Week 10.3, Friday, Oct 25

Homework 5 Due October 26 @ 11:59PM on Gradescope
Midterm 2 on October 30 (8-9:30PM)  MTHW 210 and BRNG 2280
Practice Midterm 2 Released
No PSOs next week (Due to Midterm)
No class on Monday, October 28th
Review Session: Monday, October 28th from 7-9PM (WALC 1018)
Midterm 2

- Friday PSO → BRNG 2280 (Exam Capacity 62)
- All Others → MTHW 210 (Exam Capacity 111)
- Focus: Dynamic Programming and Graph Algorithms
  - Content covered today is fair game
  - No Network Flow
- Same Rules as Midterm 1
  - Allowed to prepare 1 page of handwritten notes
  - No calculators, phones, smartwatches etc…
  - Make sure your writing implement shows up clearly when scanned!
    - Number 2 pencils work
3.6 DAGs and Topological Ordering
Directed Acyclic Graphs

Def. An **DAG** is a directed graph that contains no directed cycles.

Ex. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Def. A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Clicker: The numbers in the vertices are claimed to be in topological order. Are they?

A. Yes
B. No
C. Don’t know
Clicker: The numbers in the vertices are claimed to be in topological order. Are they?

A. Yes
B. No
C. Don’t know
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.

- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\).
- Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
- Shortest Path Computation is Faster in a DAG
Directed Acyclic Graphs

Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪
Directed Acyclic Graphs

**Lemma.** If $G$ has a topological order, then $G$ is a DAG.

**Q.** Does every DAG have a topological ordering?

**Q.** If so, how do we compute one?
Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- **Base case:** true if $n = 1$.
- **Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.**
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:
- Find a node $v$ with no incoming edges and order it first
- Delete $v$ from $G$
- Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - $\text{count}[w] = \text{remaining number of incoming edges}$
  - $S = \text{set of remaining nodes with no incoming edges}$

- Initialization: $O(m + n)$ via single scan through graph.

- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ hits 0
  - this is $O(1)$ per edge
Shortest Path in a DAG

**Input:** DAG $G=(V,E)$ (adjacency list), edge costs $c_e$ and source $s$

**Precondition:** Assume nodes are $v_1,...,v_n$ topologically sorted

- $O(n + m)$ additional work to satisfy pre-condition

**Output:** array $D$ s.t $D[v]$ denotes the minimum cost path from $s$ to $v$

(predecessor array $PRED$ s.t. $PRED[w] = v$ if $(v,w)$ is the last edge on the shortest path from $s$ to $w$)

For $v=1,...,n$

- $D[v] := \infty$ //No path from $s$ to $v$ found yet
- $D[s] := 0$

For $v=1,...,n$

- **Foreach** edge $(v,w)$ in $E$
  - **if** $D[w] > D[v] + c_{vw}$
  - $D[w] := D[v] + c_{vw}$
  - $PRED[w] := v$

$O(m)$ time --- each edge considered once
Minimum Spanning Tree (Recap)

Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued edge weights $c_e$, an MST is a subset of the edges $T \subseteq E$ such that $T$ is a spanning tree whose sum of edge weights is minimized.

Cayley's Theorem. There are $n^{n-2}$ spanning trees of $K_n$.

can't solve by brute force
Recap: Greedy Algorithms for MST

Kruskal's algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

Prim's algorithm. Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

Remark. All three algorithms produce an MST.

Proof with Distinct Edge Weights: Cut/Cycle property
- Min weight edge crossing cut must be included in MST
- Max weight edge in cycle must not be included in MST
Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

**Impact.** Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

**Observation.** For integers $a$ and $b$ we have $a < b$ if and only if $a + 1 \leq b$

**Notation:** Let $w(T) = \sum_{e \in T} c_e$ (resp. $w(T)$) denote weight of tree $T$ before (resp. after) perturbing costs

**Fact.** If $w(T) < w(T')$ then $w(T) < w(T')$.

**Proof:**

\[ w(T) < w(T) + 1 \leq w(T') \leq w(T') \]

\( \downarrow \quad \downarrow \quad \downarrow \)

n-1 edges each perturbed by at most $1/n^2$

(by observation) Perturbations are positive

\[ \leq w(T') \]

e.g., if all edge costs are integers, perturbing cost of edge $e_i$ by $i/n^3$
Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

e.g., if all edge costs are integers, perturbing cost of edge $e_i$ by $i / n^3$

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```java
boolean less(i, j) {
    if      (cost(e_i) < cost(e_j)) return true
    else if (cost(e_i) > cost(e_j)) return false
    else if (i < j)               return true
    else            return false
}
```
Implementation: Kruskal’s Algorithm

**Implementation.** Use the **union-find** data structure.

- Build set $T$ of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.

\[
m \leq n^2 \Rightarrow \log m \text{ is } O(\log n) \quad \text{essentially a constant}
\]

```plaintext
Kruskal(G, c) {
    Sort edges weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
    $T \leftarrow \emptyset$

    foreach (u ∈ V) make a set containing singleton u

    for i = 1 to m
        (u,v) = $e_i$
        if (u and v are in different sets) {
            $T \leftarrow T \cup \{e_i\}$
            merge the sets containing u and v
        }
        are u and v in different connected components?

    return T
    merge two components
}
```
Union-Find Operations

- **Make(v)**
  - Add a new singleton set \{v\} with name v (canonical element)
  - **Kruskal:** Sets represent connected components. Initially each node is in its own connected component.

- **Find(u)**
  - **Input:** \(u \in S\)
  - **Output:** Name of the set A containing u
  - **Require:** Find(u)=Find(v) if and only if \(u \& v\) in the same set A then (Kruskal: used to test if \(u\) and \(v\) are in same connected component)

- **Union(A,B)**
  - **Input:** Names of sets A and B in the Union-Find data structure
  - **No Output:** Merge the sets A and B into a single set \(A \cup B\)
  - **Require:** If we had \(u \in A\) and \(v \in B\) then we require that Find(u)=Find(v) after this operation is completed (Kruskal: used to merge connected components after adding edge \((u,v)\))
Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

- Make(.) called $O(n)$ times (O(1) per operation with right implementation)
- Union(.) called $O(n)$ times (O(1) per operation with right implementation)
- Find(.) called $O(m)$ times (O($\alpha(m, n)$) per op with right implementation)

essentially a constant

Kruskal(G, c) {
    Setup Union-Find run Make(v) for each node v
    Sort edges weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
    $T \leftarrow \phi$

    foreach $(u \in V)$ make a set containing singleton u

    for $i = 1$ to $m$
        $(u,v) = e_i$
        if (Find(u) $\neq$ Find(v)) {
            $T \leftarrow T \cup \{e_i\}$
            Union(u,v)
        }
    return $T$
}
Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

- Build set $T$ of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.

\[
m \leq n^2 \Rightarrow \log m \text{ is } O(\log n) \quad \text{essentially a constant}
\]

```latex
Kruskal(G, c) \{
    \text{Initially each node is in own connected component}
    \text{Setup Union-Find run Make(v) for each node v}
    \text{Sort edges weights so that } c_1 \leq c_2 \leq \ldots \leq c_m.
    T \leftarrow \emptyset
    \text{foreach } (u \in V) \text{ make a set containing singleton u}
    \text{for } i = 1 \text{ to } m \text{ are u and v in different connected components?}
    (u,v) = e_i \text{ if } (\text{Find}(u) \neq \text{Find}(v)) \{
    T \leftarrow T \cup \{e_i\}
    \text{Union}(u,v)
    \}
    \text{return } T
\}
```
Union Find Data-Structure

Represent sets as rooted trees

**Example:** Tree rooted at $x$ below represents the set \{x,u,w,v,y\}

**Canonical Element:** the root of the tree
(u and w are in the same set if and only if they have the same root)
SetupUnionFind(S)
Initialization: S={1,...,n}

List<Node> Sets;

SetupUnionFind(n)

for (i=1 to n) {
    Node v;
    v.Index = i;
    v.Size = 1;
    v.Parent = null;
    Sets.Add(v)
}

struct Node {
    int Index;
    Node * Parent;
    int Size;
}
Union-Find Implementation

MakeUnionFind(S)
Initialization: S={1,...,n}

Node 1
1 1 null

Node 2
2 1 null

Node n
n 1 null

Pointers to parent

Size of set

node Find(v) {
    if (v.parent == null)
        return v
    else
        vRoot = Find(v.parent)
        return vRoot
}

26
node Find(v) {
    if (v.parent == null)
        return v
    else
        vRoot = Find(v.parent)
        return vRoot
}

Example: Find(v) = x

Running time upper bounded by height of the tree
Union-Find Implementation

```
Union(Node u, Node v) {
    uRoot = Find(u),  vRoot=Find(v)
    if (uRoot==vRoot) return
    else if (uRoot.size > vRoot.size)
        vRoot.Parent = uRoot; uRoot.size+= vRoot.size;
    else
        uRoot.Parent = vRoot; vRoot.size+= uRoot.size;
}
```

Example: Union(u,v)

```
x  3  null
   ↑
 u  1  
  
 v  2  null
   ↑
 w  1  
  
y  1  
```

uRoot is new root of Merged set
vRoot is new root of Merged set
Union-Find Implementation

Example: Union(u, v)

Union(Node u, Node v) {
    uRoot = Find(u), vRoot = Find(v)
    if (uRoot = vRoot) return
    else if (uRoot.size > vRoot.size)
        vRoot.Parent = uRoot; uRoot.size += vRoot.size;
    else
        uRoot.Parent = vRoot; vRoot.size += uRoot.size;
}

Union by Size:
Height of tree bounded by O(log n)
Path Compression

```java
node Find(v) {
    if (v.parent == null)
        return v
    else
        vRoot = Find(v.parent)
        v.Parent = vRoot;
    return vRoot
}
```

Example: Find(y)
Path Compression

node Find(v) {
    if (v.parent == null)
        return v
    else
        vRoot = Find(v.parent)
        v.Parent = vRoot;
        return vRoot
}

Example: Find(y) - every node on path from y to root x now points directly to x
Union Find: Running Time

(Path Compression + Union by Size)

- Union: $O(1)$ per operation
- Find:
  - Worst Case: $O(\log n)$ per operation
  - Amortized: $O(\alpha(n))$ per operation

- $\alpha(n)$ - Inverse Ackermann Function
  (Grows Incredibly Slowly)

- $\alpha(n) \leq 5$ for any value of $n$ you will ever use!

- Could achieve same result with union by rank
  (height of tree)
EXAMPLE
[Can weighted Union produce this tree?]

Tree before and after a Find(9) with path compression
MST Algorithms: Theory

Deterministic comparison based algorithms.

- \(O(m \log n)\)  
  [Jarník, Prim, Dijkstra, Kruskal, Boruvka]
- \(O(m \log \log n)\).  
  [Cheriton-Tarjan 1976, Yao 1975]
- \(O(m \beta(m, n))\).  
  [Fredman-Tarjan 1987]
- \(O(m \log \beta(m, n))\).  
  [Gabow-Galil-Spencer-Tarjan 1986]
- \(O(m \alpha(m, n))\).  
  [Chazelle 2000]

Holy grail.  \(O(m)\).

Notable.

- \(O(m)\) randomized.  
  [Karger-Klein-Tarjan 1995]
- \(O(m)\) verification.  
  [Dixon-Rauch-Tarjan 1992]

Euclidean.

- 2-d: \(O(n \log n)\).  
  compute MST of edges in Delaunay
- \(k\)-d: \(O(kn^2)\).  
  dense Prim