Goal: MST cut and cycle properties → Prim, Kruskal greedy algorithms
Spanning Tree vs. Steiner Tree
Spanning Trees vs. Steiner Trees

Min-Cost Spanning Tree Problem: “Easy”, solvable with a greedy algorithm
Min-Cost Steiner Tree Problem: “Hard”, no known polynomial-time algorithm

- The red dot is called a Steiner point (= an “intermediate junction”)
- The maximum cost savings from adding Steiner points
  = min ratio of (min Steiner tree cost) / (min spanning tree cost)
  depends on the metric, or distance function
  - Cf. “Manhattan”, “Euclidean”, “Chebyshev”, ...

Adapted from Prof. G. Robins, UVA
Minimum Steiner Tree is a Hard Problem
(Minimum Spanning Tree was Easy)

• How would you approach finding a low-cost Steiner tree?

Adapted from Prof. G. Robins, UVA
Iterated 1-Steiner Algorithm (1990)

Given a pointset $S$, what point $p$ minimizes $\text{MST}(S \cup \{p\})$

Algorithmic idea: Iterate (greedily)

In practice: solution cost is within 0.5% of $\text{OPT}$ on average

Adapted from Prof. G. Robins, UVA
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 $T$.

Greedy Algorithms

- **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 of the form a-b, b-c, c-d, …, y-z, z-a.

  ![Diagram of a cycle]

  Cycle $C = 1\rightarrow 2, 2\rightarrow 3, 3\rightarrow 4, 4\rightarrow 5, 5\rightarrow 6, 6\rightarrow 1$

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

  ![Diagram of a cut]

  Cut $S = \{ 4, 5, 8 \}$
  Cutset $D = 5\rightarrow 6, 5\rightarrow 7, 3\rightarrow 4, 3\rightarrow 5, 7\rightarrow 8$

Illustration
Intersection of a Cycle and a Cutset

- **Fact.** A *cycle* and a *cutset* intersect in an even number of edges.  Why?

  Cycle  $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
  Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
  Intersection = $3-4, 5-6$

- **Picture:**

Toward Greedy MST Algs: Cut Property

• Simplifying assumption: All edge costs $c_e$ 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 $e$.

Toward Greedy MST Algs: Cut Property

- **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 $e$.

- **Proof.** (EXCHANGE ARGUMENT)
  - Assume toward a contradiction that $e$ does not belong to $T^*$
    - 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 $S$  
    - there exists another edge, say $f$, that is in both $C$ and $D$
  - $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree
  - Since $c_e < c_f$, cost($T'$) < cost($T^*$)
  - This contradicts the assumption  
    - $e$ must be in $T^*$

Toward Greedy MST Algs: Cycle Property

• Simplifying assumption: all edge costs $c_e$ 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 $f$.

Toward Greedy MST Algs: Cycle Property

• Simplifying assumption: all edge costs \( c_e \) 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 \( f \).

Toward Greedy MST Algs: Cycle Property

- **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.

- **Proof. (EXCHANGE ARGUMENT)**
  - Assume toward a contradiction that f belongs to T*.
    ⇒ 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
    ⇒ there exists another edge, say e, that is in both C and D
  - T' = T* ∪ \{e\} - \{f\} is also a spanning tree
  - Since \(c_e < c_f\), \(\text{cost}(T') < \text{cost}(T^*)\)
  - This contradicts the assumption  ⇒ f must not be in T*

Prim’s Algorithm and Correctness

- Prim's algorithm [Jarník 1930, Dijkstra 1957, Prim 1959]
  - Initialize $S = \text{any node}$
  - Apply cut property to $S$
  - Add min cost edge in cutset corresponding to $S$ to $T$, and add one new explored node $u$ to $S$
Implementation of Prim’s Algorithm

- **Implementation**: use a priority queue (as in Dijkstra)
  - Maintain set of explored nodes $S$ // ~ set “R” in Dijkstra
  - For each unexplored node $v$, maintain attachment cost $a[v] = \text{cost of cheapest edge } v \text{ to a node in } S$  // ~ temporary label in Dijkstra
  - Complexity: $O(E \log V)$ with a binary heap

```plaintext
Prim(G, c) {
    foreach (v ∈ V) a[v] ← ∞  
    Initialize an empty priority queue Q  
    foreach (v ∈ V) insert v onto Q  
    Initialize set of explored nodes $S ← ∅$

    while (Q is not empty) {
        u ← delete min element from Q  
        S ← S ∪ { u }  
        foreach (edge e = (u, v) incident to u)
            if (((v \notin S) and (c_e < a[v])))
                decrease priority a[v] to c_e
    }
}
```

Kruskal’s Algorithm and Correctness

- Kruskal’s algorithm. [Boruvka, 1926; Kruskal, 1956]
  - Consider edges in ascending (i.e., sorted) 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

Example: Kruskal’s Algorithm
The Union-Find Problem

• Need a fast way to test if adding $e_i$ to $T$ creates a cycle
• At $i^{th}$ iteration, $T$ is a set of trees
  – Initially, each tree contains one node
  – Adding $e_i$ is okay unless $u$ and $v$ are in the same tree

• “Disjoint set UNION-FIND” data structure to support:
  – Make-Set($u$): creates a set containing $u$ (for initialization)
  – Find-Set($u$): Return representative element of set that contains $u$
  – Union($u$, $v$): Merge the sets containing $u$ and $v$ (and choose new representative)

• Vertices of the graph = elements to be stored in the sets

Source: slides from Prof. L. Carter, UCSD 2002
Algorithms for Union-Find

• Approach #1: Define array A with A[u] = representative of u
  – Find-Set(u): return A[u] \(\rightarrow O(1)\) time
  – Union(u,v): Let x = A[u], y = A[v]; change all x’s to y’s in A
    \(\rightarrow \Omega(|V|)\) time

Source: slides from Prof. L. Carter, UCSD 2002
Algorithms for Union-Find

• Approach #1: Define array A with A[u] = representative of u
  – Find-Set(u): return A[u] \( \Rightarrow O(1) \) time
  – Union(u,v): Let x = A[u], y = A[v]; change all x’s to y’s in A
    \( \Rightarrow \Omega(|V|) \) time

• Approach #2: Also keep a list of members for each set
  – Find-Set(u): return A[u] \( \Rightarrow O(1) \) time
  – Union(u,v): For each member z of u’s list, add it to v’s list, set A[z] = y
  – Worst case: ???
  – Key idea: Move elements of smaller list to larger
  – No element moves more than \( \lg |V| \) times \( // \) remember, \( \lg = \log \text{ base 2} \)
  – Even if Union can take \( O(|V|) \) time, doing \( |V| \) unions takes \( O(|V| \lg |V|) \) time
    \( \Leftarrow \) “amortized analysis”

• Kruskal: \( \Theta(|E| \lg |E|) \) time to sort edges;
  \( |E| \) Find-Set tests;
  \( |V| \) “\( T \cup \{e\} \)” (Union) operations \( \Rightarrow \Theta(|E| \lg |V|) \)

\( // \) Why is \( (E \lg E) \) the same as \( (E \lg V) \) ???

Source: slides from Prof. L. Carter, UCSD 2002
Implementation of Kruskal's Algorithm

- Use the **union-find** data structure
  - Build set T of edges in the MST
  - Maintain set for each connected component

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

    foreach (u \in 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
        }
    return T
}
```

EXTRA MATERIAL: DFS in MST

== A Heuristic for the Geometric Traveling Salesperson Problem
Euclidean TSP

• **Euclidean Traveling Salesman Problem**: Let $C_1, C_2, \ldots, C_n$ be a set of points in the plane. Find a tour (= cycle) over all $n$ points with minimum total edge cost.

• **Fact**: $\text{cost}(\text{MST}) < \text{cost}(\text{Tour}_{\text{opt}})$
  - (1) MST is the minimum-cost graph that connects all vertices, and has only $n-1$ edges
  - (2) Any TSP tour must also connect all vertices, and will have $n$ edges

    • **Note**: a tour = a spanning tree (not necessarily a minimum spanning tree) plus another edge
Approximation Algorithm : Euclidean TSP

– Idea: Consider the tour that consists of a DFS traversal of an MST (starting from any city).
– If the tour is about to revisit a city, then skip that city (== “shortcut to the next city”)

– Key observation: If the triangle inequality holds, the shortcuts can only reduce the tour cost!
Approximation Algorithm: Euclidean TSP

DSF traversal of MST

Taking shortcut from DSF tour. (e.g. replacing a-b-c-b-a-d, by a-b-c-d)

\[ \text{Tour}_{\text{Heur}} \leq 2 \times \text{MST} \leq 2 \times \text{Tour}_{\text{opt}} \]
Concept of “Approximation” Algorithm

- **Approximation algorithm**: An algorithm that returns *near-optimal* solutions (i.e. is "provably good") is called an *approximation algorithm*.

- **Performance Ratio**: An approximation algorithm for a problem has a *performance ratio* of $\rho(n)$ if for any instance $I$ of size $n$, the cost $C$ of the solution produced by the approximation algorithm is within a factor of $\rho(n)$ of the cost $C^*$ of an optimal solution:

$$\max_{|I|=n} \left( \frac{C}{C^*} \right) \leq \rho(n)$$