References for Today’s Lecture

- Required reading
  - Chapter 20

- References
  - AMO Chapter 13
  - CLRS Chapter 23
Minimum Spanning Trees

- Optimality Conditions
- Kruskal’s Algorithm
- Prim’s Algorithm
Combinatorial Optimization

• A *combinatorial optimization problem* consists of
  
  – a ground set of elements $E$,
  – an associated set $\mathcal{F}$ of subsets of $E$ called the *feasible subsets*.
  – A cost vector $\mathbb{R}^E$.

• The cost $c(S)$ of a feasible subset is $\sum_{s \in S} c_s$.

• The goal is to find a subset of minimum cost.
Minimum Spanning Trees

• Recall that a spanning tree $T$ of $G$ is a connected acyclic subgraph that spans all the nodes of $G$.

• The total cost of a spanning tree is the sum of the costs of the arcs in the tree.

• Given an undirected graph $G = (N, A)$ with $n$ nodes and $m$ arcs and with a length or cost $c_{ij}$ associated with each arc $(i, j) \in A$, the minimum spanning tree problem is to find a spanning tree with the smallest total cost (length).

• This is a combinatorial optimization problem.
Optimality Conditions

• Cut Optimality Conditions
• Path Optimality Conditions

Properties of a Spanning Tree

• For every non-tree arc \((k, l)\), a spanning tree \(T\) contains a unique path from node \(k\) to node \(l\). The arc \((k, l)\) together with the unique path defines a cycle.

• If we delete any tree arc \((i, j)\) from a spanning tree, we partition the node set into two subsets, which define a cut in the graph.
**Cut Optimality Conditions**

**Theorem 1. [13.1]** A spanning tree $T^*$ is a minimum spanning tree if and only if for every tree arc $(i, j) \in T^*$, $c_{ij} \leq c_{kl}$ for every arc $(k, l)$ contained in the cut formed by deleting arc $(i, j)$ from $T^*$.
Proof of Theorem 13.1

1. Show if $T^*$ is a MST, then $T^*$ must satisfy the Cut Optimality Conditions.

2. Show if any tree $T^*$ satisfies the Cut Optimality Conditions, then $T^*$ is a MST.
Path Optimality Conditions

**Theorem 2. [13.3]** A spanning tree $T^*$ is a MST if and only if for every non-tree arc $(k, l)$ of $G$, $c_{ij} \leq c_{kl}$ for every arc $(i, j)$ contained in the path in $T^*$ connecting nodes $k$ and $l$. 
Proof of Theorem 13.3

1. Show if $T^*$ is a MST, then $T^*$ satisfies the Path Optimality Conditions.

2. Show if for every non-tree arc $(k, l)$ of $G$ $c_{ij} \leq c_{kl}$ for every arc $(i, j)$ contained in the path in $T^*$ connecting nodes $k$ and $l$, then $T^*$ is a MST.
Algorithm Based on Cut Optimality

- Prim’s algorithm is motivated by the cut optimality conditions.
- We build up the tree one edge at a time as one connected component.
- In each iteration, we will connect one more node to the current tree.
- We do this by adding the edge that is the minimum length edge across the cut induced by the current set of connected nodes.
- Why does this guarantee optimality?
- How do we do this?
Prim’s Algorithm

**algorithm Prim**

\[ T = \emptyset \]
\[ S = \{1\}; \quad \bar{S} = N - \{1\} \]

while (|S| < n) do

- find arc \((i, j)\) in \([S, \bar{S}]\) with minimum cost
- \(T = T \cup \{(i, j)\}\)
- \(S = S \cup \{j\}; \quad \bar{S} = \bar{S} - \{j\}\)
Complexity

- Number of iterations?
- Dominant step of each iteration?
**Prim’s Algorithm**

- For each node $j \in \bar{S}$
  - $d(j) = \min$ cost of arcs in the cut incident to a node $j \notin \bar{S}$
  - $d(j) = \min\{c_{ij} : (i, j) \in [S, \bar{S}]\}$
  - pred($j$) = $i$ such that $c_{ij} = \min\{c_{ij} : (i, j) \in [S, \bar{S}]\}$.

- To find min cost arc, compute $\min\{d(j) : j \in \bar{S}\}$.

- Suppose $\hat{j}$ is the min, then (pred($\hat{j}$), $\hat{j}$) is min cost arc.

- Move $\hat{j}$ to $S$ and update distance and predecessor labels for nodes adjacent to $\hat{j}$. 
Running Time of Prim’s Algorithm

• Note that Prim’s Algorithm is a graph search procedure.

• However, the procedure for determining the search order is more complex than previous ones.

• At each step, we need to update some node labels and then be able to determine the node with the minimum label.

• The key to implementing the procedure is an efficient data structure.

• What is the running time for a naive implementation?
Implementation with Priority Queues

• The running time depends critically on how keep track of the minimum label as the algorithm progresses.

• To get a strongly polynomial time algorithm, we must use a more general data structure for maintaining a priority queue.

• For a given order set $H$, this data structure should support the operations
  
  – $\text{push}(\text{item, value})$ (to add and change value of an item)
  – $\text{peek}()$
  – $\text{pop}()$
Binary Heaps

- A binary heap is a balanced binary tree with additional structure that allows it to function efficiently as a priority queue.

- The additional structure needed to support these operations is that each node has a higher priority than either of its children.

- Balanced binary trees can be stored very efficiently in a single array.
  - The root is stored in position 0.
  - The children of the node in position $i$ are stored in positions $2i + 1$ and $2i + 2$.
  - This determines a unique storage location for every node in the tree and makes it easy to find a node’s parent and children.
  - Using an array, basic operations can be performed very efficiently.
Creating the Heap

- Any node whose priority is higher than either of its children is said to satisfy the *heap property*.

- Consider a tree in which all nodes except for the root have the heap property.

- We can easily transform this into a tree in which every node has the heap property (*how?*).

- This operation is called *heapify()*.

- By calling *heapify()* on each node, starting at the lowest level and working upward, we can transform an unordered binary tree into a heap.

- This is how we create the initial heap.

- Note that this step is unnecessary for implementing Dijkstra’s. Why?
Operations on a Heap

- The node with the highest priority is always the root.
- To change the priority of a node
- To insert a node
- To delete a node
- What are the running times of these operations?
Analyzing Prim’s with a Binary Heap
Algorithm Based on Path Optimality

- Kruskal’s algorithm motivated by path optimality conditions.
- We build up the tree one edge at a time, but this time we build multiple components simultaneously.
- In each step, we will add the minimum edge that does not form a cycle with the edges already added.
- Why does this guarantee optimality?
- How do we implement it?
Kruskal’s Algorithm

algorithm *Kruskal*

sort edges in non-decreasing order of length

LIST := ∅

while (|LIST| < |N| − 1 and ∃ unexamined edges) do
    e := unexamined edge with minimum length
    if adding e to LIST does not create a cycle
        add e to LIST
    else discard e
Kruskal’s Algorithm: Complexity

• The algorithm has two steps.
  – Sorting the edge list: \( O(m \log m) = O(m \log n) \)
  – Building the tree: ??

• To determine which edges we are allowed to add in each step requires a data structure for storing connected components.

• The data structure must support two operations.
  – \texttt{find}(i, j): Are \( i \) and \( j \) in the same component?
  – \texttt{union}(i, j): Merge the components \( i \) and \( j \).
Quick Find Implementation of Union-Find

- The simplest implementation involves an array of length $n$.

- We will maintain the array such that two items are in the same subset if and only if the array entries are equal.

- This makes the $\text{find}(i, j)$ constant time, so we call this implementation *quick find*.

- How do we implement $\text{union}(i, j)$?

- What is the running time?

- Note that this could also be implemented using linked lists.
Quick Union Implementation of Union-Find

- To speed up the union operation, we maintain the array in a different fashion.
- We will consider the $i^{th}$ entry of the array to be a pointer to another item.
- To perform $\text{find}(i, j)$,
  - Follow the pointers from nodes $i$ and $j$ until reaching a node that points to itself, called the representative.
  - If the same representative is reached from both nodes $i$ and $j$, then they are in the same subset.
- To perform $\text{union}(i, j)$, perform the find operation and then point the representative for $i$ to the representative for $j$.
- What is the performance now?
Weighted Quick Union

- Note that the quick union algorithm essentially builds a tree out of the nodes in each component, with the root being the representative.
- As in a heap, the running time of the find operation depends on the depth of the trees.
- Each union operation essentially connects two trees together by pointing the root of one tree to the root of the other.
- One way to limit the depth of the tree is to always point the smaller tree to the larger one.
- This ensures that each find takes less than $\log n$ steps.
- Note that we must now keep track of the number of nodes in each tree, but that’s easy to do.
- Another approach is to keep track of the height of each tree and always point the shorter tree to the taller one.
Path Compression

- Ideally, we would like each item to point directly to the representative of its subset.

- One possibility is to simply keep track of all the nodes encountered in the path to the root.

- After reaching the root, set all the nodes on the path to point to the root.

- This is easy to implement recursively and doesn’t change the asymptotic running time.

- An easier method to implement is *compression by halving*, which is setting each node to point to its grandparent.

- Combining weighted quick union with path compression yields a total running time for connected components of approximately $O(m)$. 
Analyzing Kruskal’s with Optimized Union-Find