COMP 355
Advanced Algorithms

Graphs: Topological Sort
Section 22.3-22.5 (CLRS): Chapter 3 (KT)
Minimum Spanning Trees
Sections 23.1-23.2 (CLRS): Sections 4.5 (KT)
Graph Search Algorithms

BFS and DFS almost the same for directed and undirected graphs

BFS on directed graphs: still $O(m + n)$

- It is possible for node $s$ to have a path to a node $t$ even though $t$ has no path $s$
- Computing the set of all nodes $t$ with the property that $s$ has a path to $t$

DFS on directed graphs: still $O(m + n)$

- At node $u$, recursively launches depth-first search, in order, for each node to which $u$ has an edge
**Strong Connectivity**

**Def.** Node $u$ and $v$ are mutually reachable if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

**Def.** A directed graph is strongly connected if every pair of nodes is mutually reachable.

**Lemma.** Let $s$ be any node. $G$ is strongly connected iff every node is reachable from $s$, and $s$ is reachable from every node.

- Pf. $\Rightarrow$ Follows from definition.
- Pf. $\Leftarrow$ Path from $u$ to $v$: concatenate $u$-$s$ path with $s$-$v$ path. Path from $v$ to $u$: concatenate $v$-$s$ path with $s$-$u$ path. ▪

ok if paths overlap

1/27/2014
**Strong Connectivity: Algorithm**

**Theorem.** Can determine if G is strongly connected in $O(m + n)$ time.

**Pf.**

- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{\text{rev}}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma.

![Diagram showing strongly connected and not strongly connected graphs](image-url)
**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 \(v_1, v_2, ..., v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
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 \(v_j\).
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

- Suppose that $G$ has a topological order $v_1, ..., v_n$ 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 $v_1, ..., v_n$ is a topological order, we must have $j < i$, a contradiction.
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.
Directed Acyclic Graphs

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
Topological Ordering Algorithm: Example

Topological order:
Topological Ordering Algorithm: Example

Topological order: $v_1$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3$
Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4 \)
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5$
Topological Ordering Algorithm: Example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$
Topological Ordering Algorithm: Example

Topological order: \( v_1, v_2, v_3, v_4, v_5, v_6, v_7 \).
Topological Sorting Algorithm: Running Time

**Theorem.** Algorithm finds a topological order in $O(m + n)$ time.

**Pf.**
- Maintain the following information:
  - $\text{count}[w] =$ remaining number of incoming edges
  - $S =$ 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
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 $K_n$. 

$G = (V, E)$

$T$, $\sum_{e \in T} c_e = 50$

Can't solve by brute force
MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.
MST Problem

Given a connected, undirected graph $G = (V, E)$, a spanning tree is an acyclic subset of edges $T \subseteq E$ that connects all the vertices together.

We define the cost of a spanning tree $T$ to be the sum of edges in the spanning tree

$$w(T) = \sum_{(u,v) \in T} w(u, v).$$

A minimum spanning tree (MST) is a spanning tree of minimum weight.
MST Problem

- Three spanning trees for the same graph
- (a) is not a MST
- (b) and (c) are both MSTs
Greedy Algorithms

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.
**Def.** An undirected graph is a *tree* if it is connected and does not contain a cycle.

**Theorem.** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.

- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
**MST Terms**

**Def.** We say that a subset $A \subseteq E$ is *viable* if $A$ is a subset of edges in some MST.

**Def.** We say that an edge $(u, v) \in E \setminus A$ is *safe* if $A \cup \{(u, v)\}$ is viable. ($E \setminus A$ means the edges of $E$ that are not in $A$.)

**When is an edge safe?**

Let $S$ be a subset of the vertices $S \subseteq V$.

- A cut $(S, V \setminus S)$ is a partition of the vertices into two disjoint subsets (a)
- An edge $(u, v)$ crosses the cut if $u \in S$ and $v \notin S$ (b)
- Given a subset of edges $A$, we say that a cut respects $A$ if no edge in $A$ crosses the cut (c)
Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

Cycle property. Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. 
**Cycles and Cuts**

**Cycle.** Set of edges the form a-b, b-c, c-d, ..., y-z, z-a.

![Graph showing a cycle](image1)

Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

**Cutset.** A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.

![Graph showing a cutset](image2)

Cut S = {4, 5, 8}
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8
Claim. A cycle and a cutset intersect in an even number of edges.

Pf. (by picture)
Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$.

Pf. (exchange argument)
• Suppose $e$ does not belong to $T^*$, and let's see what happens.
• Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$.
• Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$ $\Rightarrow$ there exists another edge, say $f$, that is in both $C$ and $D$.
• $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
• Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
• This is a contradiction.
Cycle Property Proof

Simplifying assumption. All edge costs $c_e$ are distinct.

Cycle property. Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$.

Pf. (exchange argument)
• Suppose $f$ belongs to $T^*$, and let's see what happens.
• Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$.
• Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$ $\Rightarrow$ there exists another edge, say $e$, that is in both $C$ and $D$.
• $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
• Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
• This is a contradiction.
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.

![Case 1](image1.png)
![Case 2](image2.png)
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.

$m \leq n^2 \Rightarrow \log m$ is $O(\log n)$ essentially a constant

KruskalMST($G=(V,E), w$) {
    $A = \{}$
    \hspace{1cm} // initially $A$ is empty
    Place each vertex $u$ in a set by itself
    Sort $E$ in increasing order by weight $w$
    for each $((u, v) \text{ in this order})$
    \hspace{1cm} if (find(u) != find(v)) {
        \hspace{1.5cm} // $u$ and $v$ in different trees
        \hspace{1.5cm} add $(u, v)$ to $A$
        \hspace{1.5cm} \hspace{1cm} // join subtrees together
        \hspace{1.5cm} union(u, v)
        \hspace{1.5cm} \hspace{1cm} // merge these two components
    }
    return $A$
}
Kruskal’s Algorithm Example

Fig. 21: Kruskal’s Algorithm. Each vertex is labeled according to the set that contains it.
Kruskal’s Algorithm Demo

The diagram shows a network with labeled nodes and edges.

Nodes: 1, 2, 3, 4, 5, 6, 7, 8, 9
Edges: 1-5, 5-7, 7-6, 6-4, 4-8, 8-3, 3-1, 1-9, 9-2, 2-6
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Kruskal’s Algorithm Demo
Next Time

• Prim and Boruvka’s Algorithms for MST
• Section 23.2 (CLRS)
• Section 4.5(KT)