April 19 – Graph Traversals & MST
Brad Solomon
Mid-Project Check-ins this week!

Discuss:

Current Progress (First deliverable done?)

Future Progress (What do you have left to do?)

Group Cohesion (Any issues or concerns?)
Learning Objectives

• Review BFS and discuss pseudo-code for DFS on graphs

• Analyze and contrast BFS/DFS algorithms

• Introduce Minimum Spanning Tree (MST) problem
Traversals: BFS

**Adjacent Edges**

<table>
<thead>
<tr>
<th>v</th>
<th>d</th>
<th>P</th>
<th>Adjacent Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>-</td>
<td>B C D</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>A</td>
<td>A C E</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>A</td>
<td>A B D E F</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>A</td>
<td>A C F H</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td>C</td>
<td>B C G</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>C</td>
<td>C D G</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>E</td>
<td>E F H</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
<td>D</td>
<td>D G</td>
</tr>
</tbody>
</table>

**Traversal Order:** G H F E D C B A
BFS Observations

**Obs. 1:** BFS can be used to count components.

**Obs. 2:** BFS can be used to detect cycles.

**Obs. 3:** In BFS, $d$ provides the shortest distance to every vertex.

**Obs. 4:** In BFS, the endpoints of a cross edge never differ in distance, $d$, by more than 1:

$$|d(u) - d(v)| \leq 1$$
Traversal: DFS
Traversal: DFS

Discovery Edge

Back Edge
BFS(G):
Input: Graph, G
Output: A labeling of the edges on G as discovery and cross edges

foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
        BFS(G, v)

BFS(G, v):
Queue q
setLabel(v, VISITED)
q.enqueue(v)
while !q.empty():
    v = q.dequeue()
    foreach (Vertex w : G.adjacent(v)):
        if getLabel(w) == UNEXPLORED:
           setLabel(v, w, DISCOVERY)
            setLabel(w, VISITED)
            q.enqueue(w)
        elseif getLabel(v, w) == UNEXPLORED:
           setLabel(v, w, CROSS)
DFS(G):

Input: Graph, G
Output: A labeling of the edges on G as discovery and back edges

foreach (Vertex v : G.vertices()):
    setLabel(v, UNEXPLORED)
foreach (Edge e : G.edges()):
    setLabel(e, UNEXPLORED)
foreach (Vertex v : G.vertices()):
    if getLabel(v) == UNEXPLORED:
        DFS(G, v)

DFS(G, v):

    Queue q
    setLabel(v, VISITED)
    q.enqueue(v)
    while !q.empty():
        v = q.dequeue()
        foreach (Vertex w : G.adjacent(v)):
            if getLabel(w) == UNEXPLORED:
                setLabel(v, w, DISCOVERY)
                setLabel(w, VISITED)
                DFS(G, w)
            elseif getLabel(v, w) == UNEXPLORED:
                setLabel(v, w, BACK)
DFS Observations

**Obs. 1:** DFS can be used to count components.

**Obs. 2:** DFS can be used to detect cycles.

**Obs. 3:** In DFS, $d$ provides no clear meaning.
# DFS vs BFS

<table>
<thead>
<tr>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros:</strong></td>
<td><strong>Pros:</strong></td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
</tbody>
</table>
Minimum Spanning Tree Algorithms

**Input:** Connected, undirected graph $G$ with edge weights (unconstrained, but must be additive)

**Output:** A graph $G'$ with the following properties:
- $G'$ is a spanning graph of $G$
- $G'$ is a tree (connected, acyclic)
- $G'$ has a minimal total weight among all spanning trees
Kruskal’s Algorithm

**KruskalMST(G):**
1. DisjointSets forest
2. foreach (Vertex v : G):
   3.   forest.makeSet(v)
4. PriorityQueue Q  // min edge weight
5. foreach (Edge e : G):
   6.   Q.insert(e)
7. Graph T = (V, {})
8. while |T.edges()| < n-1:
   9.   Vertex (u, v) = Q.removeMin()
10.  if forest.find(u) != forest.find(v):
11.    T.addEdge(u, v)
12.    forest.union( forest.find(u),
13.                   forest.find(v) )
14. return T
Kruskal’s Algorithm

KruskalMST(G):
DisjointSets forest

foreach (Vertex v : G):
forest.makeSet(v)

PriorityQueue Q // min edge weight

foreach (Edge e : G):
Q.insert(e)

Graph T = (V, {})

while |T.edges()| < n-1:
Vertex (u, v) = Q.removeMin()
if forest.find(u) != forest.find(v):
T.addEdge(u, v)
forest.union( forest.find(u),
forest.find(v) )

return T

Priority Queue:
Heap Sorted Array
Building (Line 6-8)
Each removeMin (Line 13)
### Kruskal’s Algorithm

**Priority Queue:**

<table>
<thead>
<tr>
<th>Total Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Sorted Array</td>
</tr>
</tbody>
</table>

**Priority Queue:**

1. **Heap**
2. **Sorted Array**

```python
KruskalMST(G):

1. DisjointSets forest
2. foreach (Vertex v : G):
   3. forest.makeSet(v)
5. PriorityQueue Q // min edge weight
7. foreach (Edge e : G):
   8. Q.insert(e)
9. Graph T = (V, {})
11. while |T.edges()| < n-1:
   12.   Vertex (u, v) = Q.removeMin()
   14.   if forest.find(u) != forest.find(v):
   15.     T.addEdge(u, v)
   16.     forest.union( forest.find(u),
   17.                     forest.find(v) )
19. return T
```
Kruskal’s Algorithm

Which Priority Queue Implementation is better for running Kruskal’s Algorithm?

• Heap:

• Sorted Array: