CS61B – Algorithms

Anmol Parande

Spring 2019 - Professor Hug

1 Tree Traversals

1.1 Preorder Traversal

In a preorder traversal a node is visited before any of its children are visited.

```
public void preorder(TreeNode node) {
    if (node == null) {
        return;
    }
    visit(node);
    for (TreeNode child : node.children) {
        preorder(child);
    }
}
```

1.2 In Order Traversal

For a Binary Tree, an in order traversal is simply the elements when read from left to right.

```
public void inorder(BNode node) {
    if (node == null) {
        return;
    }
    inorder(node.left);
    visit(node);
    inorder(node.right);
}
```

1.3 Post Order Traversal

In a postorder traversal, a node is visited after its children are visited.

```
public void postorder(TreeNode node) {
    if (node == null) {
        return;
    }
    for (TreeNode child : node.children) {
        postorder(child);
    }
    visit(child);
}
```

1.4 Level Order Traversal

In a level order traversal, the nodes of each level of the tree are visited in order. It is done through a Breadth-First Search of the tree (see below)

2 Breath-First Search

The idea of breadth first search is to visit all children before visiting the children's children. Its runtime is $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges.

```
repeat until queue is empty:
    remove vertex v from queue
    for each unmarked vertex n adjacent to v
        mark n
        keep track of some statistic
        add n to end the queue
```

3 Depth-First Search

The idea of depth-first search is to visit all the children of a child before visiting the nodes other children. The preorder is the order in which DFS calls are made. The postorder is the order of DFS returns. It's runtime is $O(|V| + |E|)$.

3.1 Pre-Order

```
add start node to the stack
while stack is not empty:
    v = stack.pop
    mark(v)
    for vertex n adjacent to v:
        mark(n)
        stack.push(n)
```

3.2 PostOrder

```
DFS(Vertex v) {
    mark v visited
    set color of v to gray
    for each successor v' of v {
            if v' not yet visited {
                DFS(v')
            }
    }
    doAction(v)
}
```

4 Dijkstra's Algorithm

Dijkstra's algorithm helps build the shortest paths tree starting from a particular source node.

```
add all vertices to priority queue with infinity priority
set the source's priority to 0
while priority queue is not empty
    remove the smallest vertex v
    mark v
    relax edges from v
```

The relaxation procedure is as follows

```
if w is visited
    return
w = weight(parent) + edge.weight
if w < v.priority
    priority = w
```

Dijkstra's algorithm runs in $O(|E|log|V|)$

5 A^* Search

A^* search is like Dijkstra's in that it finds the shortest path between two nodes. However, it is different because the priority of each vertex is the distance to that vertex plus a heuristic value

A^* requires a goal vertex to run. It's runtime is $O(|E|log|V|)$

6 MSTs

A Minimum Spanning Tree (MST) is a tree connecting all vertices in a graph which has the minimum total weight (the sum of its edges is minimized).

6.1 Prim's Algorithm

```
add source vertex to PQ
set all other vertex priorities to infinity
for edge out of V:
    if destination is unmarked:
        relax(V) using the distance to the tree
```

Notice that this is basically Dijkstra's algorithm but we are using distance to tree rather than distance to source. The runtime is $O(|E|log|V|)$.

6.2 Kruskal's Algorithm

```
Make a priority queue of all edges
Create a Disjoint Set of all the vertices
while !pq.empty && mst.size() < V - 1:
    e = pq.pop
    if source not connected to target in DisjointSet
        connect(source, target)
        add e to MST
```

The runtime is also $O(|E|log|V|)$.