `Stack`?
- How do we make sure we don't revisit the same node?

BFS implementation in Java

```
List<Node> bfsTraversal(Node node) {
    if (node == null) return new LinkedList<Node>();                    // if node is null then immediately

    LinkedList<Node> visited = new LinkedList<>();                      // list of visited nodes
    Queue<Node> queue = new LinkedList<>();                             // create queue of nodes
    queue.enqueue(node);                                                // add `root` to the queue

    while (!queue.isEmpty()) {                                          // while queue is not empty
        Node current = queue.dequeue();                                 // dequeue a node

        if (visited.contains(current)) continue;                        // short-circuit if node already visited
        visited.add(current);                                           // mark the node as visited

        List<Node> neighbors = getNeighbors(current);                   // get neighbors of current node

        for (Node neighbor : neighbors) {
            if (!visited.contains(neighbor)) {                          // we only add unvisited neighbors
                queue.add(neighbor);                                    // enqueue the next neighbor for traversal
            }
        }
    }

    return visited;                                                     // return traversal list
}
```

Exercise

Let's analyze the queue if we attempt to do a BFS traversal starting from node "1"

or `bfsTraversal(node 1)`

```
     4
   / | \
  /  |  \
 /   |   \
1    2    3
```

Solution

```
visited: {            } queue: { 1    }   // start by adding the node to queue
visited: { 1          } queue: {      }   // dequeue 1 and mark 1 as visited
visited: { 1          } queue: { 4    }   // enqueue un-visited neighbors of 1
visited: { 1, 4       } queue: {      }   // mark 4 as visited
visited: { 1, 4       } queue: { 2, 3 }   // enqueue un-visited neighbors of 4
visited: { 1, 4, 2    } queue: { 3    }   // dequeue 2 and mark 2 as visited
visited: { 1, 4, 2, 3 } queue: {      }   // dequeue 3 mark 3 as visited
```

Depth-first search (DFS)

> Algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores as far as possible along each branch before backtracking.

DFS implementation notes

- Do we need a `Queue` or `Stack`?

DFS Implementation in Java

```
public List<Node> dfsTraversal(Node node) {
    LinkedList<Node> visited = new LinkedList<>();    // list of visited nodes
    Stack<Node> stack = new LinkedList<>();           // Initialize an empty stack
    stack.push(node);                                 // add the node to the top of the stack

    while (!stack.isEmpty()) {
        Node current = stack.pop();                   // pop the top current from the stack
        if (visited.contains(current)) continue;      // short-circuit if node already visited

        visited.add(current);                         // add current node as visited

        List<Node> neighbors = getNeighbors(current); // get neighbors of current node
        for (Node neighbor : neighbors) {
            if (!visited.contains(neighbor)) {        // we only add unvisited neighbors
                stack.push(neighbor);                 // enqueue the next neighbor for traversal
            }
        }
    }

    return visited;                                   // return traversal list
}
```

Exercise

Let's analyze the stack if we attempt to do a DFS traversal starting from node "1"

or `dfsTraversal(node 1)`

```
     4
   / | \
  /  |  \
 /   |   \
1    2    3
```

Solution

```
visited: {            } stack: { 1    }   // start by adding the node to stack
visited: { 1          } stack: {      }   // pop 1 and mark 1 as visited
visited: { 1          } stack: { 4    }   // push un-visited neighbors of 1
visited: { 1, 4       } stack: {      }   // mark 4 as visited
visited: { 1, 4       } stack: { 2, 3 }   // push un-visited neighbors of 4
visited: { 1, 4, 2    } stack: { 3    }   // pop 2 and mark 2 as visited
visited: { 1, 4, 2, 3 } stack: {      }   // pop 3 mark 3 as visited
```