Flipped lecture – L01

• Why flip?
  – Why not?
  – Find the most efficient way to deliver content
  – Improve the outcomes
  – Focus more on conceptual and conventional
  – Quick poll

• Format of the flipped Lecture
  – Mini lecture – a quick overview of concepts (30 mins)
    - Directed Graphs, MST’s
  – Group Worksheets (30 minutes)
    • Work with 2-3 students next to you
  – Discussion of Solutions (20 mins)
# Time spent on videos

<table>
<thead>
<tr>
<th>Topic</th>
<th>Completion (%)</th>
<th>Total Spend Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.14.5 Directed Graphs - Strong Components</td>
<td>100%</td>
<td>197 hours 57 minutes 3 seconds</td>
</tr>
<tr>
<td>7.14.4 Directed Graphs - Topological Sort</td>
<td>45.7%</td>
<td>40 hours 33 minutes 10 seconds</td>
</tr>
<tr>
<td>7.14.1 Directed Graphs - Introduction</td>
<td>44%</td>
<td>87 hours 3 minutes 58 seconds</td>
</tr>
<tr>
<td>7.14.3 Directed Graphs - digraph Search</td>
<td>40.4%</td>
<td>79 hours 55 minutes 6 seconds</td>
</tr>
<tr>
<td>8.15.3 Minimum Spanning Trees - Edge Weighted Graph</td>
<td>29.7%</td>
<td>58 hours 50 minutes 53 seconds</td>
</tr>
<tr>
<td>8.15.5 Minimum Spanning Trees - Prim's Algorithm</td>
<td>27.1%</td>
<td>53 hours 40 minutes 58 seconds</td>
</tr>
<tr>
<td>8.15.4 Minimum Spanning Trees - Kruskals Algorithm</td>
<td>24.2%</td>
<td>36 hours 19 minutes 25 seconds</td>
</tr>
<tr>
<td>8.15.2 Minimum Spanning Trees - Greedy Algorithms</td>
<td>18.3%</td>
<td>36 hours 15 minutes 21 seconds</td>
</tr>
<tr>
<td>8.15.1 Minimum Spanning Trees - Introduction</td>
<td>18.3%</td>
<td>36 hours 15 minutes 21 seconds</td>
</tr>
</tbody>
</table>
public class Digraph

Digraph(int V)

Digraph(In in)

void addEdge(int v, int w)

Iterable<Integer> adj(int v)

int V()

int E()

Digraph reverse()

String toString()
Implementation of a weighted digraph using Adjacency List represented by **Array of Bags**

(flexible list length)
# Order of Growth

<table>
<thead>
<tr>
<th>representation</th>
<th>space</th>
<th>insert edge from $v$ to $w$</th>
<th>edge from $v$ to $w$?</th>
<th>iterate over vertices adjacent from $v$?</th>
</tr>
</thead>
<tbody>
<tr>
<td>list of edges</td>
<td>$E$</td>
<td>1</td>
<td>$E$</td>
<td>$E$</td>
</tr>
<tr>
<td>adjacency matrix</td>
<td>$V^2$</td>
<td>$1^\dagger$</td>
<td>1</td>
<td>$V$</td>
</tr>
<tr>
<td>adjacency lists</td>
<td>$E + V$</td>
<td>1</td>
<td>$\text{outdegree}(v)$</td>
<td>$\text{outdegree}(v)$</td>
</tr>
</tbody>
</table>
In degree and out degree

• Indegree of a vertex v
  – Number of edges directed at the vertex v
  – Order of growth
    • adjacency list - E+V
    • adjacency matrix - V

• Outdegree of vertex v
  – Number of edges from the vertex v to other vertices
  – Order of growth
    • Adjacency list - outdegree(v)
    • Adjacency matrix - V
Directed graphs: quiz 1

Which is order of growth of running time to iterate over all vertices adjacent from \( v \) in a digraph using the adjacency-lists representation?

A. \( \text{indegree}(v) \)
B. \( \text{outdegree}(v) \)
C. \( \text{degree}(v) \)
D. \( V \)
E. \( I \ don't \ know. \)
Topological Order

**Proposition.** A digraph has a topological order iff no directed cycle.

**Pf.**
- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

---

**orderings**

**Orderings.**
- **Preorder:** order in which \( \text{dfs}() \) is called.
- **Postorder:** order in which \( \text{dfs}() \) returns.
- **Reverse postorder:** reverse order in which \( \text{dfs}() \) returns.
If there is a topological sort, does it matter which node we start with in DFS?
What does the code do if vertex 0 is marked as s?
Def. Vertices $v$ and $w$ are strongly connected if there is both a directed path from $v$ to $w$ and a directed path from $w$ to $v$.

Def. A strong component is a maximal subset of strongly-connected vertices.

Key property. Strong connectivity is an equivalence relation:

5 strongly-connected components

How to find strong components?
How to Find SCC’s

Kosaraju-Sharir algorithm: intuition

Reverse graph. Strong components in $G$ are same as in $G^R$.

Kernel DAG. Contract each strong component into a single vertex.

Idea.
- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.

Proposition. Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to $E + V$. 
Computing SCC’s

- Step 1: Compute the reverse post order of $G^R$
- Step 2: Visit $G$ in the order of reverse post order found in step 1
Minimum Spanning Tree (MST)

Facts and Questions
1. An undirected weighted graph
2. Connected
3. Find a subgraph that minimizes the total weight of edges
4. How many edges are in a MST?
**Proposition:** A connected graph with distinct edge weights has a unique MST

How to find it?
1. Sort all the edges - $E \log E$ \textbf{or} build a minPQ of edges
2. Find $V-1$ smallest edges \textbf{or} do delMin, $V-1$ times
Cut property

**Def.** A cut in a graph is a partition of its vertices into two (nonempty) sets.

**Def.** A crossing edge connects a vertex in one set with a vertex in the other.

**Cut property.** Given any cut, the crossing edge of min weight is in the MST.
What is the min weighted edge crossing the cut \{2,3,5,6\}?

**Greedy MST algorithm**

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.
API’s

```java
public class Edge implements Comparable<Edge>

    Edge(int v, int w, double weight)

    int either()

    int other(int v)

    int compareTo(Edge that)

    double weight()

    String toString()
```

```java
public class MST

    MST(EdgeWeightedGraph G) constructor

    Iterable<Edge> edges() edges in MST

    double weight() weight of MST
```
Kruskal’s

```java
private Queue<Edge> mst = new Queue<Edge>();

public KruskalMST(EdgeWeightedGraph G)
{
    MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
    UF uf = new UF(G.V());
    while (!pq.isEmpty() && mst.size() < G.V()-1)
    {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (!uf.connected(v, w))
        {
            uf.union(v, w);
            mst.enqueue(e);
        }
    }
}
```

<table>
<thead>
<tr>
<th>operation</th>
<th>frequency</th>
<th>time per op</th>
</tr>
</thead>
<tbody>
<tr>
<td>build pq</td>
<td>1</td>
<td>$E$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>union</td>
<td>$V$</td>
<td>$\log^* V^+$</td>
</tr>
<tr>
<td>connected</td>
<td>$E$</td>
<td>$\log^* V^+$</td>
</tr>
</tbody>
</table>
public KruskalMST(EdgeWeightedGraph G) {
    MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
    UF uf = new UF(G.V());
    while (!pq.isEmpty() && mst.size() < G.V() - 1) {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (!uf.connected(v, w)) {
            uf.union(v, w);
            mst.enqueue(e);
        }
    }
}
Prim’s Algorithm (lazy)

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until V - 1 edges.
Lazy Prim’s

```java
public LazyPrimMST(WeightedGraph G) {
    pq = new MinPQ<Edge>();
    mst = new Queue<Edge>();
    marked = new boolean[G.V()];
    visit(G, 0);

    while (!pq.isEmpty() && mst.size() < G.V() - 1) {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (marked[v] && marked[w]) continue;
        mst.enqueue(e);
        if (!marked[v]) visit(G, v);
        if (!marked[w]) visit(G, w);
    }
}
```

```java
private void visit(WeightedGraph G, int v) {
    marked[v] = true;
    for (Edge e : G.adj(v))
        if (!marked[e.other(v)])
            pq.insert(e);
}
```

**Proposition.** Lazy Prim’s algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to $E$ (in the worst case).

**Pf.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Frequency</th>
<th>Binary Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete min</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
<tr>
<td>Insert</td>
<td>$E$</td>
<td>$\log E$</td>
</tr>
</tbody>
</table>
Prim’s (eager implementation)

**Eager solution.** Maintain a PQ of vertices connected by an edge to $T$, where priority of vertex $v = \text{weight of shortest edge connecting } v \text{ to } T$.

- Delete min vertex $v$ and add its associated edge $e = v-w$ to $T$.
- Update PQ by considering all edges $e = v-x$ incident to $v$
  - ignore if $x$ is already in $T$
  - add $x$ to PQ if not already on it
  - decrease priority of $x$ if $v-x$ becomes shortest edge connecting $x$ to $T$
Prims Eager Trace

```
V | Edges | weight
---|-------|-------
A  |
B  |
C  |
D  |
E  |
F  |
G  |
```
Implementing a PQ with decreaseKey

```java
public class IndexMinPQ<Key extends Comparable<Key>> {
    IndexMinPQ(int N) {
        create indexed priority queue with indices 0, 1, ..., N - 1
    }
    void insert(int i, Key key) {
        associate key with index i
    }
    void decreaseKey(int i, Key key) {
        decrease the key associated with index i
    }
    boolean contains(int i) {
        is i an index on the priority queue?
    }
    int delMin() {
        remove a minimal key and return its associated index
    }
    boolean isEmpty() {
        is the priority queue empty?
    }
    int size() {
        number of keys in the priority queue
    }
}
```
The idea of decrease key

- Maintain parallel arrays keys[], pq[], and qp[] so that:
  - keys[i] is the priority of i
  - pq[i] is the index of the key in heap position i
  - qp[i] is the heap position of the key with index i
- Use swim(qp[i]) to implement decreaseKey(i, key).

```
i  0  1  2  3  4  5  6  7  8
keys[i]    A  S  O  R  T  I  N  G  -
pq[i]      -  0  6  7  2  1  5  4  3
qp[i]      1  5  4  8  7  6  2  3  -
```