Minimum Spanning Trees

Slides based on Kevin Wayne / Pearson-Addison Wesley

Graphs and Trees

Def. A graph $G = (V,E)$ is a collection of vertices (nodes) and edges

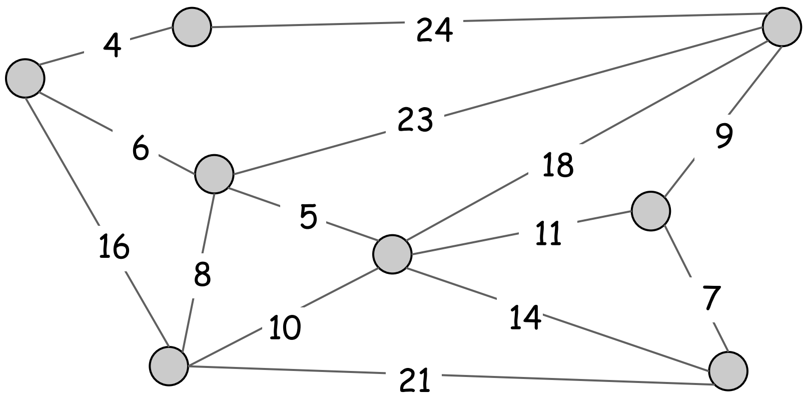
Def. An undirected graph $G = (V,E)$ is a tree if it is connected and does not contain a cycle.

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

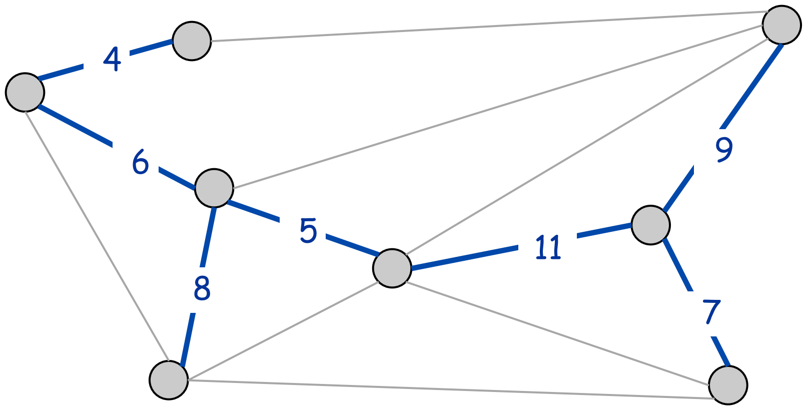
- In other words, $G' = (V,T)$ is a tree
- A given graph $G = (V,E)$ can have many possible spanning trees
- If $|V| = n$, there are $n-1$ edges in a spanning tree

Minimum Spanning Tree

Minimum spanning tree. Given a connected graph $G = (V, E, w)$ with real-valued edge weights w_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.



$G = (V, E, w)$



$T, \ \Sigma_{e \in T} w_e = 50$

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

$\uparrow$

can't solve by brute force

Greedy Algorithms

Strategy 1. Start with $T = \phi$. Consider edges in ascending order of weight. Insert edge e in T unless doing so would create a cycle.

Strategy 2. 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. (abuse of notation; T refers to edges but we're using it here to refer to the associated vertices as well)

Strategy 3. Start with T = E. Consider edges in descending order of weight. Delete edge e from T unless doing so would disconnect T.

Greedy Algorithms

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

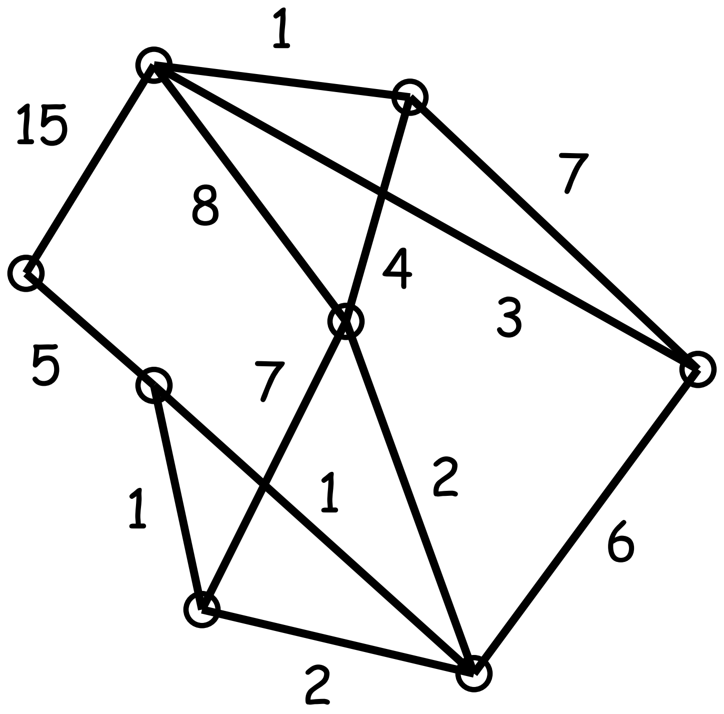
Prim's algorithm. Start with some root node s and greedily grow a tree T from s outward. At each step, add the cheapest edge e to T that has exactly one endpoint in T.

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

Remark. All three algorithms produce an MST.

Kruskal's Algorithm

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

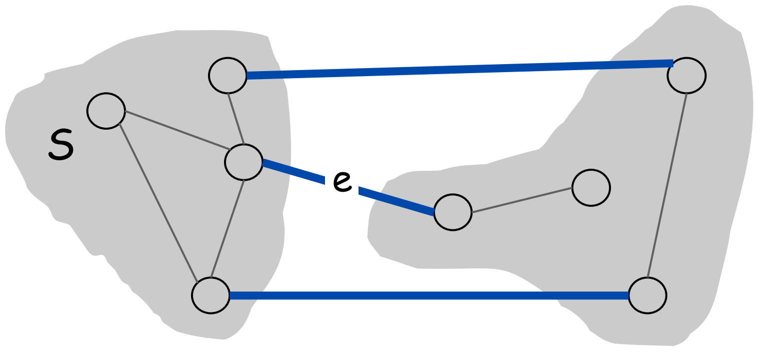


```
Kruskal ( G=(V,E,w) )
1. Let T = Ø;
2. Sort the edges in increasing order of weight
3. For edge e do
4.     If T U e does not contain a cycle then
5.         Add e to T
6. Return T
```

Kruskal - Proof of Correctness

Simplifying assumption. All edge weights w_e are distinct.

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



e is in the MST

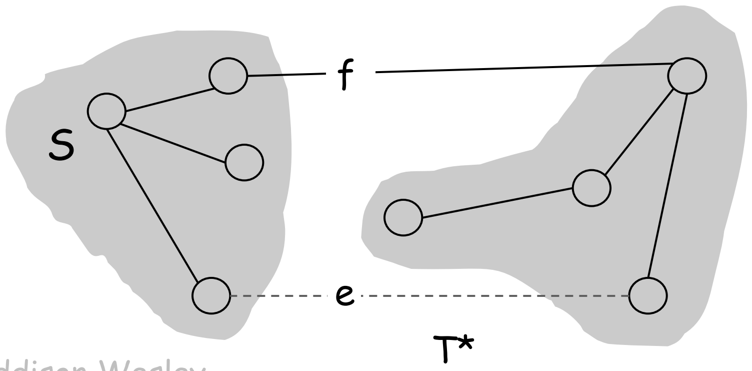
Kruskal – Proof of Correctness

Simplifying assumption. All edge weights w_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min weight edge with exactly one endpoint in S. Then any 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*.
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $w_e < w_f$, weight(T') < weight(T*).
- This is a contradiction. ▪



Kruskal – Proof of Correctness

Claim. Kruskal's Algorithm produces an MST of $G = (V,E)$

Pf.

- First, argue that every edge added by Kruskal's Algorithm belongs to every MST.
  - Consider any edge $e = (v,w)$ being added.
  - Define S as the set of nodes reachable from v just before e is added. Then $v \in S$, and $w \in V-S$. (Else e would create a cycle).
  - Edge e must be cheapest among all edges from S to $V-S$, else the algorithm would have considered another such edge earlier.
  - By the cut property, e is part of every MST.

- Second, argue that the output (V,T) of Kruskal's Algorithm includes enough edges to be a spanning tree of G.
  - By definition of Kruskal Algorithm, (V,T) has no cycles.
  - Also, (V,T) is connected, else the algorithm would not have ended yet.

Implementation: Kruskal's Algorithm

Implementation: Basic Version.

- Sort the edges by weight. O(m log m)
- Process the edges one at a time
  - Check to see if adding the edge e = (v,w) results in a cycle
  - Use BFS from starting node v
    - ✎ BFS is O(m + n)
    - ✎ More accurately in this case O(n + n) = O(n) because the graph being searched never has more than n-1 edges.

- Overall complexity: O(mn)

- Ultimate goal: O(m)
  - We'll be satisfied with O(m log n)

Implementation: Kruskal's Algorithm

Implementation: Union-Find Data Structure.

- Basic idea: we don't want to have to re-compute the connected components (to check for a cycle) each time an edge is added.
- Instead, if we efficiently keep track of what connected component any node belongs to, we can easily identify whether an edge under consideration should be included. For edge e = (v,w):
  - If v and w belong to different connected components, then edge e can be added without creating a cycle
  - If v and w belong to the same connected component, there already exists a path from v to w, so adding e would create a cycle.

Implementation: Kruskal's Algorithm

Implementation: Union-Find Data Structure.

- Support 3 basic operations:
  - Init(S): for a set S returns an initialized Union-Find data structure in which all elements are in separate sets (their own connected components).
  - Find(u): returns the set S containing u
  - Union(A,B): changes the data structure by merging the sets A and B into a single set. (Note we'll actually implement the function so that it takes individual nodes as parameters and forms the union of the sets to which those nodes belong.)

Kruskal's Algorithm

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

```
Kruskal ( G=(V,E,w) ) using Union-Find data structure
1. Let T = Ø;
2. Sort the edges in increasing order of weight
3. Init(V)    # initialize union-find arrays
4. For edge e = (u,v) do
5.    If Find(u) ≠ Find(v) # no cycle in T ∪ e
6.       Add e to T
7.       Union(u,v) # really union of sets containing u, v
8. Return T
```

Kruskal's Algorithm

```
Init (V)
1. for every vertex v do
2.    boss[v]=v
3.    size[v]=1
4.    set[v]={v}

Find (u)
1. Return boss[u]

Union (u,v)
1. if size[boss[u]]>size[boss[v]] then
2.    set[boss[u]]=set[boss[u]] union set[boss[v]]
3.    size[boss[u]]+=size[boss[v]]
4.    for every z in set[boss[v]] do
5.         boss[z]=boss[u]
6. else do steps 2.-5. with u,v switched
```

Kruskal's Algorithm

Complexity Analysis for $G = (V,E)$, $|V| = n$, $|E| = m$

- Sort the edges: $O(m \log m) = O(m \log n^2) = O(m \log n)$
- Init. Union-Find Data Structures: $O(n)$
- All Find(u) operations together: $O(m)$
- All Union(u,v) operations together: $O(n \log n)$ * see next slide

- Total Complexity: $O(m \log n)$
  - Complexity dominated by initial sorting of edges by weight

```
Kruskal ( G=(V,E,w) ) using Union-Find data structure
1. Let T = Ø;
2. Sort the edges in increasing order of weight
3. Init(V)    # initialize union-find arrays
4. For edge e = (u,v) do
5.     If Find(u) ≠ Find(v) # no cycle in T ∪ e
6.         Add e to T
7.         Union(u,v) # really union of sets containing u, v
8. Return T
```

Kruskal's Algorithm

Complexity Analysis for G = (V,E), |V| = n, |E| = m

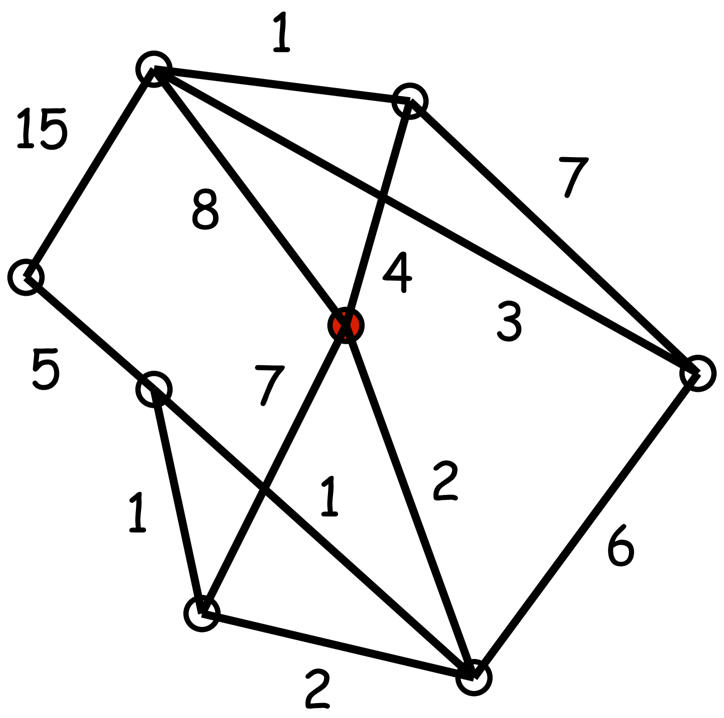
- All Union(u,v) operations together: O(n log n)
  - Cost of union operation dominated by updating boss[]
  - How many total boss updates are done across all unions?

  - Because we always choose to update the smaller set, it guarantees that any time a vertex has its boss updated, the set it belongs to at least doubles in size
    - ✐ Each vertex has its boss updated at most log n times

```
Union (u,v)
1. if size[boss[u]]>size[boss[v]] then
2.    set[boss[u]]=set[boss[u]] union set[boss[v]]
3.    size[boss[u]]+=size[boss[v]]
4.    for every z in set[boss[v]] do
5.       boss[z]=boss[u]
6. else do steps 2.-5. with u,v switched
```

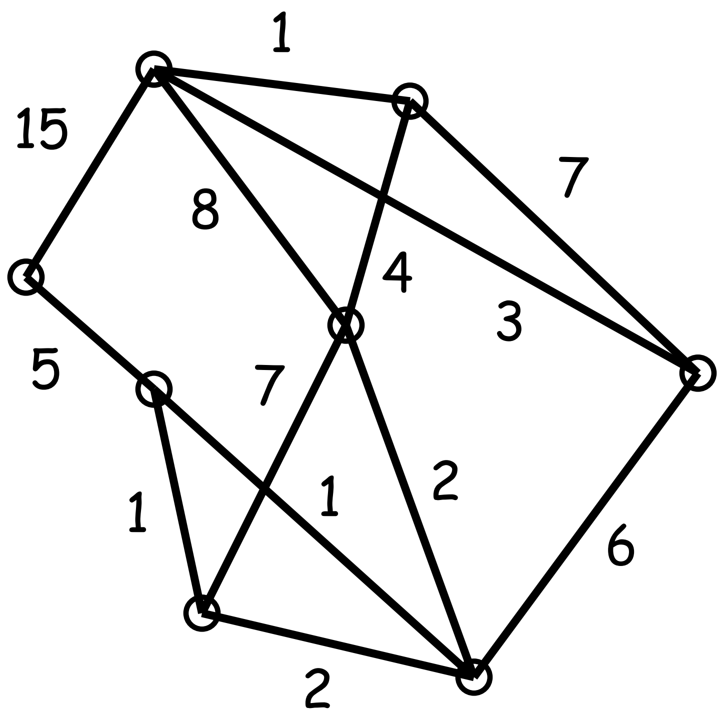
Prim's Algorithm

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.



Prim's Algorithm

```
Prim ( G=(V,E,w) )
1. Let T = Ø; H = V - {s}
2. For every vertex v in H do
3.     cost[v]= ∞, parent[v]=null
4. cost[s]=0, parent[s] = none
5. Update (s)
6. For i=1 to n-1 do
7.     u=vertex from H of
          smallest cost #extract_min
•      Add (u,parent[u]) to T
•      Update(u)
• Return T
```



```
Update (u)
1. For every neighbor v of u (such that v in H)
2.     If cost[v]>w(u,v) then
3.         cost[v]=w(u,v), parent[v]=u    # change_key
```

Prim – Proof of Correctness

Claim. Prim's Algorithm produces an MST of G = (V,E).

Pf. (assuming distinct edge weights)

- First, argue that every edge added by Prim's Algorithm belongs to every MST.
  - By definition of the algorithm, each edge added is the minimum weight edge between the current partial spanning tree S and the remaining vertices V - S
  - By the cut property, e is part of every MST.

- Second, argue that the output (V,T) of Prim's Algorithm includes enough edges to be a spanning tree of G.
  - By definition of algorithm, (V,T) has no cycles as each edge added connects to an as yet unconnected vertex
  - Also, (V,T) is connected, else the algorithm would not have ended yet.

Prim – Complexity

Complexity Analysis for G = (V,E), |V| = n, |E| = m

- We store attachment weight information for each vertex in a heap data structure
  - Each heap operation requires O(log n) time
- n-1 iterations in which an EXTRACT_MIN heap operation is performed
  - O(n log n)
- Each edge can result in at most one CHANGE_KEY heap operation
  - O(m log n)
- Overall complexity: O(m log n)

Prim – Complexity

Complexity Analysis for $G = (V,E)$, $|V| = n$, $|E| = m$

- Alternatively, we store edge information in an adjacency matrix and store attachment weight information for each vertex in a separate list

- n iterations in which the vertex with smallest attachment weight is chosen:
  - $O(n^2)$

- In each iteration, neighbors of the added vertex may have their attachment weight updated:
  - $O(n^2)$

- Overall complexity: $O(n^2)$
  - Note: this is actually better than using a heap in the case that G is dense ($m \approx n^2$)

Lexicographic Tiebreaking

To remove the assumption that all edge weights are distinct: perturb all edge weights 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 weights is MST with original weights.

$\uparrow$ e.g., if all edge weights are integers, perturbing weight of edge e_i by i / n^2

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

```
boolean less(i, j) {
   if      (weight(e_i) < weight(e_j)) return true
   else if (weight(e_i) > weight(e_j)) return false
   else if (i < j)                     return true
   else                                return false
}
```